An Application-specific Checkpointing Technique for Minimizing Checkpoint Corruption

Guanpeng Li*, Karthik Pattabiraman*, Chen-Yong Cher† and Pradip Bose†
*Department of Electrical and Computer Engineering, University of British Columbia (UBC), Vancouver, Canada
†IBM T.J. Watson Research Center, New York, USA
*{gpli,karthikp}@ece.ubc.ca †{chenyong,pbose}@us.ibm.com

***Abstract*—Checkpointing is widely deployed in computer systems to recover from failures due to both hardware and software errors. However, as faults propagate, checkpoints may become corrupted by saving erroneous states and make errors unrecoverable, especially at aggressive checkpoint frequencies. In this paper, we proposed a technique that automatically analyzes a given program to guide checkpoint strategies in order to minimize checkpoint corruptions.**

To understand checkpoint corruptions, we first perform a large-scale fault injection study across ten benchmark applications. We then classify checkpoint corruptions, and comprehensively characterize the fault propagations leading to these corruptions. Leveraging these findings, we build RECOV, a compiler-based tool that automatically identifies the program locations that have lowest density of fault propagation for placing checkpoints, and combines it with low-overhead protection techniques. Our experimental results shows that RECOV can eliminate nearly 92% of the checkpoint corruptions with about 5% performance overhead. RECOV reduces the unavailability of the system by 8.25 times even at very aggressive checkpoint frequencies, showing that it is effective in practice.

I. INTRODUCTION

Checkpointing is widely deployed in today's computer systems to deal with hardware and software errors. To ensure high availability of critical systems, researchers have investigated high frequency checkpointing to achieve rapid recovery [9], [18], [26], [22], [26], [28]. In these high frequency checkpointing schemes, checkpoints are taken every few milliseconds to enable faster recovery from failures.

One of the biggest sources of errors in computer systems is hardware transient faults. As transistor sizes decrease, the rate of transient faults is projected to increase drastically [4], [8]. Unlike traditional systems that mask hardware faults from software through techniques such as Dual Modular Redundancy (DMR), researchers have predicted that future hardware will expose many of these faults to the software to maintain energy consumption thresholds [10], [20].

A hardware fault can cause many kinds of failures, such as crashes, Silent Data Corruptions (SDCs) and hangs. Of these categories, crashes are the most frequent [27], [15]. However, past research has substantially ignored crashes as it was assumed that checkpointing techniques can easily recover from crash-causing errors. Unfortunately, this is not always the case, as crash-causing faults can propagate to checkpoints and corrupt them before causing a crash. This makes the errors unrecoverable from the checkpoint since the program will continue to crash upon recovery from a corrupted checkpoint.

In general, the probability of checkpoint corruption increases as checkpoint frequency increases. In high-frequency checkpointing techniques, checkpoint corruption becomes a major problem. Unfortunately, prior work has either ignored the problem completely [18], or has attempted to solve the problem by keeping two checkpoints [26], [2] (i.e., dual checkpointing), so that even if one checkpoint is corrupted, the other checkpoint can be used for recovery. While this can somewhat decrease checkpoint corruptions, the problems still remain: (1) Faults may still propagate to both checkpoints at high checkpoint frequencies. The interval chosen between the checkpoints to bypass fault propagations is highly application-specific, and the information of distribution of fault propagation is not easily available [15], [26]. (2) Keeping two checkpoints incurs double the memory overhead, thereby straining hardware resources and increasing costs [18].

In this paper, we aim to minimize checkpoint corruptions even while keeping only a single checkpoint, thereby achieving low memory overhead, and at the same time allowing high checkpoint frequencies. There are three main insights underlying our work, namely (1) crash-causing locations in a program fall into only a few dominant patterns, which allows us to comprehensively and automatically identify them, (2) very few crash-causing faults propagate for a long time and we can identify and selectively protect these long-latency crash (LLC) locations using our prior work [15], and (3) there are many natural locations in a program in which very few crash-causing faults propagate across, so that placing checkpoints in these locations will minimize the probability of corruption - we call these locations quiescent states. Our technique therefore uses a combination of selective protection of the LLC causing locations, and identification of quiescent states in the program for checkpoint placement to minimize the probability of checkpoint corruption while keeping only a single checkpoint. We implement our technique in a tool called RECOV that performs static and dynamic analysis of the program to identify the LLC causing locations and quiescent states for placing checkpoints. RECOV is implemented in the *LLVM* open-source compiler [14] and is completely automated, requiring no annotations from programmers. *To the best of our knowledge, we are the first to build an automated technique to minimize checkpoint corruption on an application-specific basis, without multiple checkpoints even at very aggressive checkpoint frequencies.*

We make the following research contributions in this paper:

- We measure the checkpoint corruptions at different checkpoint frequencies through a large-scale fault injection experiment. We find that (1) there is a non-negligible rate of checkpoint corruptions due to fault propagation at aggressive checkpoint frequencies, and (2) checkpoint corruptions are highly variable across applications as they depend on the program's properties. Hence there is a need for application-specific techniques to minimize checkpoint corruption.
- We propose RECOV, a technique that uses static

analysis to identify the program points in which there is minimal fault propagation (we call them *Quiescent States*) for placing checkpoints, and combines this with low-overhead protection mechanisms to minimize checkpoint corruption rate.

- We implement RECOV in the LLVM compiler, and evaluate it in terms of its ability to minimize the checkpoint corruption rate at different checkpoint frequencies, as well as the performance overhead incurred.
- We evaluate the availability improvement provided by RECOV, and compare it with the improvement provided by the dual-checkpoint scheme such as the one proposed by prior work [26].

We find that RECOV can eliminate 91.8% of the checkpoint corruptions in a program even at aggressive checkpoint frequencies (e.g., 1000 instructions) with only 5.03% overhead. We further find that at a checkpoint interval of 1,000 instructions, RECOV provides a 8.25 times reduction in the unavailability (1 - availability) over the original application, and 6.01 times reduction over the dual-checkpointing scheme.

II. FAULT MODEL AND BACKGROUND

In this section, we first present our fault model and define the terms used. We then describe checkpoint techniques we are considering, and finally we explain how checkpoints can be corrupted by faults.

A. Fault Model

In this paper, we consider hardware transient faults (i.e., soft errors) that occur in computational components of processors. This includes arithmetic logic unit(ALU), pipeline stages and flip-flops. We do not consider faults that occur in memory, register file or cache, as these can be protected by ECC. Similarly, we do not consider faults in processor's control logic as we assume that it is protected. Finally, we do not consider faults that occur in the instruction's encoding as these faults can be detected through error correction codes. Our fault model is in line with other work in the area [12], [10], [25]. While we have focused on hardware faults in this paper, our technique is general and can be easily extended to software faults.

B. Terms

We use the following terms in our paper.

- **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 failure.
- **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 fault latencies.
- **Checkpoint Corruption**: Checkpoint saves erroneous values in the program. This because the checkpoint is taken after fault activation and before the fault causes a crash, and hence captures the erroneous value. In general, the longer the crash latency of a fault, the higher the probability it can corrupt a checkpoint during its propagation.
- **Checkpoint Corruption Rate**: The probability of a checkpoint being corrupted due to a randomly occurring fault in the program that ultimately crashes the program (after corrupting the checkpoint).

C. Checkpointing Techniques

Checkpointing is a commonly deployed recovery method in computer systems. Checkpointing systems can be categorized based on, (1) the content saved in checkpoints and, (2) the location where checkpoints are taken, and (3) the number of checkpoint copies kept. In terms of the content saved, checkpoints can be categorized into (1) user-level checkpoints, and (2) system-level checkpoints. User-level checkpoints selectively save the program's state that's necessary for restarting the program. On the other hand, system-level checkpoints simply save all the architectural registers and memory data since the checkpointing system has no knowledge about the programs' state. In terms of the locations where checkpoints are taken, there are two categories: (1) application-specific checkpoints, and (2) application-generic checkpoints. Application-specific checkpoints are placed at specific locations in the program, whereas the application-generic checkpoints are usually taken periodically and do not use any information from the program. This makes application-generic checkpoints much more predictable in terms of recovery latency in case of a failure. In terms of the number of copies kept, checkpoints can be categorized into (1) single-checkpointing scheme, and (2) dual-checkpointing scheme. In single-checkpointing scheme, there is only one checkpoint copy kept at any time. In contrast, the dual-checkpointing scheme keeps two checkpoints so that if one checkpoint copy is corrupted, it can roll back to the earlier one which may be uncorrupted. However, the dual-checkpointing scheme incurs high memory overheads [18], [2], and may not be able to avoid checkpoint corruptions in many applications [15], [26]. While dual checkpointing can be generalized to multiple checkpoints, most prior work has not done so due to the high cost of keeping multiple checkpoints.

In this paper, we consider system-level checkpoints which save all the visible states of program, and application-specific checkpoints placed by an automated system to minimize checkpoint corruptions. Although we consider application-specific checkpoints, we show that the checkpointing strategy we propose can take checkpoints periodically, and hence have the same advantages as application-generic checkpoints. We consider system-level checkpoints as we do not assume apriori knowledge of what states are important to the program. Finally, we consider single-checkpointing scheme so that the checkpoint system has low memory overhead.

D. Our Earlier Work: CRASHFINDER

One of the main reasons for checkpoint corruption is long latency crash (LLC) causing faults , or faults that propagate for a long time before causing crashes. In our earlier work [15], we proposed an automatic technique, CRASHFINDER, to identify

LLC-causing locations in program. CRASHFINDER first statically analyzes a program and identifies all possible locations of faults that can propagate to memory and cause crashes. These locations are categorized based on their type, into pointer, index variable and global variable. CRASHFINDER then systematically samples the identified locations through fault injections to filter out false-positives which do not cause long-latency crashes. It achieves a recall of 92.47% with 100% precision. In other words, CRASHFINDER precisely identifies 92.47% of LLC-causing faults in a program without any false-positives. It also achieves nine orders of magnitude speed-up over exhaustive fault-injection based techniques for identifying these faults. In this paper, we use CRASHFINDER to identify long-latency causing locations and selectively protect them.

E. Fault Injection

Fault injection is a process to introduce errors into the system to study the behavior of the system under errors. It can be done at different level of the system such as at the gate-level, circuit-level, architecture level and application level. Prior work [7] has found that there may be significant differences in the raw rates of faults exposed to the software layer when fault injections are performed in the hardware. However, we are interested in faults that are not masked by the hardware and make their way to the application. Therefore, we inject faults directly at the application level.

Since we consider transient errors that occur in computational components, we inject single bit flips in the return values of the target instruction randomly chosen at runtime. We consider single bit flips as this is the de-facto fault model for simulating transient faults in the literature [10], [12]. However, our technique can be also extended for multi-bit flips. We will consider such multi-bit flip faults in future work.

F. LLVM Compiler

In this paper, we use the LLVM compiler [14] for performing the static analysis to determine which program locations lead to crashes and transforming target programs based on our proposed technique. 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 crashes 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. This also allows us to build a platform independent toolchain which implements our proposed technique to minimize checkpoint corruption. Finally, LLVM has been shown to be a good match for doing fault injection studies [27].

III. INITIAL FAULT INJECTION STUDY

In this section, we perform an initial fault injection study for charactering checkpoint corruptions. The goal of this study is to understand how checkpoints can be corrupted due to fault propagation in order to mitigate the corruptions (explained in Section IV). We first explain the setup of our fault injection experiment in Section III-A, and then present the results in Section III-B. Finally, we examine the typical code patterns that lead to checkpoint corruptions in programs.

A. Fault Injection Experiment

We perform fault injections using the LLFI fault injector, which operates on the LLVM IR code [27]. We inject faults as single bit flips into the destination registers of the program's dynamic instructions. Both the target bit and target dynamic instruction are randomly chosen from the set of all dynamic instructions executed by the program. The way we inject faults ensures that faults are activated right away once we corrupt the target value. We categorize the failure outcomes into Silent Data Corruptions(SDCs), crashes, hangs and benign in our experiment. SDCs lead to different program outputs from the fault free runs, while benign faults are ones that are masked and have no effect on the program output. Hangs are due to programs deadlocking or running much longer than normal. Since hangs are very few, we ignore them in our experiment. Crashes raise hardware traps or exceptions such as reading outside legal memory segments. We focus on crashes as these are what checkpointing techniques target.

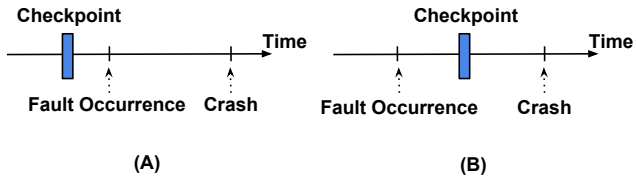

Fig. 1: Examples of Uncorrupted Checkpoint and Corrupted Checkpoint based on Fault Occurrence

To estimate checkpoint corruptions at different checkpointing frequencies, we simulate the act of taking a checkpoint by marking program locations corresponding to the checkpoint frequency (we measure checkpoint frequency in terms of dynamic LLVM instructions rather than wall clock time to ensure they are platform neutral). We then inject faults into the program and check if the fault causes a crash. For the faults that cause a crash, we inspect the location where the crash occurred, and determine if the location was beyond the marked checkpoint. If so, we consider the checkpoint as corrupted, as we assume system-level checkpointing in which all program states are written to the checkpoint. The fact that the fault caused a crash implies that some program state was corrupted by the fault at the time of the checkpoint, and consequently the state will be written to the checkpoint and corrupt it.

For example, in Figure 1(A), the checkpoint is taken before *Fault Occurrence*, hence the checkpoint is not corrupted as there is no fault that propagates to it. As a result, the program can be safely recovered from the *Crash*. However, in Figure 1(B), the checkpoint is taken after *Fault Occurrence* and before the *Crash*. This makes the checkpoint corrupted as the checkpoint captures all program states including the erroneous values introduced by the fault. Therefore, the program may not be able to recover from the checkpoint.

Our benchmark applications are chosen from the SPEC [13], PARBOIL [23], SPLASH-2 [29] and PARSEC [3] suites. We choose eight arbitrary programs from these suites. We also choose two more open-source applications that are widely used in the scientific computing domain, anmely *hercules* and *PureMD* [1], [24]. The details of the benchmarks are explained in Section V-A. We inject a total of 3,000 faults in each application at each checkpoint interval. This yields an error bar ranging from 0.06% to 0.6% depending on the application, in estimating the checkpoint corruption rate at the 95% confidence interval, which is tight enough for our purposes.

An important consideration is the minimum checkpoint interval we need to consider for aggressive checkpoints. We profile the number of CPU clock cycles taken to execute an LLVM IR instruction using the *RDTSC* instruction. We divide the total number of clock cycles executed by the program by the number of dynamic LLVM IR instructions executed. We find that on average, an LLVM instruction takes 184 CPU clock cycles on our x86 machine. Many checkpointing systems consider checkpoint intervals at the granularity of hundreds of thousands CPU cycles [22], [18], say 1,000,000 CPU cycles. This is roughly on the order of tens of thousands of LLVM instructions on our Intel Xeon machine. Considering increasing fault rates [4], as well as increasing CPU performance leading to faster checkpoints, we expect checkpointing systems of the future will take checkpoints on the order of thousands of instructions. Therefore, we start with a checkpoint interval of 1000 LLVM IR instructions.

In total, we consider seven different checkpoint intervals for each application. They are 1,000, 5,000, 10,000, 50,000, 100,000, 500,000 and 1,000,000 dynamic instructions. Therefore we inject a total of 210,000 faults in our initial fault injection study (10 benchmarks * 7 checkpoint intervals * 3000 faults).

B. Fault Injection Results

As mentioned earlier, hangs were negligible in our experiment and are not reported. We find that on average, crashes constitute about 28.28% of the faults, SDCs constitute 4.89%, and the remaining are benign faults (about 66%). We focus on crashes in the rest of this paper as we aim to study checkpoint corruptions due to crash-causing faults.

Figure 2 shows the results of checkpoint corruption rates at 7 different checkpoint intervals for each benchmark. For some of the benchmarks, the total number of executed instructions is lower than the checkpoint intervals considered. For example, the *blackscholes* program executes only 48,000 instructions in total, and hence there are no checkpoint corruptions measured when the checkpoint interval exceeds this number. As shown in the figure, we make two observations. First, as the checkpoint interval decreases (i.e., checkpoint frequency increases), the checkpoint corruption rate becomes higher (as expected). This is especially so at very aggressive checkpoint intervals (1,000 dynamic instructions), where the checkpoint corruptions vary from 0.19% to 10.17%. Therefore we cannot assume checkpoint corruptions are negligible at aggressive frequencies. Secondly, the checkpoint corruption rate is program specific, meaning it varies from one application to another depending on the application's properties. Therefore, we need an application-specific method to analyze the program and mitigate checkpoint corruptions based on the application's properties.

To better understand the reasons for checkpoint corruption, we divide the crash-causing faults leading to checkpoint corruptions into two categories: (1)*Long-latency Crashes (LLCs)*, and (2)*Short-latency Crashes (SLCs)*. Recall that we define *LLCs* as the crashes with latencies of more than 1,000 dynamic instructions - these crashes are usually caused by faulty values being written to memory and being used later. On the other hand, *SLCs* have latencies of fewer than 1,000 instructions, and are usually caused by faulty values communicated in registers alone. Due to the temporal locality of register access, *SLCs* usually have crash latencies ranging from one to hundreds of dynamic instructions.

Figure 3 shows the distribution of the categories of crash-causing faults that lead to checkpoint corruptions for each benchmark at 1,000, 100,000 and 1,000,000 dynamic instruction checkpoint intervals. As seen in the figure, about 83.83% checkpoint corruptions are due to *LLCs* on average in the four checkpoint intervals. However, as the checkpoint interval decreases, checkpoint corruptions due to *SLCs* become more prevalent in some applications. For example, in *libquantum*, *cutcp*, *stencil*, *Hercules* and *ocean*, at a checkpoint interval of 1,000 instructions, about 50%, 100%, 66.66%, 48.71% and 68.75% checkpoint corruptions are caused by *SLCs* respectively. The main reason for this is that there are lots of memory access (e.g, array operations) inside loops in these applications - hence faults that occur in these occurrences cause *SLCs* and corrupt checkpoints with higher probability. *Therefore, at aggressive checkpointing frequencies, both SLCs and LLCs become prominent and need to be mitigated in many applications.*

C. Code Patterns Leading to Checkpoint Corruptions

As we mentioned in the previous section, we categorize checkpoint corruption causing crashes into two categories, namely, *SLCs* and *LLCs*. We illustrate the two categories using a fragment of simplified C code found in the *ocean* benchmark as a running example in Figure 6. We also show its control flow graph and the assembly code translated from the C code. The example has two loops in the function *jacobcalc2* at line 4 and line 8 respectively (*L1* and *L2*).

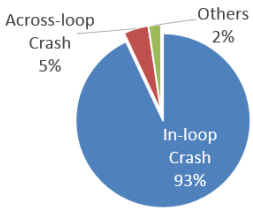

Fig. 4: Distribution of *Short-latency Crash* Patterns

```
1  void jacobcalc2(...){
2    ...
3    double* t2c = z[im-1];
4    for(i=firstrow;i<=lastrow;i++){
5      ...
6      t1c = (double *) t2b[i];
7      ...
8      for (iindex=firstcol;iindex<=lastcol;iindex++) {
9        ...
10       if(...){
11         ...
12         f1 = t1b[iindex+1] + ...;
13         f2 = t1d[iindex-1] * ...;
14         ...
15         t1c[iindex] = factjacob*(f1+f2+f3+f4+f5+f6+
       f7+f8);
16         ...
17       }else{
18         ...
19       }
20     }
21     ...
22   }
23   ...
24   t1c = (double *) t2c[0];
25 }
```

Fig. 5: Running Example from *ocean*

Intuitively, there are two factors that determine whether a fault will corrupt a checkpoint: (1)The more often a crash-causing location is executed, the higher the chance for faulty values originating in the location to propagate to checkpoints.

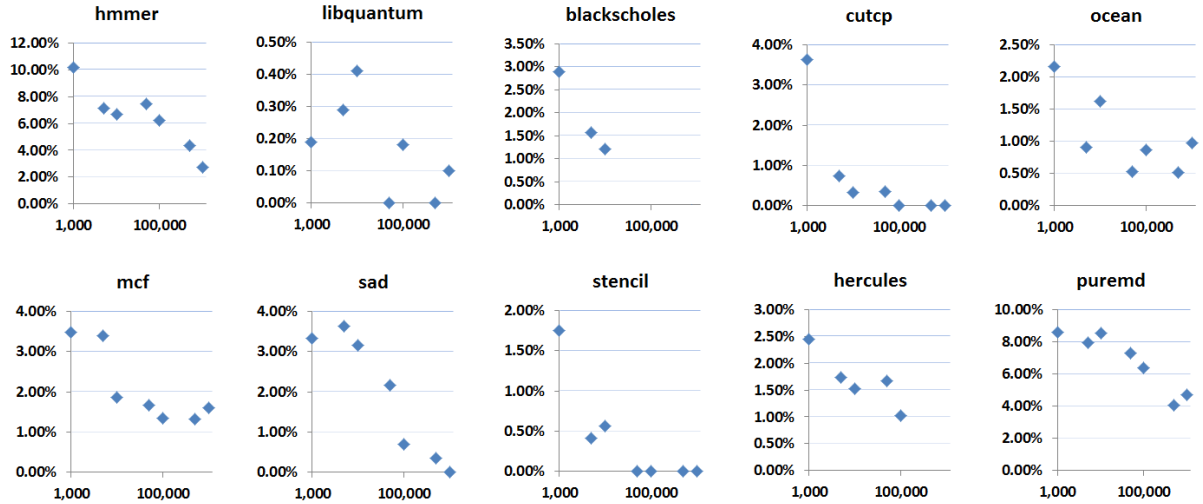

Fig. 2: Checkpoint Corruption Rates at Checkpoint Intervals of 1,000, 5,000, 10,000, 50,000, 100,000, 500,000 and 1,000,000 Dynamic Instructions with Error Bar from 0.06% to 0.6%

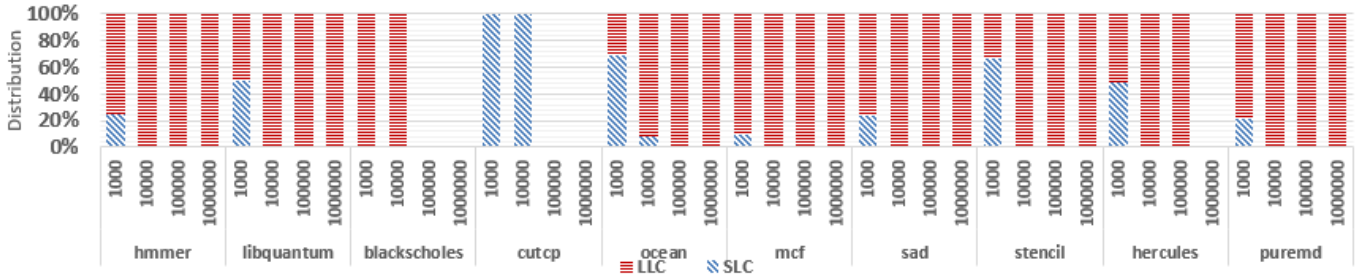

Fig. 3: Distribution of Crash-causing Faults Leading to Checkpoint Corruptions at Checkpoint Intervals of 1,000, 10,000, 100,000 and 1,000,000 Dynamic Instructions (Empty bars indicates no checkpoint corruption)

(2)The longer a fault propagates, the higher the chance for the fault to propagate to checkpoints.

SLCs usually occur when a value in a register is used as an address in a memory operation. These values are communicated within registers and lead to crashes before being stored back to memory. Due to the temporal locality of register access, *SLCs* usually have latencies less than few hundreds of instructions. Although they have shorter crash latencies, some of the crash-causing locations of *SLCs* may be on the hot paths of the program, and therefore have a higher chance to corrupt checkpoints.

In order to investigate code patterns leading to *SLCs*, we randomly choose five applications from the ten benchmarks in our study. They are *hmmer, liquantum, blackscholes, cutcp* and *ocean*. We manually inspect all the cases of checkpoint corruptions due to *SLCs* and find that the locations that lead to checkpoint corruptions typically occur within loop. This is because the majority of the program's dynamic instructions are executed by loops. Therefore, based on the relative location of a crash-causing locations, we further categorize *SLCs* into two dominant patterns: (1) *In-loop Crashes* and (2) *Across-loop crashes*. The distribution of the patterns are shown in Figure 4. We find that most (93%) of the checkpoint corruptions due to *SLCs* fall into the sub-category of *In-loop Crash*.

1)In-loop Crashes happen when a fault occurs at a crash-causing location inside a loop, and the resulting crash occurs in the same loop iteration. For example, in Figure 6, if we inject a fault at ID4, the faulty value will be used as a memory address in the *load* instruction at ID8, thereby causing a crash at ID8 in the same iteration. Therefore, any checkpoint taken between ID4 and ID8 will be corrupted.

2)Across-loop Crashes occur when a memory address value has a fault before entering the loop, and then the value is used within the loop to access memory. For example, in Figure 6, *t1c* at ID2 is defined in *bb2* before entering the loop *L2*, and it is used in *L2* at ID6 and then ID7. This corresponds to lines 6 and 15 in Figure 5. If we inject a fault at ID2, it will likely crash at ID7.

We find that *In-loop Crashes*, especially the ones in the inner-most loop such as *L2*, are responsible for 93% of checkpoint corruptions among *SLCs* (40 of 43 cases). This is because inner-most loops generate the majority of dynamic instructions at runtime. For example, if both *L1* and *L2* have the same number of static instructions (say 100 instructions) and iterate the same amount of times (say 100 iterations), then 99% of the total instructions are from the inner-most loop *L2*. On the other hand, *Across-loop Crashes* are much less frequent - we observe only 2 such cases among the 43 cases.

LLCs usually occur when a corrupted value is written to memory, and causes a crash later when the value or some other affected value is loaded and used in memory address operations. For example, in Figure 6, if we inject a fault into *t2c* at ID1, the faulty value will be saved to memory as a pointer. Later the pointer will be loaded by ID10 and will

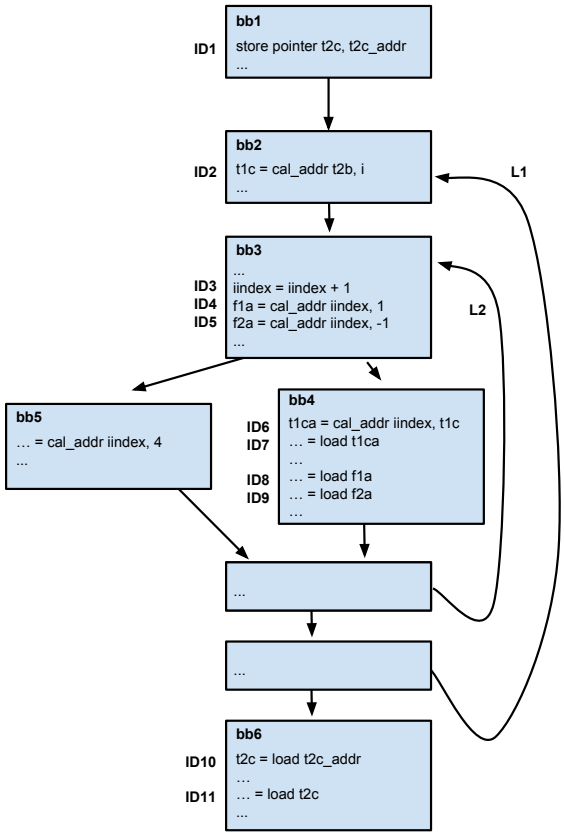

Fig. 6: Control Flow Graph of the Running Example

likely cause a crash at ID11. This corresponds to lines 6 and 24 in Figure 5. These faults have a longer crash latency as the faulty value propagates out of registers' communication and goes into memory. A more detailed explanation of LLCs and their causes can be found in our earlier work [15].

IV. Approach

In this section, we describe our proposed technique, RECOV, to minimize the checkpoint corruption rate of a program. The main idea is to find locations in program where there is limited propagation of crash-causing faults for placing checkpoints - we call such natural locations in program *Quiescent States*. If we can find many such *Quiescent States* in the program, we can aggressively take checkpoints with few checkpoint corruptions.

The main insights to identify *Quiescent States* are gathered from Section III, and are recapped here: (1) Inner-most loops have the most number of crash-causing locations, and hence *In-loop Crashes* are the most prevalent.(2) Most crash-causing locations in inner loops are updated in every iteration before being used in the loop body. Hence registers holding these values are rarely live out across loop iterations, and (3) Only very few faults propagate to the memory and cause *LLCs*.

From the first two observations, we can conclude that *Quiescent states occur at the ends of loop iterations of inner-most loops*. Therefore, one can bypass SLCs by placing checkpoints at the ends of inner-most loops. Because inner loops are often small, they have few instructions in the loop bodies (as we demonstrate later), and hence even at our most aggressive checkpointing frequency of 1000 instructions, we are able to take checkpoints frequently by placing checkpoints at the end of inner loops. From the third observation, we can conclude that LLCs are infrequent, and hence can be prevented by protecting the LLC causing locations at low overhead. We use our earlier work on CRASHFINDER [15] to find locations that can cause LLCs, and selectively protect them by duplicating their backward slices. We show that the overheads of protecting these locations are low as they are relatively infrequently executed.

In the rest of this section, we explain our approach to avoid checkpoint corruptions due to SLCs, then present the heuristics we use to identify backward slice of LLC-causing locations and their protection.

A. Identification of Quiescent States

To identify quiescent states, we consider the two categories of SLCs below:

(1) In-loop Crashes: As we observed, most checkpoint corruptions due to *SLCs* fall into this category. We find that the crash-causing locations of *In-loop Crashes* usually depend on the induction variables of the loop. Further, the crash often occurs in the same iteration as the fault when these are corrupted. The only exception is the update operation of loop induction variables, which can propagate faulty values across iterations, and must be handled separately.

For example, in Figure 6, in the loop *L2*, *bb3* is a dominator of *bb4* (a dominator is a basic block that is always executed before the block(s) it dominates). In *bb3*, ID4 defines an address values based on the induction variable *iindex*, and this value is used in ID8, and causes a crash. In every iteration of the loop *L2*, as long as *t1a* is not redefined after ID8, the crashes at ID8 will likely happen in the same iteration as the one in which the fault occurs. However, if a fault occurs on the induction variable *iindex* at ID3, even though it is calculated at the beginning of the loop, the faulty value will propagate to every iteration. Consequently, it either causes a crash in the subsequent memory access or may aggressively corrupt some parts of the stack, and cause a *LLC*. Since the update operations of induction variables only have a small number of instructions, we can protect the update operation of the induction variable, in this case ID3. Therefore, placing a checkpoint at the end of a loop will bypass most *In-loop Crashes*, and these are identified as *Quiescent States*.

(2) Across-loop Crashes: These cause relatively fewer SLCs than *In-loop crashes*. In this case too, we take checkpoints at the end of loops. For example, if we inject a fault at ID2 before entering the loop *L2*, it will likely cause a crash at ID7 if *bb4* is executed in the first iteration of the loop *L2*. Then taking a checkpoint at the end of the loop *L2* can bypass the propagation of the crash. On the other hand, If *bb5* is executed in the first iteration, the erroneous value *t1c* will not be used and no crash will occur. In this case, if we take a checkpoint at the end of the loop *L2*, the checkpoint will be corrupted. We choose to ignore this latter case as *Across-loop crashes* only constitute a small percentage of checkpoint corruptions, which makes this case very rare. Therefore, we place checkpoints at the ends of loop iterations to mitigate many but not all of this category of SLCs.

In summary, we find that placing checkpoints at the ends of loop bodies can bypass the majority of the propagations of *SLCs*, in both the in-loop and across-loop sub-categories.

B. Protection of LLC causing locations

As mentioned, we use our prior work, CRASHFINDER [15], to identify the LLC-causing locations. We then duplicate the

backward slice of these LLC-causing locations, and insert a check comparing the value computed by the duplicated code and the original value when they are defined. A mismatch between the values indicates an error in the LLC-causing location, and triggers a checkpoint rollback.

The target LLC-causing locations fall into three types: (1) induction variables, (2) address pointer values, and (3) global variables. We first identify backward slice of each LLC-causing location and stop at procedural boundaries - this is similar to what prior work has done [17]. However, only identifying intra-procedural backward slices may affect coverage of the protection. We therefore further trace the backward slice via memory dependencies based on each type as follows:

Induction Variable: There is nothing more to be done in this case as the majority of induction variables are defined and used inside a single loop in a function. Thus identifying intra-procedural backward slice of induction variable achieves high coverage for the protection.

Pointer Variable Since CRASHFINDER identifies all locations that store pointer values to memory, the ones that have memory dependencies have already been included for protection. Thus, most parts of inter-procedural backward slices of this type are identified for protection.

Global Variable: For this type, we connect the individual intra-procedural backward slices that have memory dependencies by identifying their aliasing memory locations. As LLVM reserves a named address to store the value of global variable in memory, statically resolving aliases of global variables is straightforward.

After identifying all the backward slices of the LLC-causing locations, we duplicate the instructions on the backward slices and compare the values at the end of each static dependency sequence to detect any deviation. We evaluate the amount of instructions in the backward slices we compute and runtime overhead of protecting these instructions in Section V.

C. Work-flow of RECOV

Figure 7 shows the work-flow of RECOV to minimize the checkpoint corruption rate for a given program. RECOV takes as input the source code of a program and compiles it to the LLVM IR form. RECOV first identifies all the loops in the program through *static analysis*. Then RECOV finds the last instructions of all exit nodes of the loops, and outputs the locations of these instructions as *Quiescent States*. RECOV then uses our prior technique *CrashFinder* to identify the locations that lead to *LLCs*. Finally, RECOV transforms the source code by duplicating the backward slices of the locations that lead to *LLCs*, and places checkpoint instructions at the *Quiescent States*.

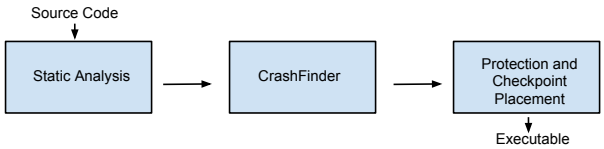

Fig. 7: Work-flow of RECOV

V. EXPERIMENTAL SETUP

We empirically evaluate RECOV in terms of its accuracy and effectiveness in minimizing checkpoint corruption rates. We also measure the performance overheads of RECOV. Our experiments are all carried out on an Intel Xeon-E5 PC running Red Hat Linux, with 64 GB RAM.

We first present benchmarks used in V-A, followed by the research questions in V-B. We then present the methodology to answer each of the research questions in V-C

A. Benchmarks

As mentioned before, we choose a total of ten benchmarks from various domains for evaluating RECOV. All the benchmark applications are compiled and linked into native executables using LLVM, with standard optimizations enabled (O2 optimizations). We present the detailed information of the benchmarks in Table I.

TABLE I: Characteristics of Benchmark Programs used to evaluate RECOV

Benchmark	Benchmark Suite/Author	Description
libquantum	SPEC	A library for the simulation of a quantum computer
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
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
Hercules	Carnegie Mellon University	Finite-element octree-based earthquake simulator by the Quake Group at Carnegie Mellon University
PuReMD	Purdue University	Reactive molecular dynamics simulation program

B. Research Questions

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

RQ1: *How far apart in terms of dynamic instructions are the Quiescent States identified by* RECOV *in programs?*

RQ2: *How much does* RECOV *reduce the checkpoint corruption rate by placing checkpoints at Quiescent States and protecting the LLC-causing locations?*

RQ3: *What are the performance overheads incurred by* RECOV*, at runtime?*

RQ4: *How much reduction in unavailability does* RECOV *provide, and how does it compare to the dual-checkpointing scheme?*

C. Experimental Methodology

We describe our methodology for answering each of the RQs below.

1) Distance between Quiescent States (RQ1): We identify all *Quiescent States* in the program using RECOV, and measure how many instructions are executed on average in between two consecutive *Quiescent States*.

2) Effectiveness of Checkpoint Corruption Minimization (RQ2): We use fault injection to study the effectiveness of RECOV in minimizing checkpoint corruptions. As before, we perform fault injections using the LLFI fault injector [27] as

described in Section III. We inject a total of 3,000 faults at the checkpoint intervals 1,000, 10,000, 100,000 and 1,000,000 instructions for each benchmark, and measure the checkpoint corruption rates after deploying RECOV (we only considered 4 of the 7 intervals we considered earlier due to time constraints). We then compare the minimized checkpoint corruption rates with the checkpoint corruption rates derived from our initial fault injection study in Section III, and quantify how much minimization has been achieved using RECOV.

3) Performance overheads of RECOV *(RQ3):* We measure the runtime overhead incurred by RECOV, as it protects all LLC-causing locations by duplicating their backward slices.

4) Unavailability Reduction (RQ4): We calculate the availability RECOV provides, and compare it with original availability (baseline) and the availability of two variations of the dual-checkpointing scheme (see below) [1]. Because the availabilities are all close to 100% (not surprising as availablity is typically measured in nines), we choose to calculate the unavailability i.e., (1 - availability) and measure its reduction over the baseline for both RECOV and the dual-checkpointing scheme. We consider four checkpointing schemes including RECOV described below.

- **Single-checkpoint Scheme(baseline)** keeps only one checkpoint at any time and always rolls back to this checkpoint as it is only available checkpoint. If the program fails again upon roll-back, the checkpoint is assumed to be corrupted, and the program restarts from the beginning. This is the baseline we use for comparison.
- **Dual-checkpoint Scheme A** keeps the two latest checkpoints at any time. Once a failure occurs, it always rolls back to the earlier checkpoint. If the program crashes again at the same location, the earlier checkpoint is assumed to be corrupted (and so is the later checkpoint, by definition), and hence the program restarts from the beginning.
- **Dual-checkpoint Scheme B** also keeps the two latest checkpoints at any time. If a failure occurs, it attempts to roll back to the later checkpoint. If the failure occurs again at the same location, the program assumes that the later checkpoint is corrupted, and rolls back to the earlier checkpoint. If the failure occurs again, it restarts from the beginning.
- **RECOV** keeps only one checkpoint, to which it rolls back if a failure occurs. If the checkpoint is corrupted, the program restarts from the beginning. We assume that the program is protected using RECOV when this scheme is deployed.

VI. RESULTS

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

A. Distribution of Quiescent States (RQ1)

We present the distribution of the distances between *Quiescent States* in Table II for the ten benchmarks. As can be seen from the table, most applications have *Quiescent States* occurring less than every 50 dynamic instructions on average. The exception is *blackscholes*, which has an average distance of 100 dynamic instructions between *Quiescent States*, this is because the application has larger loop bodies. Further, the majority of applications have a maximum distance between *Quiescent States* of under 310 dynamic instructions - the only exception is *mcf* whose maximum distance is 754. This is due to the existence of large amount of code between two loops in this program. In all benchmarks, the maximum distance between any two *Quiescent States* is less than 1,000 dynamic instructions. Therefore, by placing checkpoints at *Quiescent States*, RECOV is able to support even our most aggressive checkpoint interval of 1000 instructions.

TABLE II: Distance between Quiescent States (In Dynamic Instructions)

Benchmark	Maximum Distance	Average Distance
hmmer	229	21
libquantum	99	8
blackscholes	148	100
cutcp	159	16
ocean	281	23
mcf	754	14
sad	306	45
stencil	141	38
Hercules	158	21
PuReMD	256	16
Average	254	31

B. Effectiveness of Checkpoint Corruption Minimization (RQ2)

Figure 8 also shows the minimized checkpoint corruption rate by using RECOV. As shown, RECOV can completely eliminate checkpoint corruptions in four programs, namely *libquantum, blackscholes, ocean* and *stencil* even at the most aggressive checkpointing interval of 1,000 instructions. For the other programs, although checkpoint corruptions cannot be entirely eliminated at the 1000 instruction interval, the checkpoint corruption rates can be respectively reduced by 30 times, 7.7 times, 3.6 times, 31 times, 5.3 times and 4.2 times over the baseline.

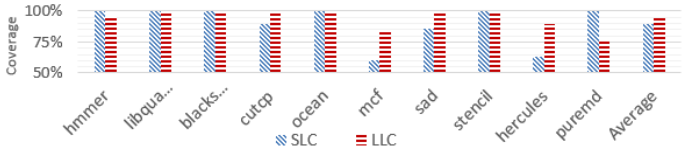

Fig. 9: Breakdown of the Coverage of Checkpoint Corruptions at Checkpoint Interval of 1,000 Dynamic Instructions

We also show the breakdown of the coverage provided by RECOV for checkpoint corruptions due to *LLCs* and *SLCs* respectively in Figure 9. We choose to only measure this at a checkpoint interval of 1,000 dynamic instructions since this interval had the maximum checkpoint corruptions. As shown in the figure, on average, RECOV eliminates 87% of the checkpoint corruptions due to SLCs, and 96% of the checkpoint corruptions due to LLCs. This shows that RECOV eliminates a significant amount of checkpoint corruptions due to both SLCs and LLCs. For two of the benchmarks, *mcf* and *Hercules*, however, coverages for checkpoint corruptions due to SLCs are only 60% and 63.16% . This is because registers that cause SLCs are live out across loop iterations in these programs. Hence our heuristics for mitigating checkpoint corruptions due to SLCs do not cover these cases. Having said that, the

[1]Although prior work on dual checkpointing has not considered such aggressive checkpoint frequencies [26], we choose to evaluate the dual-checkpointing schemes with the same set of checkpoint frequencies as RECOV for standardizing the comparison.

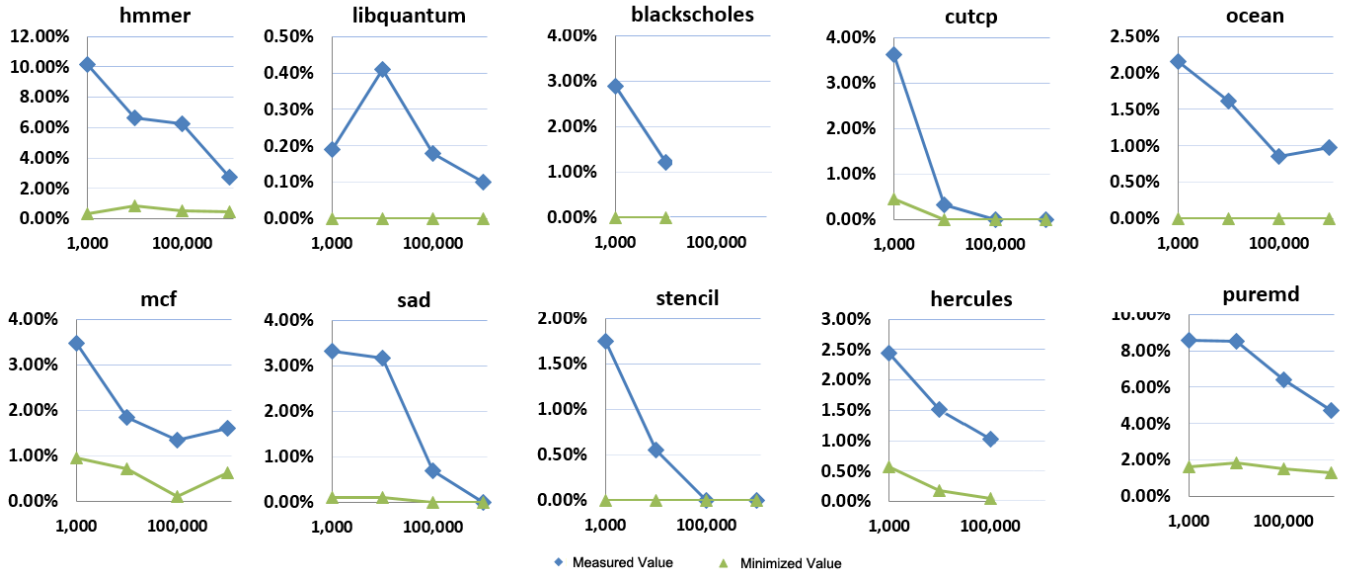

Fig. 8: Minimization of Checkpoint Corruption Rate due to RECOV across Benchmarks

TABLE III: Percentage of Instructions Duplicated by RECOV

mcf	libq.	hmm.	black.	cutcp	ocean	sad	stenc.	herc.	pure.	Avg.
9.79%	9.24%	6.18%	8.18%	22.21%	4.94%	6.96%	2.45%	26.94%	24.42%	9.44%

heuristics work well on average. In another two benchmarks, *PuReMD* and *mcf*, the coverages for checkpoint corruptions due to LLCs are only 76.19% and 86.11% respectively. This is because theCRASHFINDER tool [15], which we use to identify LLCs, also uses heuristics and hence may miss some LLCs that propagate to checkpoints and corrupt them.

C. Runtime Overhead of RECOV (RQ3)

Runtime Overhead: We find that the geometric mean of runtime overheads incurred by RECOV to be 5.03% across all applications. The overheads for individual application are shown in Figure 10. Recall that the runtime overhead of RECOV is due to the protection of the LLC causing locations by duplicating their backward slices. Because only a very small amount of instructions are responsible for *LLCs* [15], the overheads of protecting them are low. The individual overheads vary across applications, for example, *libquantum* incurs 16.85% runtime overhead whereas *stencil* only has 1.45% runtime overhead. This is because different applications have different amount of LLC-causing locations and instructions on their backward slices [15]. Table III shows the number instructions duplicated in the backward slice for each benchmark. On average, RECOV protects 9.44% of dynamic instructions across the benchmarks.

Note that the performance of duplication is not directly proportional to the amount of instruction duplicated. The reason is twofold: (1) duplicating different types of instructions incurs different overheads, and (2) the differences in lengths of static dependency sequences in applications affects the overhead of comparing the values at the ends of the backward slices.

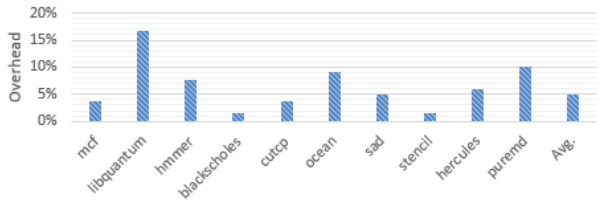

Fig. 10: Runtime Overhead of LLC Protection

D. Unavailability Reduction Provided by RECOV (RQ4)

The appendix derives the expressions for the availabilities of the four schemes we consider including RECOV. These are derived from a probabilistic model seeded with the measured values of the probabilities of checkpoint corruption and the checkpoint intervals, for each application (also shown in the appendix).

Based on these expressions, we calculate the reductions in unavailability provided by RECOV and the dual checkpointing schemes in Table IV. Note that for some of the benchmarks, the total number of executed instructions is lower than the checkpoint intervals considered and hence there are no checkpoint corruptions when the checkpoint interval exceeds this number. We indicate such rows by '-' in the table. We consider two variants of the dual checkpointing scheme as explained earlier. As can be seen from the table, the availabilities of both these schemes are similar. Therefore, we only consider the dual-checkpointing scheme 'B' for our explanation below.

On average, RECOV reduces the unavailability over the baseline at checkpoint intervals of 1,000, 10,000, 100,000 and 1,000,000 instructions respectively by factors of 8.25, 2.90, 2.14 and 3.03. In comparison, the dual-checkpoint scheme reduces the unavailability over the baseline by 1.37, 1.05, 1.03 and 1.28 times at the same checkpoint intervals. This is much lower than the unavailability reduction provided by RECOV.

Note that in some applications the dual-checkpointing scheme has higher unavailability than the baseline, for ex-

TABLE IV: Unavailability Values of the four schemes

Benchmark	C.K. Interval	Single-checkpoint (baseline)(%)	Factor of Reduction		
			Dual-checkpoint A	Dual-checkpoint B	ReCov
hmmer	1,000	1.4434	1.32	1.32	34.62
	10,000	0.9486	1.00	1.00	8.79
	100,000	0.8929	1.10	1.10	12.86
	1,000,000	0.4019	1.69	1.75	6.53
libquantum	1,000	0.0203	253.45	504.98	802.28
	10,000	0.0442	1.23	1.24	174.32
	100,000	0.0232	2.90	5.77	9.17
	1,000,000	0.0508	0.63	1.26	2.00
blackscholes	1,000	0.2919	0.71	0.97	2.68
	10,000	1.2691	0.52	0.97	1.18
	100,000	-	-	-	-
	1,000,000	-	-	-	-
cutcp	1,000	0.1662	7.69	7.69	8.13
	10,000	0.0340	2129.49	4196.86	4503.75
	100,000	0.0148	92.58	183.99	195.81
	1,000,000	0.0164	10.28	20.43	21.75
ocean	1,000	0.2654	3.22	3.22	13532.89
	10,000	0.1994	1.09	1.09	1016.75
	100,000	0.1082	0.98	1.00	55.16
	1,000,000	0.1441	0.86	1.00	7.35
mcf	1,000	0.5521	1.07	1.07	3.77
	10,000	0.2952	1.06	1.06	2.62
	100,000	0.2159	1.18	1.18	11.43
	1,000,000	0.2608	1.07	1.09	3.11
sad	1,000	0.4718	1.31	1.31	35.30
	10,000	0.4480	1.35	1.35	33.11
	100,000	0.1017	0.98	1.00	55.44
	1,000,000	0.0215	0.50	1.00	1.17
stencil	1,000	0.2963	1.51	1.51	40987.99
	10,000	0.0951	3.10	3.10	1315.25
	100,000	0.0007	0.50	1.00	1.02
	1,000,000	0.0074	0.50	1.00	1.02
hercules	1,000	0.7127	1.62	1.87	5.45
	10,000	1.0169	0.63	0.97	2.51
	100,000	6.0716	0.54	0.98	1.67
	1,000,000	-	-	-	-
puremd	1,000	1.3796	1.23	1.23	6.36
	10,000	1.3654	1.05	1.05	5.36
	100,000	1.0291	1.19	1.22	4.51
	1,000,000	0.7683	0.95	1.24	2.47
Average	1,000	0.5600	1.32	1.37	8.25
	10,000	0.5719	0.80	1.05	2.90
	100,000	0.9432	0.64	1.03	2.14
	1,000,000	0.2476	1.03	1.28	3.03

ample in *blackscholes*. This is because it incurs longer recovery latency as it rolls back to the earlier checkpoint. In summary, RECOV significantly outperforms the dual-checkpointing scheme in terms of unavailability reduction for all applications considered.

VII. RELATED WORK

EDDI [16] and SWIFT [20] are compiler-based techniques that use full duplication to protect program data. Full duplication can achieve high coverage but incurs significant performance overhead. Feng et al. [10] 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 consider the impact of hardware faults on checkpoint corruption, and consequently overprotect the application, resulting in high overheads.

ReVive [18] and SafetyNet [22] focus on the efficient implementation of checkpoint systems taking checkpoints at aggressive frequencies. They provide a hardware and software co-designed solution to mitigate overheads incurred from taking checkpoints frequently. Sorin et. al. [21] and Cao et. al. [5] propose checkpointing algorithms that allow fast recovery in order to minimize recovery latency from failures. While these techniques are useful to support aggressive checkpointing and fast recovery, they do not address the issue of checkpoint corruptions.

Gu et. al. [11] found through fault injections in the Linux kernel, that a small amount of faults can propagate for long time and may need mitigation. Yim et. al. [11] highlighted the issue that faults that propagate for long may corrupt checkpoint thereby making the errors unrecoverable. Chandra et. al. [6] experimentally observed the rates of checkpoint corruptions due to hardware and software errors at different checkpoint frequencies and checkpointing methods. In very recent work, Ramachandran et. al. [19] coordinate error detection and checkpoint intervals to mitigate corruption of external outputs. However, none of the above papers specify how to identify the faults propagating to checkpoints and prevent them from corrupting checkpoints. Our prior work [15] proposed an automatic technique to identify faults causing *LLCs* in programs. However, it does not consider mitigation of checkpoint corruption due to SLCs, nor does it quantify the mitigation. As we have seen in this paper, SLCs can contribute to a significant portion of checkpoint corruption at aggressive checkpoint frequencies.

Wang et. al. [26] proposed a dual-checkpointing technique to avoid checkpoint corruptions in a virtual machine-based environment. Aupy et. al [2] analyzed propagation latencies of faults, and calculate optimal checkpoint interval for recovery time according to the distribution of the propagation latencies. While these techniques can reduce checkpoint corruptions, both of them are based on the dual-checkpointing scheme. However, as we have shown in Section VI, the dual-checkpointing scheme is not as effective as RECOV in preventing checkpoint corruption and boosting availability. Further, the dual checkpointing scheme is also more expensive in terms of memory.

VIII. CONCLUSION AND FUTURE WORK

In this paper, we found that checkpoint corruptions are non-negligible at aggressive checkpointing frequencies and that they are highly application-specific, thereby necessitating an application-specific scheme for minimizing the corruptions. We comprehensively characterized the patterns of checkpoint corruptions due to crash-causing faults into *SLCs* and *LLCs*. We find *SLCs* often happen in the same loop iteration when the faults occur, and placing checkpoints at the end of loop bodies significantly reduces checkpoint corruptions due to *SLCs*. By protecting *LLCs* causing locations through fine-grained protection techniques, and placing checkpoints at the end of the loop bodies, RECOV eliminates 91.84% checkpoint corruptions across applications. Further, it reduces the unavailability of the system by 8.25 times even at very aggressive checkpoint frequencies, and incurs a runtime performance overhead of only 5%, showing that it is practical. We also find that the checkpoints placed by RECOV can be sufficiently close together to support aggressive checkpointing frequencies.

As future work, we plan to explore integrating our technique with existing checkpointing systems, and also deploying it on a wider range of high-performance computing (HPC) applications to measure the overheads.

IX. ACKNOWLEDGMENT

We thank the anonymous reviewers of ISSRE'15 and Long Wang 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).

REFERENCES

[1] H. M. Aktulga, J. C. Fogarty, S. A. Pandit, and A. Y. Grama. Parallel reactive molecular dynamics: Numerical methods and algorithmic techniques. *Parallel Computing*, 38(4):245–259, 2012.

[2] G. Aupy, A. Benoit, T. Hérault, Y. Robert, F. Vivien, and D. Zaidouni. On the combination of silent error detection and checkpointing. In *IEEE 19th Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 11–20. IEEE, 2013.

[3] 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.

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

[5] T. Cao, M. Vaz Salles, B. Sowell, Y. Yue, A. Demers, J. Gehrke, and W. White. Fast checkpoint recovery algorithms for frequently consistent applications. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 265–276. ACM, 2011.

[6] S. Chandra and P. M. Chen. The impact of recovery mechanisms on the likelihood of saving corrupted state. In *Software Reliability Engineering, 2002. ISSRE 2003. Proceedings. 13th International Symposium on*, pages 91–101. IEEE, 2002.

[7] H. Cho, S. Mirkhani, C.-Y. Cher, J. A. Abraham, and S. Mitra. Quantitative evaluation of soft error injection techniques for robust system design. In *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, pages 1–10. IEEE, 2013.

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

[9] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408, 2002.

[10] 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.

[11] W. Gu, Z. Kalbarczyk, R. K. Iyer, and Z. Yang. Characterization of linux kernel behavior under errors. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 459–459. IEEE Computer Society, 2003.

[12] 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.

[13] J. L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *Computer*, 33(7):28–35, 2000.

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

[15] G. Li, Q. Lu, and K. Pattabiraman. Fine-grained characterization of faults causing long latency crashes in programs. In *45rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2015.

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

[17] K. Pattabiraman, Z. Kalbarczyk, and R. K. Iyer. Automated derivation of application-aware error detectors using static analysis. In *On-Line Testing Symposium, 2007. IOLTS 07. 13th IEEE International*, pages 211–216. IEEE, 2007.

[18] M. Prvulovic, Z. Zhang, and J. Torrellas. Revive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *29th Annual International Symposium on Computer Architecture*, pages 111–122. IEEE, 2002.

[19] P. Ramachandran, S. K. S. Hari, M. Li, and S. V. Adve. Hardware fault recovery for i/o intensive applications. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(3):33, 2014.

[20] 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.

[21] D. J. Sorin, M. M. Martin, M. D. Hill, and D. A. Wood. Fast checkpoint/recovery to support kilo-instruction speculation and hardware fault tolerance. *Dept. of Computer Sciences Technical Report CS-TR-2000-1420, University of Wisconsin-Madison*, 2000.

[22] D. J. Sorin, M. M. Martin, M. D. Hill, and D. A. Wood. Safetynet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *29th Annual International Symposium on Computer Architecture*, pages 123–134. IEEE, 2002.

[23] 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.

[24] R. Taborda and J. Bielak. Large-scale earthquake simulation: computational seismology and complex engineering systems. *Computing in Science & Engineering*, 13(4):14–27, 2011.

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

[26] L. Wang, Z. Kalbarczyk, R. Iyer, and A. Iyengar. VM-μCheckpoint: Design, Modeling, and Assessment of Lightweight In-Memory VM Checkpointing. *IEEE Transactions on Dependable and Secure Computing*.

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

[28] K. Whisnant, Z. Kalbarczyk, and R. K. Iyer. Micro-checkpointing: Checkpointing for multithreaded applications. In *6th IEEE International On-Line Testing Workshop*, pages 3–8. IEEE, 2000.

[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.

X. APPENDIX: AVAILABILITY FORMULA

A. Assumptions

We make three assumptions in calculating the availability. First, we assume that the time taken to capture a checkpoint is negligible compared to the checkpoint interval. This is the case for hardware checkpointing schemes such as ReVive [18], which leverage hardware support to take fast checkpoints. For software techniques, checkpoints can be taken by asynchronous processes or threads, and moved out of the critical path of the program's execution. We do this for all four checkpoint schemes, so the comparison and the evaluation is fair. This assumption is in line with other prior work [26].

Second, we assume that we can always detect whether a checkpoint has been corrupted by a fault, based on the program crashing again at the same location after a restart. While it is possible for checkpoint corruptions to go undetected for a long time, we assume that the corruption is detected before the next checkpoint is taken. Thus we assume that it is unlikely that a fault that had previously caused a crash will remain undetected for a long time after the program is restarted. Again, this assumption is made for all schemes.

Finally, we assume that restarting a program from the beginning is always a possibility even if all the checkpoints are corrupted (in other words, we assume that the program's execution does not have system-wide effects that cannot be undone, such as sending messages in a distributed system, for example). The last assumption is necessary for ensuring that we always can recover the program even if the checkpoint is corrupted. Without this assumption, even a small probability of checkpoint corruption (which all four schemes have) will result in a net steady-state availability of zero.

B. Availability Calculation

We consider the system as available whenever it is doing useful work i.e., it is not repeating the work it did earlier due to a failure. Time spent recovering from a checkpoint is therefore not counted as the system being available (since it is not doing useful work). The availability of the system can

be calculated via the following formula, where *MTTF* is mean time to failure, and *MTTR* is mean time to recover the program.

$$Availability = \frac{MTTF}{(MTTF+MTTR)}$$

TABLE V: Parameters for Calculating *MTTR*

Name	Meaning
C	Checkpoint corruption rate for the single-checkpointing scheme
C_1	Checkpoint corruption rate for the latest checkpoint, but the earlier checkpoint is not corrupted in the dual checkpointing scheme
C_2	Checkpoint corruption rate for the earlier checkpoint in the dual-checkpointing scheme
C_r	Minimized checkpoint corruption rate for RECOV
M	Total number of the dynamic instructions executed by the program
I	Checkpoint interval, in terms of dynamic instructions
R	Crash rate of the program given that a fault occurs for single- and dual-checkpoint schemes
R_r	Crash rate of the program given that a fault occurs for RECOV
λ	Fault rate, which to simulate base fault rate of the program, is set to 1/M. Note the actual fault rate in the real world is independent of M, but to mirror the experiment, we use the fault injection rate as the fault rate.

We first present the symbols we use to represent the systems' parameters in Table V. We calculate the MTTF as $1/(\lambda * R)$ for single- and dual-checkpoint schemes, and $1/(\lambda * R_r)$ for RECOV as we are concerned with crash-causing faults only. We assume that at most one fault occurs during the program's execution, so λ is equal to $\frac{1}{M}$. Note that RECOV has different crash rates as the protection eliminates some of crashes, hence has a different crash rate R_r. We assume that the fault rate is the same though, as RECOV incurs only about 5% overhead. We consider the MTTR for the four cases below.

1) Single-checkpoint Scheme: When a crash occurs and the checkpoint is corrupted, it first tries to recover from the checkpoint. The average work lost is $\frac{I}{2}$. If it finds that the program crashes again at the same location, this indicates the checkpoint is corrupted, so the program restarts from the beginning. In this case, the average work lost is $\frac{M}{2}$. So the total work lost when the checkpoint is corrupted is $(\frac{M}{2}+\frac{I}{2})$. Otherwise, it is $\frac{I}{2}$. Therefore, the *MTTR* for single-checkpoint scheme is $C(\frac{M}{2}+\frac{I}{2})+(1-C)\frac{I}{2}$.

2) Dual-checkpoint Scheme A: In this scheme, when a crash occurs, it first tries to recover from the earlier checkpoint. The earlier checkpoint was taken $2I$ instructions ago. The average lost work is $\frac{2I}{2}$, which is equal to *I*. If the recovery fails, the program restarts from the beginning, here the average lost work is $\frac{M}{2}$. So the total lost work in case the checkpoint is corrupted is $(I+\frac{M}{2})$ Therefore, the *MTTR* for dual-checkpoint A scheme is $C_2(I+\frac{M}{2})+(1-C_2)I$.

3) Dual-checkpoint Scheme B: When a crash occurs, the average work lost is $\frac{I}{2}$. The system then tries to recover from the latest checkpoint. If this fails, the checkpoint is corrupted, and it attempts to recover from the later checkpoint. The average work lost in this case is $(\frac{I}{2}+\frac{2I}{2}) = \frac{3I}{2}$. Finally, if the earlier checkpoint is corrupted as well, it rolls back to the beginning of the program, in which case the average work lost is $\frac{3I}{2}+\frac{M}{2}$. The expression for MTTR is therefore $C_1(\frac{3I}{2})+C_2(\frac{3I}{2}+\frac{M}{2})+(1-C_1)\frac{I}{2}$

4) RECOV*:* Since RECOV keeps only one checkpoint, the availability expression is the same as that of the single-checkpoint scheme (the parameter values are different though). The expression is $C_r(\frac{M}{2}+\frac{I}{2})+(1-C_r)\frac{I}{2}$.

We present the parameter values of all benchmarks in Table VII. The data in this table was obtained through measurements of each benchmark program. Note that some of the values in the table are indicated with a '-'. These correspond to cases where the checkpoint intervals considered exceed the number of instructions executed in the program, and hence there are no checkpoint corruptions.

C. Corner Cases

We consider two corner cases to check the model, namely *I=M* and *I=1*. They correspond to taking no checkpoint and taking checkpoints at every instruction. This corresponds to Scenario 1 and 2 respectively in Table VI. We expect to see similar results for the *MTTR* between the two scenarios in all schemes. This is because if we take a checkpoint every instruction (I=1), the checkpoint will almost certainly be corrupted if a fault occurs, hence the program ends up restarting from the beginning. This is equivalent to taking a checkpoint after the program completes (I=M).

TABLE VI: *MTTR* for Checkpoint Schemes when I=1 & I=M

Scheme	Scenario 1 (I=1)	Scenario 2 (I=M)
Single-checkpoint	*MTTR*=$\frac{M}{2}$, if *C*=0 and *I=M*	*MTTR*=$\frac{M+1}{2}$, if *C*=1 and *I=1*
Dual-checkpoint A	*MTTR*=$\frac{M}{2}$, if *C*=0 and *I*=$\frac{M}{2}$	*MTTR*=$\frac{M+2}{2}$, if *C*=1 and *I=1*
Dual-checkpoint B	*MTTR*=$\frac{M}{2}$, if C_1=C_2=0 and *I=M*	*MTTR*=$\frac{M+6}{2}$, if C_1=C_2=1 and *I=1*
RECOV	*MTTR*=$\frac{M}{2}$, if C_r=0 and *I=M*	*MTTR*=$\frac{M+1}{2}$, if C_r=1 and *I=1*

As expected, Scenarios 1 and 2 both yield similar *MTTR* in all four schemes. The *MTTR* values derived are dominated by $\frac{M}{2}$ since *M* is a much larger value than I. So we consider the resulting *MTTR*s in Scenario 2 as approximately equal to $\frac{M}{2}$. Thus, the model yields consistent values in the corner cases.

TABLE VII: Values of Parameters for Calculating Availability

	I	M(In Million)	C(%)	C_1(%)	C_2(%)	C_r(%)	R(%)	R_r(%)
hmmer	1,000	1602	10.17	2.49	7.68	0.34	28.80	24.53
	10,000	1602	6.65	0.01	6.64	0.88		
	100,000	1602	6.25	0.56	5.69	0.56		
	1,000,000	1602	2.74	1.21	1.53	0.44		
libquantum	1,000	266	0.19	0.19	0.00	0.00	21.37	13.50
	10,000	266	0.41	0.08	0.33	0.00		
	100,000	266	0.18	0.18	0.00	0.00		
	1,000,000	266	0.10	0.10	0.00	0.00		
blackscholes	1,000	0.048	2.89	0.00	2.89	0.00	11.87	10.67
	10,000	0.048	1.22	0.00	1.22	0.00		
	100,000	-	-	-	-	-		
	1,000,000	-	-	-	-	-		
cutcp	1,000	5755	3.62	3.15	0.47	0.47	9.20	8.70
	10,000	5755	0.74	0.74	0.00	0.00		
	100,000	5755	0.32	0.32	0.00	0.00		
	1,000,000	5755	0.34	0.34	0.00	0.00		
ocean	1,000	522	2.16	1.49	0.67	0.00	24.63	20.50
	10,000	522	1.62	0.14	1.48	0.00		
	100,000	522	0.86	0.00	0.86	0.00		
	1,000,000	522	0.98	0.00	0.98	0.00		
mcf	1,000	4128	3.47	0.24	3.23	0.95	32.00	30.87
	10,000	4128	1.85	0.11	1.74	0.73		
	100,000	4128	1.35	0.21	1.14	0.12		
	1,000,000	4128	1.61	0.13	1.48	0.52		
sad	1,000	661	3.33	0.79	2.54	0.11	28.47	24.27
	10,000	661	3.16	0.83	2.33	0.11		
	100,000	661	0.70	0.00	0.70	0.00		
	1,000,000	661	0.00	0.00	0.00	0.00		
stencil	1,000	2305	1.75	0.59	1.16	0.00	33.97	33.33
	10,000	2305	0.56	0.38	0.18	0.00		
	100,000	2305	0.00	0.00	0.00	0.00		
	1,000,000	2305	0.00	0.00	0.00	0.00		
hercules	1,000	0.43	2.44	1.26	1.18	0.57	53.77	32.70
	10,000	0.43	1.52	0.00	1.52	0.19		
	100,000	0.43	1.03	0.26	0.77	0.06		
	1,000,000	-	-	-	-	-		
puremd	1,000	51	8.60	1.60	7.00	1.61	32.53	26.97
	10,000	51	8.51	0.43	8.08	1.88		
	100,000	51	6.39	1.24	5.15	1.55		
	1,000,000	51	4.74	1.57	3.17	1.29		