Fine-Grained Characterization of Faults Causing Long Latency Crashes in Programs

Guanpeng Li, Qining Lu and Karthik Pattabiraman
Department of Electrical and Computer Engineering
University of British Columbia (UBC), Vancouver
{gpli, qining, karthikp}@ece.ubc.ca

***Abstract*—As the rate of transient hardware faults increases, researchers have investigated software techniques to tolerate these faults. An important class of faults are those that cause long-latency crashes (LLCs), or faults that can persist for a long time in the program before causing it to crash. In this paper, we develop a technique to automatically find program locations where LLC causing faults originate so that the locations can be protected to bound the program's crash latency.**

We first identify program code patterns that are responsible for the majority of LLC causing faults through an empirical study. We then build CRASHFINDER, a tool that finds LLC locations by statically searching the program for the patterns, and then refining the static analysis results with a dynamic analysis and selective fault injection-based approach. We find that CRASHFINDER can achieve an average of 9.29 orders of magnitude time reduction to identify more than 90% of LLC causing locations in the program, compared to exhaustive fault injection techniques, and has no false-positives.

Keywords: Long-latency Crashes, Hardware Faults, Checkpoint Corruption

I. INTRODUCTION

Transient hardware fault rates are predicted to increase in future computer systems due to the effects of technology scaling, manufacturing variations and diminishing voltage margins [3], [7]. In the past, such faults were masked through hardware-only solutions such as redundancy or guard banding. However, such techniques are becoming increasingly challenging to deploy as they consume significant amounts of energy, and energy is becoming a first class constraint in microprocessor design [4]. As a result, many researchers have postulated that future processors will expose hardware faults to the software, and will expect the software to tolerate them [18], [16], [23]. Future software systems therefore will need to be capable of detecting hardware faults and recovering from them.

A hardware fault can have many possible effects on a program. First, it may be masked or be benign. In other words, the fault may have no effect on the program's final output. Second, it may cause a crash (i.e., hardware exception) or hang (i.e., program time out). Finally, it may cause Silent Data Corruptions (SDCs), or the program producing incorrect outputs. Of the above outcomes, SDCs are considered the most severe, as there is no visible indication that the application has done something wrong. Therefore, a number of prior studies have focused on detecting SDC-causing program errors, by selectively identifying and protecting elements of program state that are likely to cause SDCs [10], [8], [26], [17].

Compared to SDCs, crashes have received relatively less attention from the perspective of error detection. This is because crashes are considered to be the detection themselves, as the program can be recovered from a checkpoint (if one exists) or restarted after a crash. However, all of these studies make an important assumption, namely that the crash occurs soon after the fault is manifested in the program. This is important to ensure that the program is prevented from writing corrupted state to the file system (e.g., checkpoint), or sending wrong messages to other processes [1]. While this assumption is true for a large number of faults, studies have shown that a small but non-negligible fraction of faults persist for a long time in the program before causing a crash, and that these faults can cause significant reliability problems such as extended downtimes [9], [27], [30]. We call these long-latency crashes (LLCs). Therefore, there is a compelling need to develop techniques for protecting programs from LLC causing faults.

Prior work has experimentally assessed LLCs through fault injection experiments [9]. However, they do not provide much insight into why some faults cause LLCs. This is important because (1) fault injection experiments require a lot of computation time, especially to identify relatively rare events such as LLCs, and (2) fault injection cannot guarantee completeness in identifying all or even most LLC causing locations. The latter is important in order to ensure that crash latency is bounded in the program by protecting LLC causing program locations. Yim et al. [30] analyze error propagation latency in the program, and develop a coarse-grained categorization of program locations based on whether a fault in the location can cause LLCs. The categorization is based on where the program data resides, such as text segment, stack segment or heap segment. While this is useful, it does not help programmers decide which parts of the program need to be protected, as protecting all parts of the program that manipulate the heap data or stack data can lead to prohibitive performance overheads.

In contrast to the above work, we present a technique to perform fine grained classification of program's data at the level of individual variables and program statements, based on whether a fault in the data item causes an LLC. The main insight underlying our work is that very few program locations are responsible for LLCs, and that these locations conform to a few dominant code patterns. Our technique

performs static analysis of the program to identify the LLC causing code patterns. However, not every instance of the LLC-causing code pattern leads to an LLC. Our technique further uses dynamic analysis coupled with a very selective fault injection experiment, to filter the false positives and isolate the few instances of the patterns that lead to LLCs. We have implemented our technique in a completely automated tool called CRASHFINDER, which is integrated with the LLVM compiler infrastructure [15]. *To the best of our knowledge, we are the first to propose an automated and efficient method to systematically identify LLC causing program locations for protection in a fine-grained fashion.*

We make the following contributions in this paper.

- Identify the dominant code patterns that can cause LLCs in programs through a large-scale fault injection experiment we conducted on a total of ten benchmark applications,
- Develop an automated static analysis technique to identify the LLC-causing code patterns in programs, based on the fault injection study,
- Propose a dynamic analysis and selective fault injection-based approach to filter out the false-positives identified by the static analysis technique, and identify LLCs.
- Implement the static and dynamic analysis techniques in an automated tool. We call this CRASHFINDER.
- Evaluate CRASHFINDER on benchmark applications from the SPEC [13], PARBOIL [25], PARSEC [2] and SPLASH-2 [29] benchmark suites. We find that CRASHFINDER can accurately identify over 90% of the LLC causing locations in the program, with no false-positives, and is about nine orders of magnitude faster than performing exhaustive fault injections to identify all LLCs in a program.

II. FAULT MODEL AND BACKGROUND

In this section, we first present our fault model, and then define the terms we will use. We then explain why bounding crash latency is important, and some specifics of the experimental infrastructure that we use for the analysis.

A. Fault Model

In this paper, we consider transient hardware faults that occur in the computational elements of the processor, including pipeline stages, flip-flops, arithmetic and logic units (ALUs). We do not consider faults in the memory or cache, as we assume that these are protected with ECC. Likewise, we do not consider faults in the processor's control logic as we assume it is protected. Finally, we do not consider faults in the instructions' encoding as these can be detected through other means such as error correction codes. Our fault model is in line with other work in the area [11], [17], [8], [26].

B. Terms

We use the following terms defined in prior work [30], [9].

- **Fault occurrence**: The event corresponding to the occurrence of the hardware fault. The fault may or may not result in an error.
- **Fault activation**: The event corresponding to the manifestation of the fault to the software, i.e., the fault becomes an error and corrupts some portion of the software state (e.g., register, memory location). The error may or may not result in a crash.
- **Crash:** The raising of a hardware trap or exception due to the error, because the program attempted to perform an action it should not have (e.g., read outside its memory segments).
- **Crash latency**: The number of dynamic instructions executed by the program from fault activation to crash. This definition is slightly different from prior work which has used CPU cycles to measure the crash latency. The main reason we use dynamic instructions rather than CPU cycles is that we wish to obtain a platform independent characterization of long latency crashes.
- **Long latency crashes (LLCs)**: Crashes that have crash latency of greater than 1,000 dynamic instructions. Prior work has used a wide range of values for long latency crashes, ranging from 10,000 CPU cycles [20] to as many as 10 million CPU cycles [30]. We use 1,000 instructions as our threshold as (1) each instruction corresponds to multiple CPU cycles in our system, and (2) we found that in our benchmarks, the length of the static data dependency sequences are far smaller, and hence setting 1,000 instructions as the threshold already filters out 99% of the crash-causing faults (Section IV), showing that 1000 instructions is a reasonable threshold.

C. Why bound the crash latency?

We now explain our rationale for studying LLCs and why it is important to bound the crash latency in programs. We note that similar observations have been made in prior work [9], [30], and that studies have shown that having unbounded crash latency can result in severe failures. We consider one example.

Assume that the program is being checkpointed every 8,000 instructions so that it can be recovered in the case of a failure (we set aside the practicality of performing such fine grained checkpointing for now). We assume that the checkpoints are gathered in an application independent manner, i.e., the entire state of the program is captured in the checkpoint. If the program encounters an LLC of more than 10,000 instructions, it is highly likely that one or more checkpoints will be corrupted (by the fault causing the LLC). This situation is shown in Figure 1. However, if the crash latency is bounded to 1,000 instructions (say), then it is highly unlikely for the fault to corrupt more than one checkpoint. Note that the latency between the fault activation and the fault occurrence does not matter in this case, as the checkpoint is corrupted only when the fault actually gets activated. Therefore, we focus on the crash latency in this paper, i.e., the number of dynamic instructions from the fault activation to the crash.

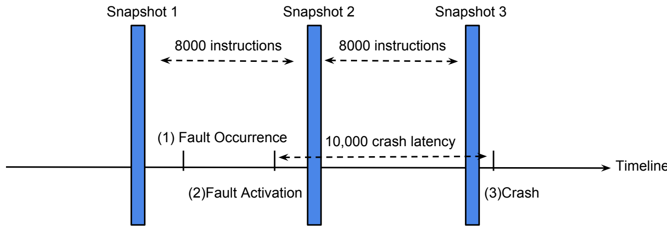

Fig. 1: Long Latency Crash and Checkpointing

Identifying program locations that are prone to LLC is critical to improve system reliability so that one can bound crash latency by selectively protecting LLC-prone locations with minimal performance overheads. For example, one can duplicate the backward slices of the LLC-prone locations, or use low-cost detectors for these locations like what prior work has done [24]. In this paper, we focus on identifying such LLC-causing program locations, and defer the problem of protecting the locations to future work.

D. LLVM Compiler

In this paper, we use the LLVM compiler [15] for performing the static analysis of which program locations lead to LLCs. Our choice of LLVM is motivated by three reasons. First, LLVM uses a typed intermediate representation (IR), in which source-level constructs can be easily represented. In particular, it preserves the names of variables and functions, which makes source mapping feasible. This allows us to perform a fine-grained analysis of which program locations cause LLCs and map it to the source code. Secondly, LLVM IR is a platform neutral representation and abstracts out many low level details of the hardware and assembly language. This greatly aids in portability of our analysis to different architectures, and simplifies the handling of the special cases of different assembly language formats. Finally, LLVM has been shown to be a good match for doing fault injection studies [28], and there are fault injectors developed for LLVM. Therefore, in this paper, when we say instruction, we mean an instruction at the LLVM IR level.

III. RELATED WORK

There have been many papers that tolerate hardware faults through software techniques. We consider these below.

EDDI [19] and SWIFT [21] are compiler-based techniques that use full duplication to protect program data. Full duplication can achieve high coverage but incurs significant runtime overhead. Feng et al. [8] have attempted to reduce runtime overhead by only protecting critical instructions in the program that are unlikely to be detected by other means. However, these techniques do not focus specifically on LLCs, and consequently overprotect the application, resulting in high overheads.

Another direction of research is on reducing the time taken to perform fault injections by pre-analyzing the application's code and only injecting faults into a few representative locations to evaluate the program's error resilience. Examples of this class of techniques are Relyzer [11] and GangES [12], both of which focus on SDC-causing errors. Unfortunately, it is difficult to extend these approaches to LLCs, as the analysis used by these approaches is specific to SDC-causing errors.

A third direction of research in this area has attempted to identify program characteristics that correlate with failure propensity through static and dynamic analysis [17], [26]. These techniques develop heuristics based on dominant code patterns that cause SDCs, and selectively protect the SDC causing code regions using the heuristics. While broadly similar to our work, the heuristics developed by these techniques are specific to SDCs and do not apply to LLC causing errors.

To the best of our knowledge, there have been only two studies of LLCs in programs. The first one by Gu et. al. [9] injected faults into the Linux kernel and found that less than 1% of the errors resulted in LLCs. They further found that many of the severe failures that result in extended downtimes in the system were caused by these LLCs, due to error propagation. The authors give examples of faults that resulted in the LLCs, but they do not attempt to categorize the code patterns that were responsible for the LLCs. The second study is by Yim et al. [30], who studied the correlation between LLC-causing errors and the fault location in the program. However, they perform a coarse-grained categorization of the fault locations based on where the data resides (e.g., stack, heap etc.). Such a coarse-grained categorization is unfortunately not very useful when one wants to protect specific variables or program locations, as protecting the entire stack/heap segment is too expensive. Although they provide some insights on the characteristics of possible LLC-causing errors, they do not develop an automated way to predict which faults would lead to an LLC and which would not. It is also worth noting that neither of the above papers achieves exhaustive characterization of the LLC-causing faults.

Rashid et. al. [20] have built an analytical trace-based model to predict the propagation of intermittent hardware errors in a program. The model can be used to predict the latency of crash causing faults in the program, and hence find the LLC locations. They find that less than 0.5% of faults cause LLCs using the model. While useful, their model requires building the program's Dynamic Dependence Graph (DDG), which can be prohibitively expensive for large programs as it is directly proportional to the number of instructions executed by it. Further, they make many simplifying assumptions in their model which may not hold in the real world. Similarly, Lanzaro et. al. [14] built an automated tool which is able to analyze arbitrary memory corruptions based on execution trace when faults present in system. While their technique is useful in terms of analyzing fault propagation, it incurs prohibitive overheads as it requires the entire trace to be captured at runtime. Further, they focus on software faults as opposed to hardware faults. Finally, they do not make any attempt to identify LLC-causing faults.

Chandra et. al. [5] study program errors that violate the fail-stop model and result in corrupting the data written to permanent storage, or communicated to other processes. They find that between 2% to 7% of faults cause such violations, and propose using a transaction-based mechanism to prevent the propagation of these faults. While transaction-based techniques are useful, they require significant software engineering effort,

as the application needs to be rewritten to use transactions. This is very difficult for most commodity systems.

In contrast to the above techniques, our technique identifies specific program locations that result in LLCs, and can hence support fine-grained protection. Further, it uses predominantly static analysis coupled with dynamic analysis and a selective fault injection experiment, making it highly scalable and efficient compared to the above approaches. Finally, our technique does not require any programmer intervention or application rewriting and is hence practical to deploy on existing software.

IV. Initial Fault Injection Study

In this section, we perform an initial fault injection study for characterizing the LLC causing locations in a program. The goal of this study is to measure the frequency of LLCs, and understand the reasons for them in terms of the program's code. In turn, this will allow us to formulate heuristics for identifying the LLC-causing code patterns in Section V. We first explain our experimental setup for this study, and then discuss the results.

A. Fault Injection Experiment

To perform the fault injection study, we use LLFI [28], an open-source fault injector that operates at the LLVM compiler's IR level. We inject faults into the destination registers of LLVM IR instructions, as per our fault model in Section II. We first profile each program to get the total number of dynamic instructions. We then inject a single bit flip in the result (i.e., register or memory location) of a single dynamic instruction chosen at random from the set of all dynamic instructions executed by the program. Our benchmarks are chosen from the SPEC [13], PARBOIL [25], PARSEC [2] and SPLASH-2 suites [29]. We choose ten programs at random from these suites, and inject a total of $1,000$ faults in each, for a total of $10,000$ fault injection experiments. The details of the benchmarks are explained in Section VII-A.

Note that our way of injecting faults using LLFI ensures that the fault is activated right away as it directly corrupts the program's state during the injection. Therefore, we do not measure activation as the set of activated faults is the same as the set of injected faults. We categorize the results into Crashes, SDCs, Hangs and Benign faults in our experiment. Because our focus in this paper is on LLCs, we record the crash latency for crash-causing faults in terms of the number of dynamic LLVM IR instructions between the fault injection and the crash. However, when the program crashes, its state will be lost, and hence we periodically write to permanent storage the number of dynamic instructions executed by the program after the fault injection. The counting of the dynamic instructions is done using the tracing feature of LLFI, which we have enabled in our experiments.

B. Fault Injection Results

We classify the results of the fault injection experiments into SDC, crash and benign. Hangs were negligible in our experiment and are not reported. Figure 2 shows the aggregated fault injection results across the benchmarks. We find that on average, crashes constitute about 35% of the faults, SDC constitute 4.2%, and the remaining are benign faults (about 60%). We focus on crashes in the rest of this paper, as our focus is on LLCs.

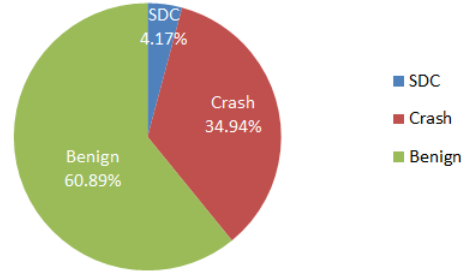

Fig. 2: Aggregate Fault Injection Results across Benchmarks

Figure 3 shows the distribution of crash latencies for all the faults that led to crashes in the injections. On average, the percentage of LLCs is about 0.38% across the ten benchmarks. Recall that we set 1,000 dynamic instructions as the threshold for determining whether a crash is an LLC. Therefore, LLCs constitute a relatively small fraction of the total crashes in programs. This is why it is important to devise fine-grained techniques to identify them, as even a relatively large fault injection experiment such as ours exposes very few LLCs in the program (38 in absolute numbers). The percentages of LLCs among all the crash causing faults, vary from 0% to 3.6% across programs due to benchmark specific characteristics. The reasons for these variations are discussed in Section VIII.

We also categorized the LLCs based on the code patterns in which the LLC locations occurred. In other words, we study the kinds of program constructs which when fault injected, are likely to cause LLCs. We choose the largest five applications from the ten benchmarks for studying the code characteristics since the larger the programs, the more code patterns they may reveal. Thus we choose sjeng, hmmer, href, libquantum and mcf for our detailed investigation.

Figure 4 shows the distribution of the LLC-causing code patterns we found in our experiments. The patterns themselves are explained in Section IV-C. *We find that about 95% of the LLC causing code falls into three dominant patterns, namely (1) Pointer Corruption (20%), (2) Loop Corruption (56%), and (3) State Corruption (19%).* Therefore we focus on these three patterns in the rest of this paper.

C. Code Patterns that Lead to LLCs

As mentioned in the previous section, we find that LLCs fall into three dominant patterns namely, *pointer related LLC*, *loop related LLC* and *state related LLC*. We explain each category with code examples in the following subsections. Although these observations were made at the LLVM IR level, we use C code for clarity to explain them.

Pointer Corruption LLC occurs when a fault is injected into pointers that are written to memory. An erroneous pointer value is stored in the memory, and this value can be used as a memory operation later on to cause crash. Because the pointer may not be read for a long time, this pattern has the potential to cause an LLC. Figure 5A shows the case we observed in sjeng

Fig. 3: Latency Distribution of Crash-Causing Errors in Programs: The purple bars represent the LLCs as they have a crash latency of more than 1000 instructions. The number shown at the top of each bar shows the percentage of crashes that resulted in LLCs. The error bars for LLCs range from 0%(cutcp) to 1.85%(sjeng).

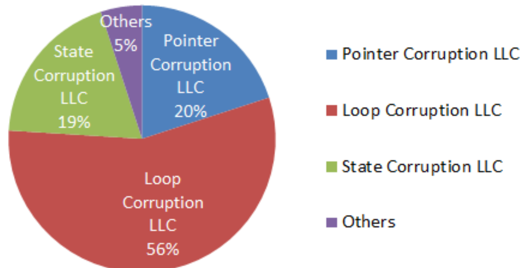

Fig. 4: Distribution of LLC Categories across 5 Benchmarks (sjeng, libquantum, hmmer, h264ref and mcf). Three dominant categories account for 95% of the LLCs.

from our fault injection experiment. In the function *reloadMT*, **p0* and *next* are assigned to a global static variable, *state*, at line 7 and line 8 respectively. The fault is injected on the pointer, **p0*, at line 10. As a result, an erroneous pointer value is saved in the memory and it is used as a memory operation in the function *randomMT* at line 18 after a long time. This leads to an LLC.

Loop Corruption LLC When faults are injected into loop conditions or array indices inside the loop, the array manipulated by the loop (if any) may aggressively corrupt the stack, and cause LLC. We categorize this as *Loop Corruption LLC*. There are two cases in which this LLC can occur.

The first case is when a fault occurs in the array index of an array written within the loop. This fault can corrupt a large area of stack since an erroneous array index is used for array address offset calculations in every iteration of the loop. This large-scale corruption to the stack significantly increases the chance of corrupting address values (i.e., pointers, return address etc) on the stack, which in turn can result in a crash much later. For example, in Figure 5B, when a fault is injected into *next* making a corrupted value saved back to it at the line 5, the struct array *perm[]* at line 9 corrupts values on the stack. When the corrupted value is used for memory operations later in the program, an LLC is observed.

The second case occurs when faults are injected into termination conditions of the loop, causing a stack overflow to occur. This is shown in Figure 5C. Assume that a fault is injected into *piece_count* at line 3, and makes it a large value. This will cause the *for* loop at line 5 to execute for a much larger number of iterations, thereby corrupting the stack and eventually leading to a LLC.

State Corruption LLC occurs when faults are injected into state variables or lock (synchronization) variables in state machine structures. These variables are declared as static or global variables and are used to allocate or deallocate particular pieces of memory. If these states are corrupted, crashes may happen between states, thus causing LLC. For the code shown in Figure 5D, when we inject a fault in *opstatus* at line 7, the variable *opstatus* becomes a nonzero value (from zero) when the state goes to *quantum_objcode_stop*. Later in the function *quantum_objcode_put* when the state is updated to *quantum_objcode_stop*, the *opstatus* variable is examined to decide whether a particular memory area should be accessed (line 23). Due to the fault injected, we observed that *objcode* is

```
1  static unsigned int state[N+1];
2  static unsigned int *next;
3  ...
4  unsigned int reloadMT(void)
5  {
6    ...
7    register unsigned int *p0 = state;
8    next = state+1;
9    ...
10   *p0++ = *pM++ ^ ... ;
11   ...
12 }
13 ...
14 unsigned int randomMT(void)
15 {
16   unsigned int y;
17   ...
18   y  = *next++;
19   ...
20 }
21 ...
```

(A) Pointer corruption (sjeng)

```
1  ...
2  arc_t *primal_bea_mpp(){
3    ...
4    for(...){
5      next++;
6      perm[next]->a = arc;
7    }
8    ...
9  }
10 ...
```

(B) Loop corruption on array index (mcf)

```
1  ...
2  void reset_piece_square (void) {
3    piece_count = 32;
4    ...
5    for (i = 1; i <= piece_count; i++){
6      ...
7      promoted_board[pieces[i]] = 1;
8      ...
9    }
10 }
11 ...
```

(C) Loop corruption on termination condition (sjeng)

```
1  int opstatus = 0;
2  unsigned char *objcode = 0;
3  ...
4  void quantum_objcode_stop()
5  {
6    ...
7    opstatus = 0;
8    free(objcode);
9    ...
10 }
11 ...
12 void quantum_objcode_start()
13 {
14   ...
15   opstatus = 1;
16   objcode = malloc(OBJCODE_PAGE * sizeof(char));
17   ...
18 }
19 ...
20 int quantum_objcode_put(unsigned char operation,
       ...)
21 {
22   ...
23   if(!opstatus)
24     return 0;
25   ...
26   for(i=0; i<size; i++)
27   {
28     objcode[position] = buf[i];
29     position++;
30   }
31 }
32 ...
```

(D) State corruption (libquantum)

Fig. 5: Code examples showing the three kinds of LLCs that occurred in our experiments.

accessed at line 28 while in the state *quantum_objcode_stop*. This leads to a LLC as it accesses the unallocated memory area *objcode*, which is illegal.

V. APPROACH

In this section, we describe our proposed technique CRASHFINDER, to find *all the LLCs* in a program. CRASHFINDER consists of three phases, as Figure 6 shows

In the first phase, it performs a static analysis of the program's source code to determine the potential locations that can cause LLCs. The analysis is done based on the code patterns in Section IV-C. We refer to this phase of CRASHFINDER as CRASHFINDER STATIC. In the second phase, it performs dynamic analysis of the program (under a given set of inputs) to determine which dynamic instances of the static locations are likely to result in LLCs. We call this phase CRASHFINDER DYNAMIC. In the last phase, it injects a selected few faults to the dynamic instances chosen by CRASHFINDER DYNAMIC. We refer to this phase of CRASHFINDER as *selective fault injection*. We describe the three phases in the three subsections.

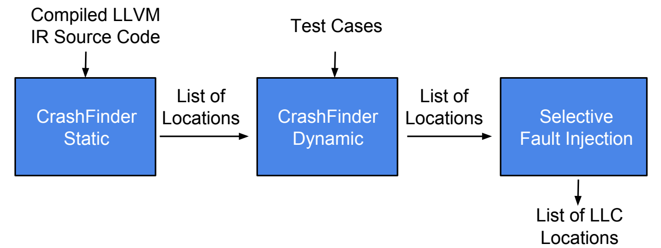

Fig. 6: Workflow of CRASHFINDER

A. Phase 1: Static Analysis (CRASHFINDER STATIC)

CRASHFINDER STATIC is the static analysis portion of our technique that statically searches the program's code for the three patterns corresponding to those identified in Section IV-C. We found that these three patterns are responsible for almost all the LLCs in the program, and hence it suffices to look for these patterns to cover the LLCs. However, not every instance of these patterns may lead to an LLC, and hence we

may get false-positives in this phase. False-positives are those locations that conform to the LLC causing patterns but do not lead to an LLC, and will be addressed in the next phase.

The algorithm of CRASHFINDER STATIC takes the program's source code compiled to the LLVM IR as an input and outputs the list of potential LLC causing locations. Specifically, CRASHFINDER STATIC looks for the following patterns in the program:

*1) **Pointer Corruption LLC**:* CRASHFINDER STATIC finds pointers that are written to memory in the program. More specifically, it examines static data dependency sequences of all pointers, and only consider the ones that end with *store* instruction.

*2) **Loop Corruption LLC**:* In this category, CRASHFINDER STATIC finds loop termination variables in loop headers and array index assignment operations. For loop termination variable(s), it looks for the variable(s) that is used for comparison with the loop index variable in loop headers. For array index assignment, CRASHFINDER STATIC first locates binary operations with a variable and a constant as operands, then checks if the result being stored is used as offset in array address calculation. If yes, then we can infer that the variable being updated will be used as the address offset of an array. In LLVM, offset calculations are done through a special instruction and are hence easy to identify statically.

*3) **State Corruption LLC**:* CRASHFINDER STATIC finds static and global variables used to store state or locks. Because these may depend on the application's semantics, we devise a heuristic to find such variables. If a static variable is loaded and directly used in comparison and branches, we assume that it is likely to be a state variable or a lock variable. We find that this heuristic allow us to cover most of these cases without semantic knowledge of the application.

*B. Phase 2: Dynamic Analysis (*CRASHFINDER DYNAMIC*)*

In this phase, our technique attempts to eliminate the false positives from the static locations identified in phase 1. One straw man approach for doing so is to inject faults in every dynamic instance of the static locations to determine if it leads to an LLC. However, a single static instruction may correspond to hundreds of thousands of dynamic instances in a typical program, especially if it is within a loop. Further, each of these dynamic instances needs to be fault injected multiple times to determine if it will lead to an LLC, and hence a large number of fault injections will need to be performed. All this adds up to considerable performance overheads, and hence the above straw man approach does not scale.

We propose an alternate approach to cut down the number of fault injection locations to filter out the false positives. Our approach uses dynamic analysis to identify a few dynamic instances to consider for injection among the set of all the identified static locations. The main insight we leverage is that there are repeated control-flow sequences in which the dynamic instances occur, and it is sufficient to sample dynamic instances in each *unique* control-flow sequence to obtain a representative set of dynamic instances for fault injection. This is because the crash latency predominantly depends on the control-flow sequence executed by the program after the

```
1  void multig(long my_id){
2    ...
3    while ((!flag1) && (!flag2)) {
4      ...
5      relax();
6      copy_red();
7      relax();
8      copy_black();
9      ...
10     }
11 }
12
13 void relax(){
14   ...
15   for (...) {
16     ...
17     t1a = (double *) t2a[i];
18     ...
19   }
20 }
```

(a)

```
sample candidate 1      → relax()
                          copy_red()
sample candidate 2      → relax()
                          copy_black()
sample candidate 3      → relax()
                          copy_red()
sample candidate 4      → relax()
                          copy_black()
                          ...
sample candidate N      → relax()
                          copy_red()
sample candidate (N+1)  → relax()
                          copy_black()
```

(b)

Fig. 7: Dynamic sampling heuristic. (a) Example source code (ocean program), (b) Execution trace and sample candidates.

injection at a given program location. Therefore, it suffices to obtain one sample from each unique control flow pattern in which the dynamic instance occurs. We determine the control flow sequences at the level of function calls. That is we sample the dynamic instances with different function call sequences, and ignore the ones that have the same function call sequences. We show in Section VIII that this sampling heuristic works well in practice.

We consider an example to illustrate the sampling heuristic to determine which dynamic instances to choose. Figure 7(b) shows the dynamic execution trace generated by the code in Figure 7(a). For example, we want to sample the dynamic instances corresponding to the variable *t1a* at line 17 in Figure 7(a). Firstly, because *t1a* is within a loop in function *relax*, it corresponds to multiple dynamic instances in the trace. We only consider one of them as candidate for choosing samples (we call it a sample candidate), since they have same function call sequences (no function calls) in between. Secondly, function *relax* is called within a loop in function *multig* at lines 5 and 7. As can be seen in the Figure 7, there are two recurring function call sequences circumscribing the execution of the static location corresponding to the sample candidates, namely *relax() copy_red()* and *relax() copy_black()*. We collect one sample of each sequence regardless of how many times they occur. In this case, only sample candidate 1 and 2 are selected for later fault injections. We find that this dramatically reduces the fault injection space thereby saving considerable time.

C. Phase 3: Selective Fault Injections

The goal of this phase is to filter out all the false-positives identified in the previous phase through fault injections. Once we have isolated a set of dynamic instances from CRASHFINDER DYNAMIC to inject for the static location, we configure our fault injector to inject two faults into each dynamic instance, one fault at a time. We choose one high-order bit and and one low-order bit at random to inject into, as we found experimentally that LLCs predominantly occur either in the high-order bits or the low-order bits, and hence one needs to sample both.

We then classify the location as an LLC location (i.e., not a false positive) if any one of the injected faults results in an LLC. Otherwise, we consider it a false-positive, and remove it from the list of LLC locations. Note that this approach is conservative as performing more injections can potentially increase the likelihood of finding an LLC, and hence it is possible that we miss some LLCs. However, as we show in Section VIII, our approach finds most LLCs even with only two fault injections per each dynamic instance. We also show that increasing the number of fault injections beyond 2 for each dynamic instance does not yield substantial benefits, and hence we stick to 2 injections per instance.

VI. IMPLEMENTATION

We implemented CRASHFINDER STATIC as a pass in the LLVM compiler [15] to analyze the IR code and extract the code patterns. We implemented the CRASHFINDER DYNAMIC also as an LLVM pass that instruments the program to obtain its control-flow. CRASHFINDER DYNAMIC then analyzes the control-flow patterns and determines what instances to choose for selective fault injection. We use the LLFI fault injection framework [28] to perform the fault injections. Finally, we used our crash latency measurement library to determine the crash latencies after injection.

To use CRASHFINDER [1], all the user needs to do is to compile the application code with the LLVM compiler using our module. No annotations are needed. The user also needs to provide us with representative inputs so that CRASHFINDER can execute the application, collect the control-flow patterns and choose the dynamic instances to inject faults.

VII. EXPERIMENTAL SETUP

We empirically evaluate CRASHFINDER in terms of accuracy and performance. We use a fault injection experiment to measure the accuracy, and use execution time of the technique to measure its performance. We evaluate both CRASHFINDER and CRASHFINDER STATIC separately to understand the effect of different parts of the technique (CRASHFINDER includes CRASHFINDER STATIC, CRASHFINDER DYNAMIC and the selective fault injection). We compare both the accuracy and the performance of both techniques to those of exhaustive fault injections that are needed to find all the LLCs in a program [2]. Our experiments are all carried out on an Intel Xeon E5 machine, with 32 GB RAM running Ubuntu Linux 12.04.

[1]CRASHFINDER and its source code can be freely downloaded from https://github.com/DependableSystemsLab/Crashfinder

[2]Our goal is to find *all* LLC causing locations in the program so that we can selectively protect them and bound the crash latency.

We first present the benchmarks used (Section VII-A, followed by the research questions (Section VII-B). We then present an overview of the methodology we used to answer each of the research questions (Section VII-C).

A. Benchmarks

We choose a total of ten benchmarks from various domains for evaluating CRASHFINDER. The benchmark applications are from SPEC [13], PARBOIL [25], PARSEC [2] and SPLASH-2 [29]. All the benchmark application are compiled and linked into native executables using LLVM, with standard optimizations enabled. We show the detailed information of the benchmarks in Table I.

TABLE I: Characteristics of Benchmark Programs

Benchmark	Benchmark Suite	Description
libquantum	SPEC	A library for the simulation of a quantum computer
h264ref	SPEC	A reference implementation of H.264/AVC (Advanced Video Coding)
blackscholes	PARSEC	Option pricing with Black-Scholes Partial Differential Equation (PDE)
hmmer	SPEC	Uses statistical description of a sequence family's consensus to do sensitive database searching
mcf	SPEC	Solves single-depot vehicle scheduling problems planning transportation
ocean	SPLASH-2	Large-scale ocean movements simulation based on eddy and boundary currents
sad	PARBOIL	Sum of absolute differences kernel, used in MPEG video encoders
sjeng	SPEC	A program that plays chess and several chess variants
cutcp	PARBOIL	Computes the short-range component of Coulombic potential at each grid point
stencil	PARBOIL	An iterative Jacobi stencil operation on a regular 3-D grid

B. Research Questions

We answer the following research questions(RQs) in our experiments.

RQ1: *How much speedup do* CRASHFINDER STATIC *and* CRASHFINDER *achieve over exhaustive injection ?*

RQ2: *What is the precision of* CRASHFINDER STATIC *and* CRASHFINDER*?*

RQ3: *What is the recall of* CRASHFINDER STATIC *and* CRASHFINDER*?*

RQ4: *How well do the sampling heuristics used in* CRASHFINDER *work in practice ?*

C. Experimental Methodology

We describe our methodology for answering each of the RQs below. We perform fault injections using the LLFI fault injector [28] as described earlier.

1) Performance: In order to answer RQ1, we measure the total time taken for executing CRASHFINDER STATIC, CRASHFINDER and the exhaustive fault injections. More specifically, for each benchmark, we measure the total time used for (1) CRASHFINDER STATIC, (2) CRASHFINDER, which includes CRASHFINDER STATIC, CRASHFINDER DYNAMIC and selective fault injections to identify LLCs and, (3) exhaustive fault injections to find LLCs.

2) Precision: The precision is an indication of the false-positives produced by CRASHFINDER STATIC and CRASHFINDER. To measure the precision, we inject 200 faults randomly at each static program location identified by CRASHFINDER STATIC or CRASHFINDER, and measure the latency. If none of the injections at the location result in an LLC, we declare it to be a false positive. Note that we choose 200 fault injections per location to balance time and comprehensiveness. If we increase the number of faults, we may find more LLC causing locations, thus decreasing the false positives. Thus, this method gives us a conservative upper bound on the false-positives of the technique.

3) Recall: The recall is an indication of the false-negatives produced by CRASHFINDER STATIC and CRASHFINDER. To measure the recall of CRASHFINDER STATIC and CRASHFINDER, we randomly inject 3,000 faults for each benchmark and calculate the fraction of the observed LLCs that were covered by CRASHFINDER STATIC and CRASHFINDER respectively. Thus 30,000 faults in total are injected over ten benchmark applications for this experiment. Note that this is in addition to the 1,000 fault injection experiments performed in the initial study, which were used to develop the two techniques. We do not include the initial injections in the recall measurement to avoid biasing the results.

4) Heuristics for Sampling: As mentioned in Section V, there are two heuristics used by CRASHFINDER to reduce the space of fault injections it has to perform. The first is to limit the chosen instances to unique dynamic instances of control-flow patterns in which the static instructions appear, and the second is to limit the number of faults injected in the dynamic instances to two faults per instance. These heuristics may lead to loss in coverage. We investigate the efficacy of these heuristics by varying the parameters used in them and measure the resulting recall.

VIII. RESULTS

This section presents the results of our experiments for evaluating CRASHFINDER STATIC and CRASHFINDER. Each subsection corresponds to a research question (RQ).

A. Performance (RQ1)

We first present the results of running CRASHFINDER and CRASHFINDER STATIC in terms of the number of instructions in each benchmark, and then examine how much speedup can CRASHFINDER achieve over exhaustive fault injections.

Table II shows the numbers of instructions for each benchmark. In the table, columns *Total S.I* and *Total D.I* show the total numbers of static instructions and dynamic instructions of each benchmark. Columns CRASHFINDER STATIC *S.I* and CRASHFINDER STATIC *D.I* indicate the numbers of static instructions and dynamic instructions corresponding to the static instructions that were found by CRASHFINDER STATIC as LLC causing locations. Columns CRASHFINDER *S.I* and CRASHFINDER *D.I* show the numbers of static instructions and dynamic instances of the static locations that CRASHFINDER identified as LLC causing locations. As can be seen from the table, on average, CRASHFINDER STATIC identified 2.99% of static instructions as LLC causing, which corresponds to about 5.87% of dynamic instructions. In comparison, CRASHFINDER further winnowed the number of static LLC-causing locations to 0.89%, and the number of dynamic instructions to just 0.385%, thereby achieving a significant reduction in the dynamic instructions. The implications of this reduction are further investigated in Section IX.

Figure 8 shows the orders of magnitude of time reduction achieved by using CRASHFINDER STATIC and CRASHFINDER to find LLCs, compared to exhaustive fault injections, for each benchmark. In the figure, CRASHFINDER STATIC refers to the time taken to run CRASHFINDER STATIC. CRASHFINDER refers to the time taken to run all three components of CRASHFINDER, namely CRASHFINDER STATIC, CRASHFINDER DYNAMIC and the selective fault injection phase. Note that the exhaustive fault injection times are an estimate based on the number of dynamic instructions that need to be injected, and the time taken to perform a single injection. We emphasize that the numbers shown represent the orders of magnitude in terms of speedup. For example, a value of 12 in the graph, means that the corresponding technique was 10^{12} times faster than performing exhaustive fault injections. In summary, on average CRASHFINDER STATIC achieves a total of 13.47 orders of magnitude of time reduction whereas CRASHFINDER achieves 9.29 orders of magnitude of time reduction over exhaustive fault injecton to find LLCs.

We also measured the wall clock time of the different phases of CRASHFINDER. The geometric means of time taken for CRASHFINDER STATIC are 23 seconds, for CRASHFINDER DYNAMIC the time taken is 3.1 hours, while it takes about 3.9 days for the selective fault injection phase. Overall, it takes about 4 days on average for CRASHFINDER to complete the entire process. While this may seem large, note that both the CRASHFINDER DYNAMIC and selective fault injection phases can be parallelized to reduce the time. We did not however do this in our experiments.

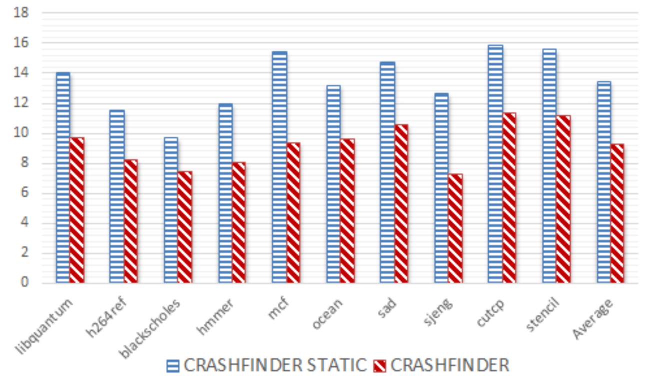

Fig. 8: Orders of Magnitude of Time Reduction by CRASHFINDER STATIC and CRASHFINDER compared to exhaustive fault injections

B. Precision (RQ2)

Figure 9 shows the precision of CRASHFINDER STATIC and CRASHFINDER for each benchmark. The average precision of CRASHFINDER STATIC and CRASHFINDER are 25.42% and 100% respectively. The reason CRASHFINDER

TABLE II: Comparison of Instructions Given by CRASHFINDER and CRASHFINDER STATIC

Benchmarks \ Metrics	Total S.I.	Total D.I. (in million)	CRASHFINDER STATIC S.I (%)	CRASHFINDER STATIC D.I (%)	CRASHFINDER S.I (%)	CRASHFINDER D.I (%)
libquantum	15319	870	1.85%	9.27%	0.18%	0.011%
h264ref	189157	116	0.85%	3.92%	0.14%	0.150%
blackscholes	758	0.13	3.29%	1.81%	0.66%	0.004%
hmmer	92287	4774	0.51%	3.53%	0.13%	0.437%
mcf	4086	6737	6.29%	8.75%	2.62%	1.383%
ocean	21300	1061	3.46%	3.11%	0.53%	0.003%
sad	3176	1982	4.47%	5.56%	0.69%	0.473%
sjeng	33931	137	1.70%	15.55%	0.16%	0.567%
cutcp	3868	11389	3.13%	6.35%	0.39%	0.001%
stencil	2193	7168	4.38%	0.84%	0.41%	0.819%
Average	**36608**	**3423**	**2.99%**	**5.87%**	**0.89%**	**0.385%**

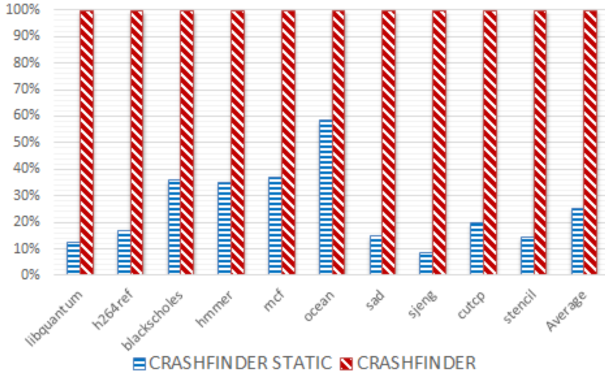

Fig. 9: Precision of CRASHFINDER STATIC and CRASHFINDER for finding LLCs in the program

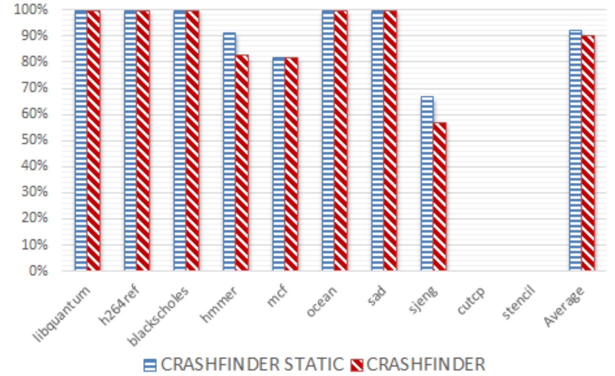

Fig. 10: Recall of CRASHFINDER STATIC and CRASHFINDER

has a precision of 100% is that all the false-positives produced by the static analysis phase (CRASHFINDER STATIC) are filtered out by the latter two phases, namely CRASHFINDER DYNAMIC, and selective fault injections. The main reason why CRASHFINDER STATIC has low precision is because it cannot statically determine the exact runtime behavior of variables. For example, a pointer can be saved and loaded to/from memory in very short intervals, and would not result in an LLC. This behavior is determined by its runtime control flow, and cannot be determined at compile time, thus resulting in false positives by CRASHFINDER STATIC. However, CRASHFINDER does not have this problem as it uses dynamic analysis and selective fault injection.

C. Recall (RQ3)

Figure 10 shows the recall of CRASHFINDER STATIC and CRASHFINDER. The average recall of CRASHFINDER STATIC and CRASHFINDER are 92.47% and 90.14% respectively. Based on the results, we can conclude that (1) CRASHFINDER STATIC is able to find most of the LLC causing locations showing that the code patterns we identified are comprehensive and, (2) our heuristics used in CRASHFINDER DYNAMIC and selective fault injections do not filter out many legitimate LLC locations since there is only a 2.33% difference between the recalls of CRASHFINDER STATIC and CRASHFINDER (however, they filter out most of the false positives as evidenced by the high precision of CRASHFINDER compared to CRASHFINDER STATIC). We will discuss this further in the next subsection.

There are two reasons why CRASHFINDER STATIC does not achieve 100% recall: (1)There are a few cases as mentioned in Section IV that do not fall into the three dominant patterns. (2) While CRASHFINDER STATIC is able to find most of the common cases of LLCs, it does not find some cases where the dependency chain spans multiple function calls. For example, the return value of an array index calculation can be propagated through complex function calls and finally used in the address offset operations in a loop. This makes the pointer analysis in LLVM return too many candidates for the pointer target, and so we truncate the dependence chain. However, there is no fundamental reason why we cannot handle these cases. Even without handling the cases, CRASHFINDER finds 92.47% of the cases leading to LLCs in the program.

Note that we did not observe any LLCs in the two benchmark programs *stencil* and *cutcp*. This may be because they have fewer LLC causing locations, and/or they have a small range of bits which may result in LLCs. This was also the case in the initial study IV.

D. Efficacy of Heuristics (RQ4)

As mentioned earlier, there are two heuristics used by CRASHFINDER DYNAMIC to speed up the injections. First, in the dynamic analysis phase (CRASHFINDER DYNAMIC), only a few instruction instances are chosen for injection. Second, in the selective fault injection phase, only a few bits in each of the chosen locations are injected. We examine the effectiveness of these heuristics in practice.

In order to understand the LLC-causing errors that are covered by CRASHFINDER STATIC but not CRASHFINDER, we manually inspected these injections. We found that all of

the missed errors are due to the second heuristic used by the selective fault injection phase. None of the missed errors were due to the first heuristic employed by CRASHFINDER DYNAMIC.

The heuristic for choosing bit positions for selective injections picks two random positions in the word to inject faults into, one from the high-level bits and one from the low-level bits. Unfortunately, this may miss other positions that lead to LLCs. We evaluated the effect of increasing the number of sampled bits to 3 and 5, but even this did not considerably increase the number of LLCs found by CRASHFINDER. This is because most of the missed errors can only be reproduced by injecting into very specific bit positions, and finding these positions will require near exhaustive injections on the words found by CRASHFINDER DYNAMIC, which will prohibitively increase the time taken to complete the selective fault injection phase. Therefore, we choose to retain the heuristic as it is, especially because the difference between CRASHFINDER STATIC and CRASHFINDER is only 2.33%.

With the above being said, the heuristic-based approach used here is an approximation. Hence, there may be multiple sources of inaccuracy in these heuristics. We will further quantify the limits of the heuristic based approach in future work.

IX. DISCUSSION

In this section, we discuss some of the implications of CRASHFINDER on selective protection and checkpointing. We also discuss some of the limitations of CRASHFINDER and improvements.

A. Implication for Selective Protection

One of the main results from evaluating CRASHFINDER is that we find that a very small number of instructions are responsible for most of the LLCs in the program. As per Table II, only 0.89% of static instructions are responsible for more than 90% of the LLC causing errors in the program (based on the recall of CRASHFINDER). Further, CRASHFINDER is able to precisely pinpoint these instructions, thereby allowing these to be selectively protected.

An example of a selective protection technique is value range checking in software [10]. A range check is typically inserted after the instruction that produces the data item to be checked. For example, the *assertion(ptr_address<0x001b, true)* inserted after the static instruction producing *ptr_address* will check the value of the variable whenever the instruction is executed. Since the total number of executions of all such LLC causing instructions is only 0.385% (Table II), the overhead of these checks is likely to be extremely low. We will explore this direction in future work.

B. Implication for Checkpointing Techniques

Our study also establishes the feasibility of fine-grained checkpointing techniques for programs, as such checkpointing techniques would incur frequent state corruptions in the presence of LLCs. For example, Chandra et al. [6] found that the frequency of checkpoint corruption when using a fine-grained checkpointing technique ranges between 25 and 40% due to LLCs. They therefore conclude that one should not use such fine-grained checkpointing techniques, and instead use application-specific coarse-grained checkpointing in which the corresponding probability of checkpoint corruption is 1% to 19%. However, by deploying our technique and selectively protecting the LLC causing locations in the program, one could restrict the crash latency, thus minimizing the chances of checkpoint corruption. Based on the 90% recall of CRASHFINDER, we can achieve a 10-fold reduction in the number of LLC causing locations, thus bringing the checkpoint corruption probability of fine-grained checkpointing down. This would make fine-grained checkpointing feasible, thus allowing faster recovery from errors. This is also a direction we plan to explore in the future.

C. Limitations and Improvements

One of the main limitations of CRASHFINDER is that it takes a long time (on average 4 days) to find the LLC causing errors in the program. The bulk of this time is taken by the selective fault injection phase, which has to inject faults into thousands of dynamic instances found by CRASHFINDER DYNAMIC to determine if they are LLCs. While this is still orders of magnitude faster than performing exhaustive fault injections, it is still a relatively high one-time cost to protect the program. One way to speed this up would be to parallelize it, but that comes at the cost of increased computation resources.

An alternate way to speed up the technique is to improve the precision of CRASHFINDER STATIC. As it stands, CRASHFINDER STATIC takes only a few seconds to analyze even large programs and find LLC causing locations in them. The main problem however is that CRASHFINDER STATIC has a very low precision (of 25.4%). However, this may be acceptable in some cases, where we can protect a few more locations and incur higher overheads in doing so. Even with this overprotection, we still only protect less than 6% of the program's dynamic instructions (Table II). However, one can improve the precision further by finding all possible aliases and control flow paths at compile time [22], and filtering out the patterns that are unlikely to cause LLCs.

Another limitation is that the recall of CRASHFINDER is only about 90%. Although this is still a significant recall, one can improve it further by (1) building a more comprehensive static analyzer to cover the uncovered cases that do not belong to the dominant LLC-causing patterns, and (2) improving the heuristic used in the selective fault injection phase, by increasing the number of fault injections in the selective fault injection phase, albeit at the cost of increased performance overheads (as we found in RQ4, this heuristic was responsible for most of the difference in recall between CRASHFINDER and CRASHFINDER STATIC).

Finally, though the benchmark applications are chosen from a variety of domains such as scientific computing, multimedia, statistics and games, there are other domains that are not covered such as database programs, or system software applications. Further, they are all single-node applications. We defer the extension of CRASHFINDER for distributed applications to our future work.

X. Conclusion and Future Work

In this paper, we identify an important but neglected problem in the design of dependable software systems, namely identifying faults that propagate for a long time before causing crashes, or LLCs. Unlike prior work which has only performed a coarse grained analysis of such faults, we perform a fine grained characterization of LLCs. Interestingly, we find that there are only three code patterns in the program that are responsible for almost all LLCs, and that these patterns can be identified efficiently through static analysis. We build a static analysis technique to find these patterns, and augment it with a dynamic analysis and selective fault-injection based technique to filter out the false positives. We implement our technique in a completely automated tool called CrashFinder. We find that CrashFinder is able to achieve 9 orders of magnitude speedup over exhaustive fault injections to identify LLCs, has no false-positives, and successfully identifies over 90% of the LLC causing locations in ten benchmark programs.

For future work, we plan to (1) apply the results of CrashFinder to selectively protect LLC causing locations and measure the overhead, (2) combine CrashFinder with fine-grained checkpointing methods to achieve fast recovery in the case of crashes, and (3) reduce the performance overhead of CrashFinder and improve its accuracy with more sophisticated static analysis.

XI. Acknowledgment

We thank the anonymous reviewers of DSN'15 and Keun Soo Yim for their comments that helped improve the paper. This work was supported in part by a Discovery Grant from the Natural Science and Engineering Research Council (NSERC), Canada, and an equipment grant from the Canada Foundation for Innovation (CFI). We thank the Institute of Computing, Information and Cognitive Systems (ICICS) at the University of British Columbia for travel support.

References

[1] C. Basile, L. Wang, Z. Kalbarczyk, and R. Iyer. Group communication protocols under errors. In *Reliable Distributed Systems. Proceedings. 22nd International Symposium on*, pages 35–44, Oct 2003.

[2] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81. ACM, 2008.

[3] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *Micro, IEEE*, 25(6):10–16, 2005.

[4] S. Borkar. Electronics beyond nano-scale cmos. In *Proceedings of the 43rd annual Design Automation Conference*, pages 807–808. ACM, 2006.

[5] S. Chandra and P. M. Chen. How fail-stop are faulty programs? In *Fault-Tolerant Computing. Digest of Papers. Twenty-Eighth Annual International Symposium on*, pages 240–249. IEEE, 1998.

[6] S. Chandra and P. M. Chen. The impact of recovery mechanisms on the likelihood of saving corrupted state. In *Proceedings of the 13th International Symposium on Software Reliability Engineering, ISSRE.*, pages 91–101. IEEE, 2002.

[7] C. Constantinescu. Intermittent faults and effects on reliability of integrated circuits. In *Reliability and Maintainability Symposium*, pages 370–374. IEEE, 2008.

[8] S. Feng, S. Gupta, A. Ansari, and S. Mahlke. Shoestring: probabilistic soft error reliability on the cheap. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 385–396. ACM, 2010.

[9] W. Gu, Z. Kalbarczyk, R. K. Iyer, and Z. Yang. Characterization of linux kernel behavior under errors. In *International Conference on Dependable Systems and Networks*. IEEE Computer Society, 2003.

[10] S. K. S. Hari, S. V. Adve, and H. Naeimi. Low-cost program-level detectors for reducing silent data corruptions. In *Dependable Systems and Networks (DSN), 42nd Annual IEEE/IFIP International Conference on*, pages 1–12. IEEE, 2012.

[11] 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 *ACM SIGARCH Computer Architecture News*, volume 40, pages 123–134. ACM, 2012.

[12] S. K. S. Hari, R. Venkatagiri, S. V. Adve, and H. Naeimi. Ganges: Gang error simulation for hardware resiliency evaluation.

[13] J. L. Henning. Spec cpu2000: Measuring cpu performance in the new millennium. *Computer*, 33(7):28–35, 2000.

[14] A. Lanzaro, R. Natella, S. Winter, D. Cotroneo, and N. Suri. An empirical study of injected versus actual interface errors. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 397–408. ACM, 2014.

[15] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization. International Symposium on*, pages 75–86. IEEE, 2004.

[16] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn. Flikker: saving dram refresh-power through critical data partitioning. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, pages 213–224. ACM, 2011.

[17] Q. Lu, K. Pattabiraman, M. S. Gupta, and J. A. Rivers. Sdctune: a model for predicting the sdc proneness of an application for configurable protection. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, page 23. ACM, 2014.

[18] S. Narayanan, J. Sartori, R. Kumar, and D. L. Jones. Scalable stochastic processors. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 335–338. European Design and Automation Association, 2010.

[19] N. Oh, P. P. Shirvani, and E. J. McCluskey. Error detection by duplicated instructions in super-scalar processors. *Reliability, IEEE Transactions on*, 51(1):63–75, 2002.

[20] L. Rashid, K. Pattabiraman, and S. Gopalakrishnan. Modeling the propagation of intermittent hardware faults in programs. In *Dependable Computing (PRDC), IEEE 16th Pacific Rim International Symposium on*, pages 19–26. IEEE, 2010.

[21] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. Swift: Software implemented fault tolerance. In *Proceedings of the international symposium on Code generation and optimization*, pages 243–254. IEEE Computer Society, 2005.

[22] V. Robert and X. Leroy. A formally-verified alias analysis. In *Certified Programs and Proofs*, pages 11–26. Springer, 2012.

[23] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. Enerj: Approximate data types for safe and general low-power computation. In *ACM SIGPLAN Notices*, volume 46, pages 164–174. ACM, 2011.

[24] S. K. Sastry Hari, M.-L. Li, P. Ramachandran, B. Choi, and S. V. Adve. mswat: low-cost hardware fault detection and diagnosis for multicore systems. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 122–132. ACM, 2009.

[25] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 2012.

[26] A. Thomas and K. Pattabiraman. Error detector placement for soft computation. In *Dependable Systems and Networks (DSN), 43rd Annual IEEE/IFIP International Conference on*, pages 1–12. IEEE, 2013.

[27] N. J. Wang and S. J. Patel. Restore: Symptom-based soft error detection in microprocessors. *Dependable and Secure Computing, IEEE Transactions on*, 3(3):188–201, 2006.

[28] J. Wei, A. Thomas, G. Li, and K. Pattabiraman. Quantifying the accuracy of high-level fault injection techniques for hardware faults. In *Dependable Systems and Networks (DSN), 44rd Annual IEEE/IFIP International Conference on*, 2014.

[29] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. In *ACM SIGARCH Computer Architecture News*, volume 23, pages 24–36. ACM, 1995.

[30] K. S. Yim, Z. T. Kalbarczyk, and R. K. Iyer. Quantitative analysis of long-latency failures in system software. In *Dependable Computing, PRDC'09. 15th IEEE Pacific Rim International Symposium on*, pages 23–30. IEEE, 2009.