LetGo: A Lightweight Continuous Framework for HPC Applications Under Failures

Bo Fang
Electrical and Computer Engineering
University of British Columbia
bof@ece.ubc.ca

Qiang Guan
Ultrascale System Research Center
Los Alamos National Laboratory
qguan@lanl.gov

Nathan Debardeleben
Ultrascale System Research Center
Los Alamos National Laboratory
ndebard@lanl.gov

Karthik Pattabiraman
Electrical and Computer Engineering
University of British Columbia
karthikp@ece.ubc.ca

Matei Ripeanu
Electrical and Computer Engineering
University of British Columbia
matei@ece.ubc.ca

ABSTRACT

Requirements for reliability, low power consumption, and performance place complex and conflicting demands on the design of high-performance computing (HPC) systems. Fault-tolerance techniques such as checkpoint/restart (C/R) protect HPC applications against hardware faults. These techniques, however, have non negligible overheads particularly when the fault rate exposed by the hardware is high: it is estimated that in future HPC systems, up to 60% of the computational cycles/power will be used for fault tolerance.

To mitigate the overall overhead of fault-tolerance techniques, we propose *LetGo*, an approach that attempts to continue the execution of a HPC application when crashes would otherwise occur. Our hypothesis is that a class of HPC applications have good enough intrinsic fault tolerance so that its possible to re-purpose the default mechanism that terminates an application once a crash-causing error is signalled, and instead attempt to repair the corrupted application state, and continue the application execution. This paper explores this hypothesis, and quantifies the impact of using this observation in the context of checkpoint/restart (C/R) mechanisms.

Our fault-injection experiments using a suite of five HPC applications show that, on average, LetGo is able to elide 62% of the crashes encountered by applications, of which 80% result in correct output, while incurring a negligible performance overhead. As a result, when LetGo is used in conjunction with a C/R scheme, it enables significantly higher efficiency thereby leading to faster time to solution.

ACM Reference format:
Bo Fang, Qiang Guan, Nathan Debardeleben, Karthik Pattabiraman, and Matei Ripeanu. 2017. LetGo: A Lightweight Continuous Framework for HPC Applications Under Failures. In *Proceedings of HPDC '17, Washington , DC, USA, June 26-30,* 14 pages.
DOI: http://dx.doi.org/10.1145/3078597.3078609

HPDC '17, Washington , DC, USA

DOI: http://dx.doi.org/10.1145/3078597.3078609

1 INTRODUCTION

Transient hardware faults, caused by particle strikes and cosmic rays, have become one of the major concerns for current high-performance computing (HPC) systems [51]. Today's large HPC systems have a mean-time between failure (MTBF) of tens of hours [30], even with hardware- and software-based protection techniques employed together. As the hardware feature sizes shrink and the complexity of the HPC systems increase, the failure rate is expected to further increase, which mandates that a larger portion of computation cycles are used for fault tolerance [15].

Transient hardware faults (i.e., bit flips) often result in application crashes, due to the runtime system detecting the error and terminating the application, thereby losing the application's work. Checkpoint/restart (C/R) is one of the most popular methods to recover from such faults [19, 54, 55] by loading a previously saved intermediate state of the application (i.e., a checkpoint), and restarting the execution. While useful, checkpoint/restart techniques incur high overheads in terms of performance, energy and memory, which will be exacerbated as the failure rate increases [13, 58].

This paper proposes LetGo, which upon detecting an impending crash, attempts to repair the application state to enable it to continue its execution (instead of recovering from a checkpoint). LetGo is based on three observations. First, a large class of HPC applications are, intrinsically, resilient to localized numerical perturbations as they require computation results to converge over time. As a result, they are able to mask some data corruptions. For example, Casas et al. [7] show that the algebraic multi-grid (AMG) solver, which is based on iterative methods, has high intrinsic resiliency. Second, many HPC applications have application-specific acceptance checks (e.g., based on energy conservation laws). These checks can be used to filter out obvious deviations in the application's output, and reduce the probability of producing incorrect results. For example, High Performance Linpack (HPL) solves a linear system using LU decomposition [46] and tests the correctness of the result by checking the residual of the linear system as a norm-wise backward error [29, 41]. Third, most crash-causing errors lead to program crashes within a small number of dynamic instructions, and are hence unlikely to propagate to a large part of the application state [36, 48]. Therefore, the impact of crash-causing faults is likely to be confined to a small portion of the application's state, thus allowing recovery.

Taken together, these observations offer an optimistic hypothesis that it may be possible to re-purpose the default mechanism that terminates an application once a crash-causing error is signalled, and attempt to repair the corrupted application state and continue the application execution. This paper explores this hypothesis, proposes heuristics to repair the application state, and quantifies the impact of using this observation in the context of C/R mechanisms.

To enable this exploration we design and implement *LetGo*. LetGo works by monitoring the application at runtime; when a crash-causing error occurs, LetGo intercepts the hardware exception (e.g., segmentation fault), and does not pass the exception on to the application. Instead, it advances the program counter of the application to the next instruction, bypassing the crash-causing instruction. Further, LetGo employs various heuristics to adjust the state of the application's register file to hide the effects of the ignored instruction and ensure, to the extent possible, that the application state is not corrupted.

Figure 1 illustrates how LetGo can be used in the context of a checkpoint/restart (C/R) scheme. As shown in Figure 1a and 1b, the default action of a C/R scheme on fail-stop failures is to rewind to the last checkpoint. LetGo allows the HPC run-time to continue the execution of an application once a crash-causing error occurs (Figure 1c) and later use application-level correctness test to detect possible state corruption. If the application passes these checks, LetGo assumes that intermediate/final states of an application are correct, and hence no recovery is needed. This reduces checkpoint overheads in two ways: first, LetGo avoids the overhead of restarting from a previous checkpoint upon the occurrence of a crash-causing error; second, since crashes are less frequent, checkpoints can be taken less frequently as well (or not at all if the developer is prepared to accept the risk of unrecoverable failures). The potential cost of LetGo is an increased rate of Silent Data Corruption (SDC) leading to incorrect results. We argue that this may be acceptable for two reasons: first, our experiments indicate that this increase is low (the resulting SDC rate is in the same range as the SDC rate of the original application), and, second, since the possibility of undetected incorrect results exists even with the original application (i.e., without using LetGo), application users independently need to develop efficient techniques to increase confidence in the application results. By leveraging these application checks, LetGo reduces the chances of an error causing a SDC. *To the best of our knowledge, LetGo is the first system that applies the idea of tolerating errors by repairing application state in the context of C/R in HPC applications.*

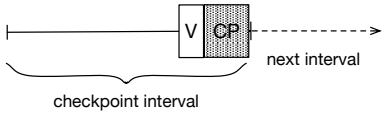
This paper makes the following contributions:

- We propose a methodology to reduce the overhead of C/R techniques for HPC applications by resuming the execution of an application upon the occurrence of a crash-causing error without going back to the last checkpoint.
- We design LetGo, a light-weight run-time system that consists of two main components: a *monitor* that intercepts and handles operating system signals generated when a crash-causing error occurs, and a *modifier* that employs heuristics to adjust program state to increase the probability of successful application continuation (Section 4.1). Importantly, LetGo requires neither modifications to the application, nor the availability of the application's source code for analysis (Section 4.3 and Section 4.2). therefore, it is practical to deploy in today's HPC context.
- We evaluate LetGo through fault-injection experiments using five DoE mini-applications. We find that LetGo is able to continue the application's execution in about 62% of the cases when it encounters a crash-causing error (for the remaining 38%, it gives up and the application can be restarted from a checkpoint as before). The increase in the SDC rate (undetected incorrect output) is low: 0.913% arithmetic mean. (Section 6.1).
- Finally, we evaluate the end-to-end impact of LetGo in the context of a C/R scheme and its sensitivity to a wide range of parameters. We find that LetGo offers significant efficiency gains (1.01x to 1.20x) in the ratio between the time spent for useful work and the total time cost, compared to the standard C/R scheme, across a wide range of parameters.

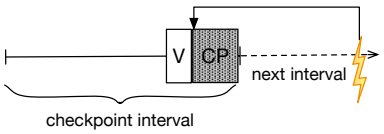
Our evaluation shows that, on average, LetGo is able to continue to completion 62% of the crashes while increasing the overall application SDC rates from 0.75% to 1.6%. This highlights a key contribution of LetGo: it creates the opportunity to trade off confidence in results for efficiency (time to solution). Certainly, for some applications - or for some operational situations - confidence in results is the user's primary concern, and LetGo will not be used. We believe, however, that there are many situations that make LetGo attractive: Firstly, since Silent Data Corruptions (SCD) can occur anyways (due to bit-flips even when LetGo is not used), users of HPC applications are already taking the risk of getting incorrect results, and have developed techniques to validate their results. Application-specific checks to diminish this risk are an active area of research [21, 22, 28, 39] and LetGo will benefit from all these efforts. Secondly, for some applications LetGo performs extremely well (e.g., for CLAMR and SANP all faults that would lead to crashes can be elided by LetGo, without resulting in any additional SDCs). In these cases, LetGo certainly represents an appealing solution. Finally, note that it is trivial to collect information on whether a run has benefited from LetGo repair heuristics and thus offers users additional information base on which to reason about confidence.

2 RELATED WORK

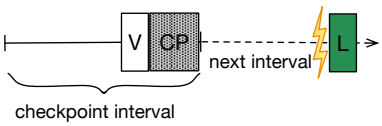
There have been many efforts to provide comprehensive solutions for HPC system to recover from failures. Recovery strategies can be grouped along two dimensions: first, on whether they are application-aware or application-agnostic, and, second on whether they roll-back to previously correct state or use heuristics to attempt to repair state and roll-forward. An example of application-agnostic approach is that of Chien et al. [18] who propose a global view resilience (GVR) framework that allows applications to store and compute an approximation from the current and versioned application data for a forward recovery. Aupy et al. [1] discuss how to combine the silent error detection and checkpointing, and schedule the checkpointing and verification in an optimal-balanced way with a rollback recovery. Other approaches rely on a detailed understanding of the application: Gamel et al. [23] design and implement a local recovery scheme specialized for stencil-based applications for a fast



(a) A standard checkpoint interval without LetGo.



(b) If a crash occurs, the HPC run-time loads the last checkpoint. In existing solutions a crash occurs each time a crash-causing error is signalled.



(c) LetGo continues the execution of the application when a crash-causing error occurs, and the HPC run-time does not load the checkpoint.

Figure 1: Illustration of how LetGo changes the behavior of an HPC application that uses checkpointing by continuing the execution when a crash-causing error occurs. Axes indicate time. The labels used for time intervals: CP - checkpoint; V - application acceptance check/verification; L - LetGo framework, lightning bolt: crash-causing error

rollback in the minimum scale, while algorithm-based fault tolerance (ABFT) techniques such as [14, 56] compute checksums for the intermediate states of the LU factorization problems and enable forward recovery. LetGo aims to provide an application-agnostic and forward recovery solution, in the same vein as GVR [18] however it is more general and efficient (as there is no need to store previous program states, and no need of any data structure or interface support).

LetGo is inspired by two key areas: *failure-oblivious computing*, which focuses on recovering from failures caused by software bugs [49, 50]; and *approximate computing* which makes the assumption that ignoring some errors during application execution, will still lead to producing acceptable results. We discuss below how LetGo relates to these areas and highlight the differences.

Failure oblivious computing: Rinard et al. [49, 50] propose failure oblivious computing, an approach that continues application execution when memory-related errors occur during execution. To this end it relies on a technique called boundless memory block: when there is a out-of-bound write, the written value is stored in a hash table indexed by the its memory location, then for out-of-bound reads it retrieves the value from the hash table if the same memory address is used (or uses a default value if the hash table has not been initialized for that value). This is enabled by compile-time instrumentation and checks for all memory accesses at runtime. Our approach differs from theirs majorly in two ways: (1) LetGo focuses on all types of crashes whereas the technique above focuses on out-of-bound memory accesses, a subset of sources of crashes in HPC systems, and (2) we focus on HPC applications where we believe such techniques are likely to have a high impact.

Recently, Long et al. [38] proposed a run-time repair and containment in the same style at the original failure-oblivious computing work. They expand the solution used to drop assumptions on application structure and impose no instrumentation on program during execution. This technique works for errors including divide-by-zero and null pointer de-referencing. Both Rinard et al. [49] and Long et al. [38] find that that the one of the biggest reasons to the success of their techniques is the common computational pattern that occurs in all of their benchmark applications, that the input of the applications can be divided into units and no interaction between computations on different input units. This is not a the typical computation model for HPC applications.

Approximate computing [27, 42] starts with the observation that, in many scenarios, an approximate result is sufficient for the purpose of the task, hence energy and performance gains may be achieved when relaxing the precision constraints. The philosophy of approximate computing is applied to different system levels including circuit design [26, 57], architectural design [24, 47] and application characterization and kernel design [9]. This concept, along with related body of research such as probabilistic computing [8, 44], offer potential platforms for applications that can tolerate imprecision in the final answer. However, LetGo does not aim to aggressively relax the accuracy of the computation, which is a different philosophy from Approximate computing or Probabilistic computing.

Failure escaping: Carbin et al. [6] design Jolt, a system that detects when an application has entered an infinite loop and allows the program to escape from the loop and continue. Jolt has a similar philosophy as LetGo: adjusting the program state when a program appears to be traped in a failing state. However, Jolt and LetGo target different failure types: Jolt is only designed to help programs escape from a infinite loop (a hang), a relatively infrequent failure scenario as our fault-injection experiments indicate.

3 BACKGROUND

Context: The effectiveness of LetGo is influenced by two factors: (1) the application-level acceptance checks that detect whether the application state is corrupted before delivering results to users, (2) the resilience characteristics of the HPC application making it able to withstand minor numerical perturbations. This section argues that a large class of HPC applications present these characteristics, and offers an example that illustrate how these two factors affect application fault-tolerance.

Factor 1: Application acceptance checks. Since the rate of hardware faults is expected to increase and applications become increasingly complex (and, as a result, the design and implementation process is error-prone), there is an increased awareness for the need of result acceptance tests, to boost the confidence in the results offered by HPC applications. Result acceptance checks are usually written by application developers to ensure that computation results do not violate application-specific properties, such as energy conservation or numeric tolerance for result approximation. These acceptance checks are typically placed at the end of the computation (i.e. the residual check performed in HPL application [46]), but they can be

also placed during application execution to detect earlier possible state corruption such as [43].

Factor 2: Fault masking in HPC applications. A large class of HPC applications are based on iterative processes (For example, stencil computations iteratively compute physical properties at time T+1 based on values at time T; iterative solvers work by improving the accuracy of the solution at each step. For an iterative method that is convergent, numerical errors introduced by a hardware fault can be eliminated during this convergence process (although it may take longer). Prior studies such as [7] show that the algebraic multi-grid solver always masks errors if it is not terminated by a crash.

Terminology. We use standard trminology for the fault-tolerance domain: fault/error/failure [2]. Hardware *faults* are defects in the system that may be caused by cosmic rays or particle strikes. We are concerned with the faults that are not masked by the hardware and are thus visible at the application level. *Errors* are the manifestation of faults visible in the application state. *Failures* are the final outcomes of errors, and include crashes, application hangs, and SDCs.

4 SYSTEM DESIGN

Our goal is to demonstrate the feasibility and evaluate the potential impact of a run-time framework that allows the program to avoid termination and correct its state after a crash-causing error occurs. The four main requirements of LetGo are:

a) *Transparency:* LetGo should be able to transparently track the system behavior, monitor for crash-causing errors, and modify the application state to enable application continuation once a crash-causing error occurs, all without modifying the application's code (R1).
b) *Convenience:* As HPC applications tend to be conservative and sensitive to the computation environment, LetGo should not make any assumption about the application's compilation level or require changes the application's compilation process (R2).
c) *Low overhead:* To be attractive for deployment in production systems, LetGo should incur minimum overheads in terms of performance, energy and memory footprint (R3).
d) *A low rate of newly introduced failures:* LetGo inherently trades the ability to continue application execution for the risk of introducing new failures. For LetGo to be practical, the increase in the rate of undetected incorrect results should be low (R4).

The rest of this section describes LetGo design in detail and, shows how LetGo satisfies the above requirements.

4.1 Overall Design

LetGo is activated when a crash-causing error occurs. LetGo detects the exceptions raised by the OS, intercepts the OS signals, and modifies the default behavior of the application for these signals. Then it diagnoses which states of the program have been corrupted, and modifies the application state to ensure, to the extent possible, application continuation. Figures 2 and 3 show this process.

LetGo contains two components: the *monitor* and the *modifier*.

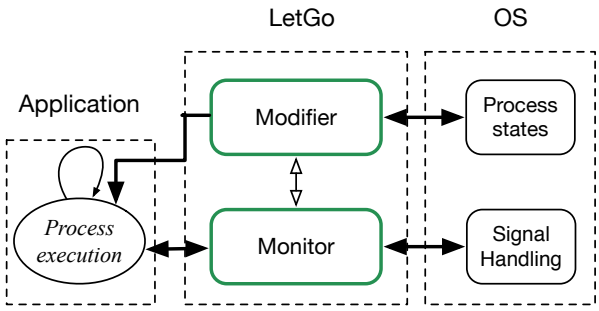


Figure 2: LetGo architecture overview.

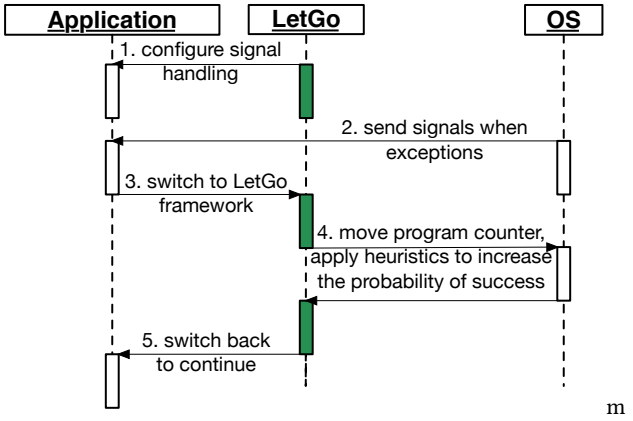


Figure 3: A sequence diagram highlighting LetGo use: the step 1-5 describe the interactions between LetGo, the application, and the operating system: LetGo starts by installing the monitor - i.e., configuring the the signal handling, and launches the application in a debugger (step 1). If the application encounters a signal, LetGo detects it (step 2) and takes the control of the application (step 3). To avoid the failure, LetGo increments the program counter (i.e., instruction pointer) of the application and adjusts the necessary program states (step 4). After the modification, LetGo lets the application continue without any further interference (step 5).

The monitor is attached to the application at startup. It changes the default behavior of the application from termination to pausing when operating system signals such as SIGSEGV and SIGBUS are received. The monitor intercepts these signals and hands the control over to the modifier.

The modifier kicks in when executing the current application instruction would lead to an application termination (crash). The modifier attempts to avoid the crash: it advances the program counter (i.e., instruction pointer) to the next instruction, and inspects and modifies application state (e.g., the stack pointer) to increase the probability of a successful continued execution. The details of the modifier are discussed in Section 4.2. Note that the application might still not be able to finish successfully and it may crash again - if so, LetGo does not intervene and allows the application to terminate.

4.2 Heuristics

We describe the modifications that the LetGo mmodifier makes to the application state (in Step 4 of Figure 3). These modifications have

two goals: first, increase the likelihood that, once the application continues execution, it does not crash again; and, second, reduce the chance that data corruption propagates further.

There are two issues to deal with: first, advancing the program counter may bypass memory loads or stores, and hence the destination register that is supposed to hold the value from the memory load (or the memory location used for store) may contain an incorrect value , which may cause subsequent errors in case this register is later used; and, second, if the fault has corrupted the stack pointer register *sp* (i.e., *rsp* in X86-64) or the base pointer register *bp* (i.e., *rbp* in X86-64), and the application continues due to LetGo, the likelihood of receiving another system exception due to a memory-related violation high because *sp* and *bp* are repeatedly used. To mitigate these challenges, LetGo employs two heuristics (to satisfy R4).

Heuristic I - This heuristic deals with memory load/store instructions. If the program crashes due to the error in a memory-load instruction, LetGo feeds the to-be-written register(s) (which holds the data loaded from the memory) with a "fake" value(s). In practice, 0 is chosen as the value to feed to the register. We choose 0 by default because the memory often contains a lot of 0s as initialization data [12]. For the case where the program stops at a memory-store instruction, the value in that memory location remains the same because the memory-store operation is not successful. In this case, we do nothing - our empirical experience suggests that this is a more practical decision than assigning a random value. In the future, this heuristic can be combined with run-time analysis for more realistic and application-dependent behaviour.

Heuristic II - As discussed above, if a fault affects the values in the stack pointer register or the base pointer register, the corrupted registers may cause consecutive memory access violations. Since LetGo avoids performing run-time tracking, determining the correctness of the values in *sp* and *bp* statically becomes challenging. To overcome this challenge, LetGo implements the following heuristic that include a detection and a correction phase:

(1) *Detection*: for each function, the difference between the values in *sp* and *bp* can be approximately bound in a range via static analysis, hence this range can be calculated with minimum effort and can be used to indicate the corruption in *sp* or *bp* at run-time.
(2) *Correction*:: since *sp* and *bp* usually hold the same value at the beginning of each function, one can be used to correct the error in the other one if necessary.

We explain the intuition behind this heuristic with two observations based on the code in Listing 1. First, *bp* is normally pointed to the top of the stack (line 2), hence *sp* and *bp* usually carry the same value at the beginning of every function call. Second, based on the size of the memory allocated on the stack (line 3), the range of *bp* can be inferred as $sp < bp < sp + 0x290$ (*bp* is always greater than *sp* because the stack grows downwards. Therefore, when the program receives an exception and stops at an instruction that involves stack operation, LetGo runs the following steps: First, it gets the size of the allocated memory by searching for the beginning of the function that the instruction belongs to and then locating the instruction that shows how much memory the function needs on the stack (by analyzing the assembly code). Second, it calculates the valid range based on the size and checks if the *bp* is in it, and, finally, if the range constraint is invalid, LetGo copies the value of the *sp* to the *bp* (or vice versa depending on which one is used in the instruction causing the crash).

Signal	Stop	Pass to program	Description
SIGSEGV	Yes	No	Segfault
SIGBUS	Yes	No	Bus error
SIGABRT	Yes	No	Aborted

Table 1: *gdb* signal handling information redefined by LetGo. 'Stop' means the program will stop upon a signal, and 'Pass to program' means this signal will not be passed to the program

Listing 1: Example of a common sequence of X86 instructions at the beginning of a function

```
1 push \%rbp
2 mov  \%rsp,\%rbp
3 sub  \$0x290, \%rsp
```

For the rest of this paper, we refer to the version of LetGo that applies these heuristics as **LetGo-E**(nhanced) and the version without heuristics as **LetGo-B**(asic). We evaluate the effectiveness of LetGo-B and LetGo-E in Section 6.

4.3 Implementation

We implement the LetGo prototype with three production-level tools that are widely adopted and readily available on HPC systems: gdb, PIN [45] and pexpect [52].

gdb: LetGo relies on *gdb* to control the application's execution. *gdb* provides the interfaces to handle operating system signals and to change the values in the program registers. We describe these two aspects in turn. LetGo uses *gdb* to redefine the behaviour of an application against OS signals as described in Table 1. Since most of application crashes are due to memory-related errors such as segmentation faults or bus errors [21, 25], LetGo currently supports three signals related to memory errors: SIGSEGV, SIGBUS and SIGABRT, and can be easily extended for more signals if needed (e.g., exceptions generated by ECC or chipkill). (Satisfying R1).

Note that the LetGo use of *gdb* does not require any source-code level analysis (or changes to the application). Applications therefore do not need to run in the debug mode, which inhibits code optimization and often results in significant performance degradation (satisfying R2 and R3). Applications can run with LetGo for any optimization/compilation requirement levels they need. We evaluate the generated overhead in Section 5.

b) PIN: It is a tool that supports dynamic instrumentation of programs. *PIN* can insert arbitrary code at arbitrary locations of an executable during its execution. LetGo uses *PIN* to conduct instruction-level analysis, such as obtaining the next PC, parsing an instruction and finding the size of allocated memory on the stack. Since LetGo only needs the static information of a program, there is no need for LetGo to keep track of dynamic program states and only dissembler inside PIN is needed. Therefore, LetGo incurs minimum performance overhead (Satisfying R3). It is possible to use other lightweight tools for parsing instructions instead of PIN.

c) pexpect: expect [37] is a tool that automates interactive applications (e.g. telnet, ftp, etc.) and it is widely for testing. LetGo uses *pexpect*, the Python extension of *expect* to automate all interactions

between LetGo and the application: e.g., configuring signal handlers and updating register values. Since these are relatively rarely executed operations, the overall performance impact is small.

All the interactions between a *gdb* process and the target application are automated via *pexpect*, and confined to a limited number of *gdb* commands such as "print" or "set". When heuristics need to be applied, LetGo relies on *PIN* to analyze the program and feed the result to *gdb*. As a prototype, the current implementation of LetGo is used to support the experimentation, to demonstrate the ability of automation, and to investigate the overheads incurred - for a production version, one can directly and efficiently implement the functionality offered by each of these tools, so the overhead estimates we offer are conservative.

5 EVALUATION METHODOLOGY

This section focuses on evaluating the ability of LetGo to transform crashes into successful application runs To this end, this section first describes the fault model and the fault injection methodology we use, then explains how the various failure outcome categories are impacted by LetGo, and proposes metrics to quantitatively evaluate LetGo effectiveness. Using this information, the next section evaluates LetGo impact on reducing C/R overheads.

5.1 Fault Model

Soft errors are one of the main sources of hardware errors in processors [4], and are the focus of this work. We consider faults occurring in the computational units of processors, such as the ALUs, pipeline latches and register files. Our methodology is agnostic to whether a fault arises in the register file or is propagated to the registers from elsewhere. We do not consider faults in caches or main memory because we assume that they are protected by ECC or chipkill in HPC systems. We use the single-bit-flip model as it is the most common transient fault model in today's systems [53]. We also assume that at most one fault occurs in an application run leading to a crash-causing error, as soft errors are relatively rare compared to typical application execution times.

5.2 Categories of Fault Outcomes

The traditional outcomes of a fault affecting an application can be categorized as crashes, detected by the application acceptance check, hangs, SDCs, and benign outcomes. When applying LetGo, (some of the) crash outcomes are transferred to other categories, thus, to evaluate LetGo we further categorize the outcomes that correspond to a crash in a non-LetGo context in multiple new classes as presented in Figure 4.

At the top level of our taxonomy (Figure 4), a fault either causes a program to crash, or not. In Figure 4 we label these two classes - *Finished* and *Crash*.

(1) A *finished* run can result further in two outcomes: the program contains errors in the output that are detected by the application's acceptance checks (labeled as *Detected*), or the output of the program passes those checks (labeled as *Pass check*). If the output passes the check, it may differ from the golden run, in which case we consider it an *SDC* [1]; or the output matches the golden run, labeled as *Benign*.

(2) A *Crash* run is where LetGo has impact. When LetGo is deployed, it may fail to continue the application and lead to a second crash (labeled as *DoubleCrash*) or make the application to finish successfully labeled *C-Finished*. In this case, the program may have similar outcomes as in the *Finished* case (when no crash occurs) - we label these as *C-Benign* (no observable outcome of the fault), *C-Detected* (incorrect output detected by application acceptance checks), and *C-SDC* (incorrect output not detected by those checks, but different from the golden run). Compared to a situation when LetGo is not used, LetGo is able to transfer some of the crash outcomes to C-Benign, C-SDC, or C-Detected outcome.

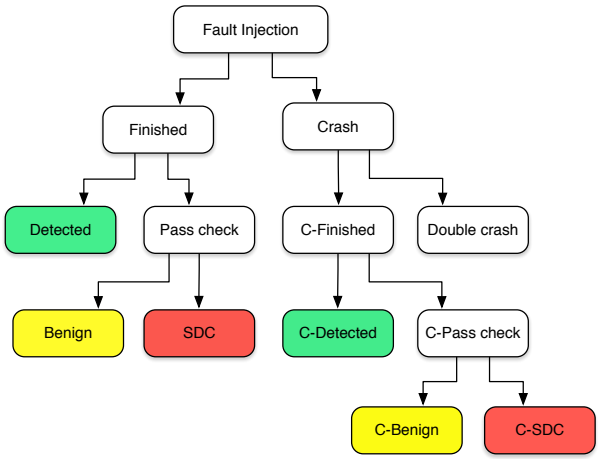


Figure 4: Classification of the fault injection outcomes. LetGo has impact only on the right side of the tree above as it attempts to avoid a crash outcome.

5.3 Metrics of Effectiveness

The effectiveness of LetGo can be estimated by answering questions like "How many crashes can be converted to continued execution?", "what is the likelihood of producing correct results after continuation?", "How often the application check can catch the errors after continuation?" and "How many incorrect results will be produced after continuation?". To this end, we define four metrics to quantitatively evaluate LetGo effectiveness:

Continuability is the likelihood that LetGo is able to convert a crashing program into the program that would finish (regardless of the correctness of the output).

$$Continuability = \frac{C\text{-}Pass\ check + C\text{-}Detected}{Crash} \quad (1)$$

Continued_detected is the likelihood that the application acceptance check catches errors in the application (if any) after continuation.

[1] This is a conservative assumption as we do not know how the results of the application are used. The application output also includes the application data that is compared between such data from the golden run, as defined in Table 2

Application	Domian	# dynamic instructions (billions)	Application data used to check for SDCs	Criteria used in application acceptance check
LULESH	Hydrodynamics	1.0	Mesh	Number of iterations: exactly the same Final origin energy: correct to at least 6 digits Measures of symmetry: smaller than 10^{-8}
CLAMR	Adaptive mesh refinement	2.8	Mesh	Threshold for the mass change per iteration
HPL	Dense linear solver	1.2	Solution vector	Residual check on the solution vector
COMD	Classical molecular dynamics	5.1	Each atom's property	Energy conservation
SNAP	Discrete ordinates transport	1.6	Flux solution	The flux solution output should be symmetric
PENNANT	Unstructured mesh physics	1.7	Mesh	Energy conservation

Table 2: Benchmark description. The last two columns present which data is used for bit-wise comparison to determine SDCs (undetected incorrect results), and, respectively describe the result acceptance check used by each application. All benchmarks are compiled with g++ 4.93 using O3 except for SNAP, which is a FORTRAN program.

$$Continued\_detected = \frac{C\text{-}Detected}{Crash} \quad (2)$$

Continued_correct is the likelihood that the programs result in the correct output after continuation.

$$Continued\_correct = \frac{C\text{-}Benign}{Crash} \quad (3)$$

Continued_SDC indicates the likelihood that application finished but results in SDCs - undetected incorrect results.

$$Continued\_SDC = \frac{C\text{-}SDC}{Crash} \quad (4)$$

Note that the *Continuability* is the sum of *Continued_detected*, *Continued_correct* and *Continued_SDC* metrics. All four values range from 0 to 1.

For an application to benefit from LetGo, it needs to satisfy the following properties: first, there is a low probability that the key program states are not affected by the failure (indicated by high Continuability), and second, there is a high probability that the program states adjusted by LetGo converge to the original path (indicated by the high Continued_correct and low Continued_SDC).

5.4 Fault Injection Methodology

To evaluate LetGo, we implement a software-based fault injection tool based on *gdb* and *PIN* [2]. Our injector does not require the application's source code, nor does it need the application to be compiled using special compilers or compilation flags.

The fault injection experiments for an application consists of two phases: we first perform *a one-time profiling phase* to count the number of dynamic instructions using the PIN tool. As we assume all dynamic instructions have equal likelihood of being affected by a fault, we use the number of total instructions to randomly choose an instruction to inject a fault for each fault injection run. During this phase, we also profile the number of times each static instruction in the program is executed during the profiling phase so at to be able to inject a fault at the appropriate dynamic instruction. For example, if we choose to inject into the 5th dynamic instance of an instruction, we need to skip the first 4 instances when the breakpoint is reached (using the *continue* command of *gdb*).

During the *fault injection phase*, we then use *gdb* to set a breakpoint at the randomly-chosen dynamic instruction and inject a fault at the instruction by flipping a single bit in its destination register (after the instruction completes). This emulates the effect of a fault in the computational units involved in the instruction.

The profiling phase is run once per application and is relatively slow. The injection phase, on the other hand, is executed tens of thousands of times, and is much faster as it does not involve running the application inside a virtual machine as PIN does. *We perform a total of* 20,000 *fault injections per application, one per run, to obtain tight error bounds of 0.1% to 0.2% at the 95% confidence interval.*

5.5 Benchmarks

We use six HPC mini-applications namely LULESH [20], CLAMR [43], HPL [46], SNAP [35], PENNANT [33] and COMD [10] (details in Table 2). Note that these benchmarks meet the assumption that application-level acceptance checks are well defined/implemented. All benchmarks exhibit convergence-based iterative computation patterns except for HPL, which is implemented with a direct method[3] [16]. Therefore, we separate the results of HPL from others and discuss them in Section 8.

Table 2 briefly describes the acceptance checks for each benchmark. For *CLAMR, HPL, PENNANT*, we use the built-in acceptance checks (written by the developers), while for *LULESH, COMD and SNAP*, we wrote the checks ourselves based on their verification specifications: Section 6.2 in [31] for *LULESH*, "Verification correctness" section in [11] for *COMD* and "Verification of Results" section in [34] for *SNAP*.

6 EXPERIMENTAL RESULTS

This section presents experiments that aim to understand whether LetGo is indeed able to continue application execution when a crash-causing error occurs with minimal impact on application correctness and efficiency. The next section evaluates the impact of LetGo in the context of an C/R mechanism.

[2]The tool is publicly available at https://github.com/flyree/pb_interceptor

[3]A direct method computes the exact answers after a finite number of steps (in the absence of roundoff)

Benchmark	Finished			Crash			
	Detected	Pass check		Double crash	C-Detected	C-Pass check	
		Benign	SDC			C-Benign	C-SDC
LULESH	0.90%	22.00%	0.13%	25.00%	2.30%	49.50%	0.17%
CLAMR	0.50%	33.30%	0.50%	25.00%	1.10%	39.60%	0.00%
SNAP	0.02%	43.94%	0.01%	20.77%	0.06%	35.20%	0.00%
COMD	1.00%	55.00%	1.10%	18.32%	0.85%	22.13%	1.60%
PENNANT	1.00%	50.00%	2.00%	19.00%	2.50%	22.70%	2.80%
AVERAGE	**0.68%**	**40.85%**	**0.75%**	**21.62%**	**1.36%**	**34.02%**	**0.91%**

Table 3: Fault injection results for five iterative benchmarks when using LetGo-E. The value for each outcome category is normalized using the total number of fault injection runs for the application. Error bars range from 0.1% to 0.2% at the 95% confidence level.

6.1 Effectiveness of LetGo

We run the fault injection experiments for both LetGo-B (the basic version that uses minimal repair heuristics) and LetGo-E (the version that uses the advanced heuristics described in Section 4). Table 3 shows the fault injection result for the five benchmarks that use iterative, convergence-based solutions when using LetGo-E. We discuss HPL, a direct method, separately in the discussion section.

We note the following: First, the the average crash rate over all applications is 56%, showing that more than half of the time when a fault occurs the application will crash (i.e., in the table this shows as the sum of values in the four columns under the "Crash" category). Second, with LetGo-E, on average, 62% of these crashes can be transformed to continue running the application to termination (only 38% are double crashes). We first discuss the results for LetGo-E, and then compare these results with those for LetGo-B to understand the effectiveness of the heuristics introduced by LetGo-E. We observe the following:

(1) **The ability of LetGo-E to enable continued execution when facing a crash-causing error**: The mean *continuability* for the benchmark set is 62%, which indicates that 62% of the time when the benchmark program receives a crashing signal, LetGo-E resumes the execution and the application completes successfully without crashing again.
(2) **LetGo-E is able to convert more than half of the crashes to produce correct results** (and thus possibly offer a solution to lower checkpoint overheads for a long-running applications).
(3) **Low rate of undetected incorrect results.** The rate of *Continued_SDC* cases for all benchmarks is on average in the same range as the SDC rate of the unmodified application. For *CLAMR* and *SNAP*, we do not observe new SDCs after applying LetGo-E. Overall, LetGo-E maintains the low SDC rate of the original application (yet it doubles it only 1.6% of the cases did the program produce incorrect results after continuing it with LetGo-E, compared with 0.75% when not using LetGo). We further discuss the impact of the increased SDC rates, and techniques to mitigate it, in Section 8.
(4) **Continued_detected of the application-level acceptance checks**. The Continued_detected of LetGo-E across the five benchmarks is 2.4%: for our benchmarks, after LetGo-E continues the execution, the application acceptance checks would detect the errors 2.4% of the time - this is slightly higher than the case without LetGo-E.

Thus, we find that LetGo-E has a high likelihood to convert crashes into either benign or detected states, while only marginally increasing the SDCs produced.

Figure 5 compares LetGo-B and LetGo-E over the four metrics. Figure 5a shows that LetGo-E achieves an improvement in Continuability for CLAMR by 32% and for PENNANT by 5% over LetGo-B , but not much for the other benchmarks (considering the error bars). Overall, LetGo-E achieves 14% on average higher Continuability than LetGo-B. Figure 5b shows that the Continued_detected declines by 1% from LetGo-B to LetGo-E on average and with only 0.8% increase in *CLAMR*. Therefore, the efficacy of the acceptance checks is not much affected by the heuristics employed by LetGo-E. Figure 5c shows that LetGo-E has higher Continued_correct over LetGo-B by 4% on average across all benchmarks. This shows that it allows more crashes to be converted into correct results than LetGo-B. In Figure 5d, we find that Continued_SDC ratio for LetGo-E remains the same as that of LetGo-B on average. In Figure 5d, we can observe that LetGo-E totally eliminate the SDCs for *CLAMR* and *SNAP*, and has almost the marginally different values of the Continued_SDC metric for all benchmarks - the worst case is 2% higher Continued_SDC faults for PENNANT. Thus, the heuristics used by LetGo-E does not add much to the incorrect executions.

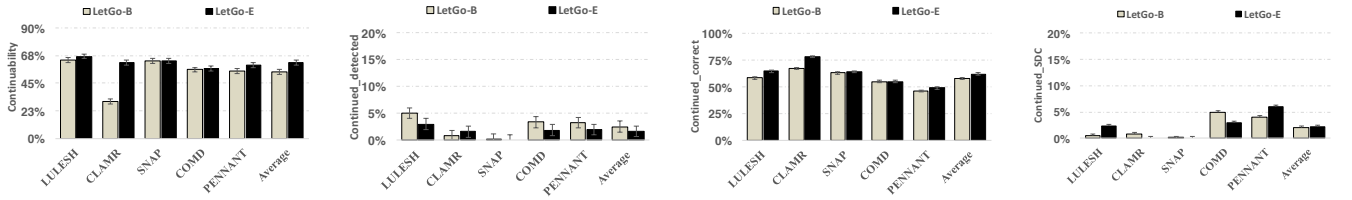
Overall, the heuristics introduced by LetGo-E lead to better continuablility (by about 14%) over LetGo-B for continuing the programs, and producing 5% more correct results than LetGo-B.

6.2 Performance Overhead

To estimate performance overhead, we experimentally measure the performance LetGo for a single application run outside the context of a C/R scheme. In Section 7, we will evaluate the end-to-end impact of LetGo when used in the context of an C/R scheme, that is in the presence of failures and considering C/R overheads.

There are two source of overhead in LetGo: a). Running the program with *gdb*. b). Adjusting program states after a crash happens. We report the time overhead for each part below. Since LetGo-E is a superset of the operations performed by LetGo-B, we report only the LetGo-E overheads to get the worst-case time overhead.

We first measure the execution time of LULESH with LetGo, under three input sizes. We find that for the three different input sizes, the number of dynamic instructions range from 1 billion



(a) Continuability **(b) Continued_detected** **(c) Continued_correct** **(d) Continued_SDC**

Figure 5: Comparison of Continuability, Continued_detected, Continued_correct and Continued_SDC between the LetGo-B and LetGo-E. LetGo-E has a higher likelihood of converting crashes into correctly executions for our benchmarks than LetGo-B but no increase in Continued_SDC cases.

to 180 billion, and LULESH with *gdb* exhibits consistently low overhead (i.e., less than 1% compared to running it without *gdb* for each case). We have observed a similar trend for the rest of the benchmarks as well. As explained in Section 4, this is because LetGo neither changes the applications' compilation levels nor does it set breakpoints on the application.

We also measured the time overhead of adjusting program states after a crash by measuring how much time is spent in LetGo-E for each benchmark (i.e., the time spent in step 4 of the Figure 3). We find that across all of our benchmarks, the wall-clock time spent in LetGo is roughly around 2-5 seconds, and, as expected, it stays constant when we increase the input size. This time is trivial compared to the overall execution time of most HPC programs. Recall that LetGo takes two actions to adjust the program states: 1). finding the next PC, 2) applying the two heuristics if necessary. As explained in Section 4.2 and Section 4.3, both actions only need a disassembler to acquire the static instruction-level information of a program - we use PIN. With a more efficient disassembler, the time overhead can be even further reduced. *Thus, LetGo incurs insignificant performance overheads in most cases, and this overhead does not increase with increase in applications' input sizes.*

7 LETGO IN A C/R CONTEXT

The previous section demonstrated that LetGo is indeed able to often continue application execution with minimal impact on application correctness and efficiency. This section aims to evaluate the end-to-end impact of LetGo in the context of a long-running parallel application using a C/R mechanism. The main challenge in this evaluation is that there are multiple configuration scenarios that need to be considered, and hence direct measurement is prohibitively expensive. To address this issue, we model a typical HPC system using C/R as a state machine and have built a continuous-time event simulation of the system. This simulation framework enables us to compare resource usage efficiency with and without LetGo. We predict the overall performance gains using LetGo, based on the effectiveness of LetGo estimated with fault injections on an application-specific basis in the previous section. We focus on LetGo-E as we found that it achieves higher Continuability and Continued_correct compared to LetGo-B. In the rest of this section, when we say LetGo, we mean LetGo-E.

Model assumptions. We make a number of assumptions that are standard for most of the checkpointing literature [5, 13, 17, 58]. Our models assume that all crashes are due to transient hardware faults, and hence restarting the application from a checkpoint will be sufficient to recover from the crash. In a similar vein, we assume that the checkpointing process itself is not corrupted by a fault. We further assume that the application does not have any other fault-tolerance mechanism than C/R (and LetGo), and that it does not modify its behaviour based on the faults encountered. Finally, when modelling a multi-node platform, we assume the HPC system uses synchronous coordinated checkpointing, which implies that checkpoints are taken at the same time across different nodes via synchronization; and that, when one node crashes, all nodes in the system have to fall back to the last checkpoint and re-execute together.

Parameter description We categorize the model parameters into three classes, summarized in Table 4:

(1) Configured: The time to write a checkpoint (*T_chk*), and the mean time between faults (*MTBFaults*) are configured by the model users based on the characteristics of the platform and the application;
(2) Estimated: The probability of a crash after a fault occurs (*P_crash*), the probability that an application passes an application-defined acceptance check (*P_v*, the probability that an application passes an an application-defined acceptance check after LetGo has been used to repair state (*P_v'*), and LetGo Continuability (*P_letgo*) are obtained from the fault injection experiments on a per application basis.
(3) Derived: The checkpoint interval T is determined to be a value that maximizes efficiency for the current configuration based on Young's formula [58] (when not explicitly mentioned otherwise in the experiment description). The recovery time (*T_r*) is, conservatively, chosen to be equal to the checkpoint overhead *T_c* as we assume the equal write and read speed access to the stable storage (also used in prior work [5]), and neglect the additional coordination overhead. We assume that the time spent for an acceptance check (*T_v*) is proportional to the checkpoint overhead because the size of the data to check is the same. We use: *T_v = 0.01 * T_c*. The overhead for synchronizing multiple nodes (*T_sync*) to take a coordinated checkpoint is (optimistically, as we do not consider system scaling effects) a constant fraction of the per/node checkpointing time (*T_chk*). We use two values for the synchronization overhead as 10% and 50% of the checkpointing overhead.

Parameter	Description	Value
T	Checkpoint interval (useful work)	$\sqrt{2 * T\_chk * MTBF}$†
T_r	Time spent for recovery from a previous checkpoint	T_chk
T_chk	Time spent writing a checkpoint	System-dependent
T_sync	Time spent in synchronization across nodes	50%*T_chk and 10%*T_chk
T_v	Time spent in application acceptance check	1%*T_chk
T_letgo	Time spent in LetGo	5s
P_crash	The probability that a application crashes when a fault occurs	Application-dependent
P_v	The probability that an application passes the verification check	Application-dependent
P_v'	The probability that an application passes the verification check with LetGo	Application-dependent
P_letgo	The Continuability of LetGo	Application-dependent
MTBF	Mean time between failure	System-dependent
MTBF_letgo	Mean time between failure with LetGo	MTBF/(1-62%)
MTBFaults	Mean time between faults	System-dependent

† EI-Sayed et al. [17] show that checkpointing under Young's formula achieves almost identical performance as more sophisticated schemes, based on exhaustive observations on the production systems.

Table 4: The description of parameters of the models

Finally we can derive mean time between failures (*MTBF*) based on the experiments in the previous section. As we observe from the fault injection experiments in the previous section, on average 56% of faults lead to crashes. Thus for simplicity, we use *MTBFaults* = 2**MTBF*. The C/R scheme with LetGo helps the application avoid crashes, which results in a longer MTBF. We refer to this new MTBF of the system after LetGo is applied as MTBF_letgo, and since the Continuability of LetGo is about 62% on average, we set *MTBF_letgo* equal to *MTBF*/(1-62%).

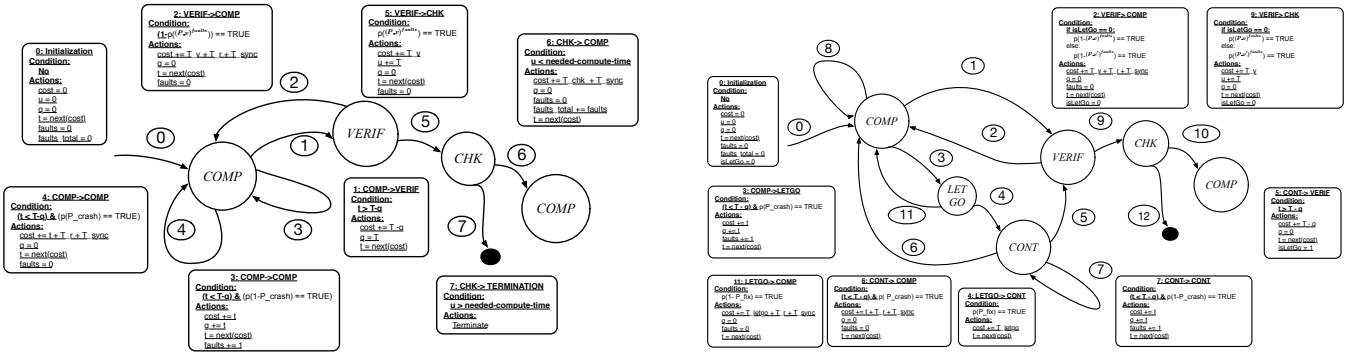
Model description. The state machine modelling a system that does not use LetGo is depicted in Figure 6a and has three states: **COMP**utation, **Ch**ec**k**point, and **VERIF**-ication. In the beginning, the application enters the COMP state for normal computation. A transition is made from the state COMP to VERIF if no crash happens (①), and the acceptance check is applied on the application data/output. If this check passes, a transition is made from the state VERIF to CHK (⑤) and a checkpoint is taken immediately. If the application does not pass the check, it transits from the state VERIF to COMP (②). A transition from the state COMP back to itself occurs when a failure is detected (④), or faults occur when the application is in the COMP state but none of them cause crashes, so that the application stays in the COMP state and the number of faults will be increased (③). When faults are accumulated in the system, the probability that the application passes the verification check is modeled as $(P\_v)^{faults}$, given the assumption that hardware transient faults occur as independent events.

Figure 6b illustrates the model for the C/R scheme when using LetGo. The state machine contains two more states: "LETGO" and "CONT"inue. Due to space limitations, we emphasize here only the transitions related to the new states. When there is a failure (i.e., crash) occurring during the computation (i.e., the application stays in the COMP state), a transition is made to the LETGO state (③). The application moves from the LetGo state to CONT if LetGo continues the execution of the application (④), otherwise, the application transits back to the COMP state (①). While the application stays in the CONT state, the occurring fault can either cause another crash and make the application transit to the COMP state (⑥), or not cause a crash and make the application proceed to the state VERIF. The "isLetGo" flag is set for choosing the different base probabilities (P_v or P_v') that the application passes the verification check (⑤). The base probability is used in the conditions of the state transitions ② and ⑨. The actual probability is then calculated using $(P\_v)^{faults}$ or $(P\_v')^{faults}$ in ⑧ and ⑦.

Evaluation metric. The goal of the simulation is to understand the impact of LetGo on resource usage efficiency in the context of a long running application using a C/R scheme in the presence of faults. We define resource usage efficiency as the ratio between the accumulated useful time and the total time spent (i.e., u/cost). To evaluate the efficiency of both setups (with and without LetGo), we perform simulations for configuration parameters corresponding to different benchmarks and different platforms.

Choice of parameters. We justify the chocies for the checkpointing overhead, and the MTBF. We first discuss the checkpointing overhead: the time spent to write a checkpoint to the persistent storage depends on the characteristics of the hardware. For more advanced hardware, the checkpointing overhead becomes less significant. However, on one side, advanced hardware support such as burst buffers represent additional costs. To the degree these are added to reduce checkpointing overheads, a checkpointing scheme with lower overhead would enable provisioning systems for lower overall cost. On the other side, even in the presence of burst-buffers, checkpointing is still a major bottleneck on deployed systems as our simulations show. We use two criteria for choosing the checkpoint overheads. Here are two data points that justify our choice of parameters to seed our simulations:

- *Back-of-the-envelope calculation*: For each checkpointing overhead value we pick for our simulations we assume that the system-level checkpointing writes some portion of the main memory to the persistent storage. A modern HPC node normally features 32 to 128GB memory. For a



(a) M-S **(b) M-L**

Figure 6: The state machines for the standard C/R scheme (a) and the C/R scheme with LetGo (b). The black circle represents the termination state of the model. We use u/cost to represent the efficiency of the model. *t: time interval till the next fault; cost: accumulated runtime; u: accumulated useful work; q: accumulated useful work within the current checkpoint interval; faults: number of faults that did not lead to crashes since the last checkpoint; faults_total: total number of faults that did not lead to crashes; isLetGo: a flag that indicates that if the P_v' is chosen or not*

burst buffer implemented with SSD, the average I/O bandwidth for write is around 1GiB/s, and the peak value is 6GiB/s [3]. For spinning disks, the I/O bandwidth is usually around 50MiB/s to 500MiB/s. As a result, our choices for the checkpoint overhead of (12s, 120s or 1200s) respectively represent, (i) a well provisioned system using burst buffers, (ii) and averagely provisioned system (e.g., using burst buffers, or compression and spinning disks); and (iii) a naive, under-provisoned system. We note that a similar set of values is also used in prior work [5, 17].

- *Future system requirements*: The Alliance for application Performance at EXtreme scale (APEX) 2020 document [32] requires that the systems delivered in 2020 have a single job mean time to interrupt of more than 24 hours, and for a delta-checkpoint scheme (i.e., the time to checkpoint 80% of aggregate memory of the system to persistent storage), the time for writing the checkpoint to be less than 7.2 minutes (432s). This suggests that our parameters are in the same ballpark as those of the current and future systems.

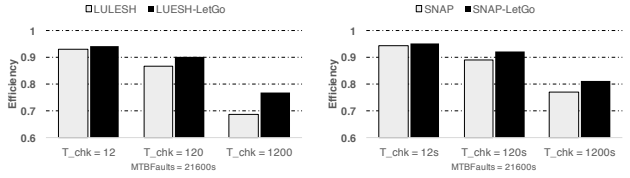
Along similar lines, we derive *MTBFaults* for existing systems from previously reported studies: we start with the system presented by [5] as a baseline. This system contains about 10,000 nodes, and usually experiences around 2 failures per day [40] (*MTBF* of 12 hours). Based on this data, we scale *MTBF* for larger systems, and also consider systems with lower node-level reliability.

Running the Simulation. We assume that the hardware transient faults hitting the system are governed by a Poisson process and we generate a random time sequence for the occurrences of the hardware faults. Then, we seed the models with various sets of parameters and run the simulation over the generated sequence for a long simulation time (10 years), to determine the asymptotic efficiency value for each benchmark.

Experimental Results. We first show the efficiency for the C/R system with and without LetGo under different checkpoint overheads. We take the system that has a MTBFaults of 21600 seconds (i.e., $MTBF = 12hours$) and a synchronization overhead of 10% of the checkpoint interval as an example.

The efficiency improvement enabled by LetGo is between 1% to 11% across our benchmarks (absolute values, the relative values are much higher (nearly 20%)). As the checkpoint overhead scales up (from 12s to 1200s), the efficiency gain increases for all applications, while, at the same time, the absolute efficiency per application decreases. Figure 7 shows the applications that have the highest and lowest efficiency gains respectively, *LULESH* and *SNAP*, when the checkpoint overhead is 1200 seconds. This trend is consistent across all applications, and across different synchronization overheads. Our results thus show that LetGo offers significant efficiency gains.

We then scale the system from 100,000 nodes to 200,000 and 400,000 nodes. Scaling results in lower MTBF for the whole system: 6 and 3 hours. Again, we use two benchmarks namely *CLAMR* and *PENNANT* as examples, shown in Figure 8. As the scale of the system increases, the efficiency of the system both with and without LetGo decreases, as expected. Importantly, the rate of decrease of efficiency is lower for the system with LetGo than without. This trend is consistent with all our benchmarks, suggesting that LetGo offers better efficiency as the system scale increases.



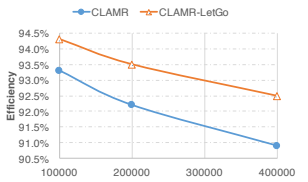
(a) Efficiency of LULESH with and without LetGo **(b) Efficiency of SNAP with and without LetGo**

Figure 7: Efficiency with and without LetGo under different checkpoint overheads for LULESH and SNAP.

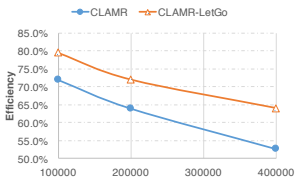
8 DISCUSSION

We address a number of interrelated issues.

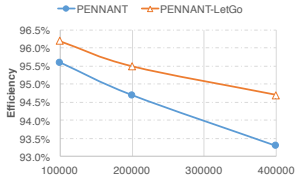
What is LetGo effectiveness wheh used for applications that do not use convergent methods? We have evaluated LetGo on five benchmarks that use convergent methods. We have also



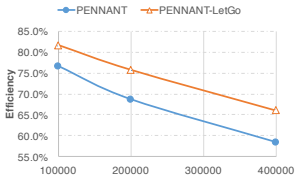
(a) The efficiency trend of CLAMR when T_chk = 12s



(b) The efficiency trend of CLAMR when T_chk = 1200s



(c) The efficiency trend of PENNANT, T_chk = 12s



(d) The efficiency trend of PENNANT, T_chk = 1200s

Figure 8: The trend of the efficiency for the C/R scheme with and without LetGo when the system scales from 100,000 nodes to 200,000 nodes and 400,000 nodes

evaluated LetGo on *HPL*, which is a linear algebra solver that uses a direct method. Our fault injection experiments show that, without the presence of LetGo, 34% of faults lead to crashes, 38% lead to incorrect output detected by the HPL residual check, about 1% lead to SDCs, and 27% lead to correct output. We find that the application-level acceptance checks are much more selective than for the applications in our initial set, and this would make HPL a good candidate for use with LetGo. However, while the crash rate is still high (34%), it is lower than that of the other applications, where it was around 60% - this would potentially reduce the impact of LetGo. When using LetGo with HPL, we obtain around 70% Continuability and 2x increases in SDC rate (from 1% to 3%). To understand the overall performance, we run the simulation of a C/R scheme with the potential results from applying LetGo-E to HPL. Our simulation results show that the efficiency of the standard C/R scheme applied to *HPL* is around 40%, and LetGo-E only marginally improves efficiency. Thus, LetGo by itself is not a good fit for applications like HPL - other error correcting mechanisms (e.g. ABFT) may be needed for such programs.

Determining when/how to use LetGo An operator will decide whether to use LetGo depending on a mix of factors: i) how frequently the system experience hardware faults that crashes the application, ii) what is the likelihood of the application to experience additional SDCs given the use of LetGo, iii) what is the checkpoint overhead for a specific C/R scheme for that application and deployment, iv) what is the acceptable increase in the SDC rate. It is reasonable to assume that the operator has (an approximation for) some of the information above as she needs to configure the checkpoint interval when LetGo is not used. Additionally, the operator needs information to estimate the increase in SDC rate due to LetGo. A large characterization study with applications from multiple categories that extends the preliminary data provided in this paper is necessary to provide these estimates.

Towards large-scale application The current implementation of LetGo focuses on the single-threaded scenario. As an initial effort, we considered this work to be the "proof of concept" for the continuous execution upon failures, and bridges the system level continuation with the correctness of application behaviors. Thus, it is in the early stage of being practical in large-scale production systems. However, the main design and implementation foundations hold no obstacles to be extended for concurrent/multi-threaded applications. Meanwhile, we would like to understand the possibility of integrating LetGo with parallel programming systems such as MPI.

Hardware Fault Models Precise data on the bit-flip rates observed in practice is notoriously hard to obtain. However, we maintain that bit-flips leading to application application crashes are still a frequent root cause for failure. For example, Martino et al. [40] show that in Blue Waters, hardware related issues are the single largest cause of failures (42%) - bugs and configuration errors are only 23%. Of these, 67% of hardware errors are memory errors, and 30% of memory errors manifested as multiple bit flips that cannot be corrected via ECC. While LetGo does not support ECC errors today, there is no fundamental obstacle in adding support for such errors. Moreover, since LetGo allows applications to continue with errors, it may be possible to use it for application-specific (re)configuration of the hardware fault tolerance mechanisms to enable energy savings.

9 CONCLUSION

This paper demonstrates that it is possible to continue HPC application execution rather than terminate it when facing crash-causing errors due to hardware transient faults. We have implemented the above idea in a system called LetGo, which monitors the execution of an application and modifies its default behavior when a termination-causing OS signal is generated. When used in the context of a C/R scheme, LetGo enables sizable resource usage efficiency gains. More specifically, for a set of HPC benchmarks, LetGo offers over 50% chance that the application can continue and produce correct results without performing a roll-back/recovery. We evaluate the impact of LetGo for long-running applications that use C/R, and find that LetGo enables sizable efficiency gains. The efficiency gains increase with both the system scale and checkpointing overheads, thus suggesting that LetGo will likely be even more important for future large-scale HPC applications and systems.

ACKNOWLEDGMENT

The authors would like to thank Panruo Wu, Sijia Gao and Bader Alahmad for their feedback on the project. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC), and the Canada Foundation for Innovation (CFI). This work was supported in part by the U.S. Department of Energy contract AC52-06NA25396. The publication has been assigned the LANL identifier LA-UR-17-20241.

REFERENCES

[1] Guillaume Aupy, Anne Benoit, Thomas Hérault, Yves Robert, Frédéric Vivien, and Dounia Zaidouni. 2013. On the Combination of Silent Error Detection and Checkpointing. In *Proceedings of the 2013 IEEE 19th Pacific Rim International Symposium on Dependable Computing (PRDC '13)*. IEEE Computer Society,

Washington, DC, USA, 11–20. DOI: https://doi.org/10.1109/PRDC.2013.10
[2] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr. 2004. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing* (2004). DOI: https://doi.org/10.1109/TDSC.2004.2
[3] Wahid Bhimji, Debbie Bard, Melissa Romanus, David Paul, Andrey Ovsyannikov, Brian Friesen, Matt Bryson, Joaquin Correa, Glenn K Lockwood, Vakho Tsulaia, and others. 2016. Accelerating science with the nersc burst buffer early user program. *Proceedings of Cray Users Group.[Online]. Available: https://cug.org/proceedings/cug2016 proceedings/includes/files/pap162. pdf* (2016).
[4] Shekhar Borkar. 2005. Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. *IEEE Micro* 25, 6 (Nov. 2005), 10–16.
[5] George et al. Bosilca. 2013. Unified Model for Assessing Checkpointing Protocols at Extreme-scale. *Concurr. Comput. : Pract. Exper.* (2013), 2772–2791.
[6] Michael Carbin, Sasa Misailovic, Michael Kling, and Martin C. Rinard. 2011. Detecting and Escaping Infinite Loops with Jolt. In *Proceedings of the 25th European Conference on Object-oriented Programming (ECOOP'11)*.
[7] Marc Casas, Bronis R. de Supinski, Greg Bronevetsky, and Martin Schulz. Fault Resilience of the Algebraic Multi-grid Solver. In *Proceedings of the 26th ACM International Conference on Supercomputing (ICS '12)*. 91–100.
[8] Suresh Cheemalavagu, Pinar Korkmaz, Krishna V. Palem, Bilge E. S. Akgul, and Lakshmi N. Chakrapani. 2005. A probabilistic CMOS switch and its realization by exploiting noise. In *the Proceedings of the IFIP international*.
[9] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan. Analysis and characterization of inherent application resilience for approximate computing. In *Design Automation Conference (DAC), 2013*.
[10] P. Cicotti, S. M. Mniszewski, and L. Carrington. An Evaluation of Threaded Models for a Classical MD Proxy Application. In *Hardware-Software Co-Design for High Performance Computing (Co-HPC), 2014*.
[11] P. Cicotti, S. M. Mniszewski, and L. Carrington. 2013. CoMD: A Classical Molecular Dynamics Mini-app. (2013). http://exmatex.github.io/CoMD/doxygen-mpi/index.html
[12] J. J. Cook and C. Zilles. A characterization of instruction-level error derating and its implications for error detection. In *2008 DSN*. DOI: https://doi.org/10.1109/DSN.2008.4630119
[13] J. T. Daly. 2006. A Higher Order Estimate of the Optimum Checkpoint Interval for Restart Dumps. *Future Gener. Comput. Syst.* 22, 3 (Feb. 2006), 303–312. DOI: https://doi.org/10.1016/j.future.2004.11.016
[14] Teresa Davies and Zizhong Chen. 2013. Correcting Soft Errors Online in LU Factorization. In *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing (HPDC '13)*. ACM, New York, NY, USA, 167–178. DOI: https://doi.org/10.1145/2462902.2462920
[15] N DeBardeleben, J Laros, JT Daly, SL Scott, C Engelmann, and B Harrod. 2009. High-end computing resilience: Analysis of issues facing the HEC community and path-forward for research and development. *Whitepaper, Dec* (2009).
[16] James W. Demmel. 1997. *Applied Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.
[17] Nosayba El-Sayed and Bianca Schroeder. 2014. Checkpoint/restart in practice: When âĂŸsimple is betterâĂŹ. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 84–92.
[18] A. Chien et al. 2015. Versioned Distributed Arrays for Resilience in Scientific Applications: Global View Resilience. *Procedia Computer Science* 51 (2015), 29 – 38. DOI: https://doi.org/10.1016/j.procs.2015.05.187
[19] G. Bosilca et al. MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes. In *Supercomputing, ACM/IEEE 2002 Conference*.
[20] Ian Karlin et al. 2012. *LULESH Programming Model and Performance Ports Overview*. Technical Report LLNL-TR-608824. 1–17 pages.
[21] Bo Fang, Qining Lu, Karthik Pattabiraman, Matei Ripeanu, and Sudhanva Gurumurthi. 2016. ePVF: An Enhanced Program Vulnerability Factor Methodology for Cross-layer Resilience Analysis. In *DSN*.
[22] Shuguang Feng, Shantanu Gupta, Amin Ansari, and Scott Mahlke. 2010. Shoestring: Probabilistic Soft Error Reliability on the Cheap. *SIGPLAN Not.* 45, 3 (March 2010), 385–396. DOI: https://doi.org/10.1145/1735971.1736063
[23] Marc Gamell, Keita Teranishi, Michael A. Heroux, Jackson Mayo, Hemanth Kolla, Jacqueline Chen, and Manish Parashar. 2015. Local Recovery and Failure Masking for Stencil-based Applications at Extreme Scales. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. ACM, New York, NY, USA, Article 70, 12 pages. DOI: https://doi.org/10.1145/2807591.2807672
[24] B. Grigorian and G. Reinman. Accelerating divergent applications on SIMD architectures using neural networks. In *2014 IEEE 32nd International Conference on Computer Design (ICCD)*.
[25] Weining Gu, Z. Kalbarczyk, and R. K. Iyer. Error sensitivity of the Linux kernel executing on PowerPC G4 and Pentium 4 processors. In *Dependable Systems and Networks, 2004 International Conference on*.
[26] V. Gupta, D. Mohapatra, A. Raghunathan, and K. Roy. 2013. Low-Power Digital Signal Processing Using Approximate Adders. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32, 1 (Jan 2013), 124–137.
[27] J. Han and M. Orshansky. Approximate computing: An emerging paradigm for energy-efficient design. In *2013 18th IEEE European Test Symposium (ETS)*.
[28] Siva Kumar Sastry Hari, Sarita V. Adve, Helia Naeimi, and Pradeep Ramachandran. 2012. Relyzer: Exploiting Application-level Fault Equivalence to Analyze Application Resiliency to Transient Faults. *SIGPLAN Not.* 47, 4 (March 2012), 123–134. DOI: https://doi.org/10.1145/2248487.2150990
[29] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms* (second ed.). Society for Industrial and Applied Mathematics, Philadelphia, PA, USA. xxx+680 pages.
[30] Chung hsing Hsu and Wu chun Feng. 2005. A Power-Aware Run-Time System for High-Performance Computing. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*. 1–1. DOI: https://doi.org/10.1109/SC.2005.3
[31] I.Karlin. 2012. LULESH Programming Model and Performance Ports Overview. (2012). https://codesign.llnl.gov/pdfs/lulesh_Ports.pdf
[32] Los Alamos National Laboratory. 2016. APEX 2020. (2016). http://www.lanl.gov/projects/apex/_assets/docs/2.4-RFP-Technical-Requirements-Document.doc
[33] Los Alamos National Laboratory. 2016. The PENNANT Mini-App v0.9. (2016). https://github.com/losalamos/PENNANT
[34] Los Alamos National Laboratory. 2016. SNAP - SN Application Proxy Summary. (2016). https://asc.llnl.gov/CORAL-benchmarks/Summaries/SNAP_Summary_v1.3.pdf
[35] Los Alamos National Laboratory. 2016. SNAP: SN (Discrete Ordinates) Application Proxy v107. (2016). https://github.com/losalamos/SNAP
[36] G. Li, Q. Lu, and K. Pattabiraman. 2015. Fine-Grained Characterization of Faults Causing Long Latency Crashes in Programs. In *DSN*. 450–461. DOI: https://doi.org/10.1109/DSN.2015.36
[37] Don Libes. 1990. expect: Curing Those Uncontrollable Fits of Interaction. In *PROCEEDINGS OF THE SUMMER 1990 USENIX CONFERENCE*. 183–192.
[38] Fan Long, Stelios Sidiroglou-Douskos, and Martin Rinard. 2014. Automatic Runtime Error Repair and Containment via Recovery Shepherding. *SIGPLAN Not.* (2014).
[39] Qining Lu, Karthik Pattabiraman, Meeta S. Gupta, and Jude A. Rivers. 2014. SDCTune: A Model for Predicting the SDC Proneness of an Application for Configurable Protection. In *Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES '14)*. ACM, New York, NY, USA, Article 23, 10 pages. DOI: https://doi.org/10.1145/2656106.2656127
[40] C. D. Martino, Z. Kalbarczyk, R. K. Iyer, F. Baccanico, J. Fullop, and W. Kramer. 2014. Lessons Learned from the Analysis of System Failures at Petascale: The Case of Blue Waters. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 610–621. DOI: https://doi.org/10.1109/DSN.2014.62
[41] Sarah E. Michalak, William N. Rust, John T. Daly, Andrew J. DuBois, and David H. DuBois. 2014. Correctness Field Testing of Production and Decommissioned High Performance Computing Platforms at Los Alamos National Laboratory *(SC '14)*. Piscataway, NJ, USA, 609–619.
[42] Sparsh Mittal. 2016. A Survey of Techniques for Approximate Computing. *ACM Comput. Surv.* (2016). DOI: https://doi.org/10.1145/2893356
[43] D. Nicholaeff, N. Davis, D. Trujillo, and R. W. Robey. 2012. Cell-Based Adaptive Mesh Refinement Implemented with General Purpose Graphics Processing Units. (2012).
[44] K. Palem and A. Lingamneni. What to do about the end of Moore's law, probably!. In *Design Automation Conference (DAC), 2012*. DOI: https://doi.org/10.1145/2228360.2228525
[45] H. et al. Patil. Pinpointing Representative Portions of Large Intel Itanium Programs with Dynamic Instrumentation. In *MICRO-37*.
[46] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. 2008. HPL - a portable implementation of the high-performance linpack benchmark for distributed-memory computers. (2008). http://www.netlib.org/benchmark/hpl
[47] A. Rahimi, L. Benini, and R. K. Gupta. 2013. Spatial Memorization: Concurrent Instruction Reuse to Correct Timing Errors in SIMD Architectures. *IEEE Transactions on Circuits and Systems II: Express Briefs* (Dec 2013).
[48] L. Rashid, K. Pattabiraman, and S. Gopalakrishnan. 2015. Characterizing the Impact of Intermittent Hardware Faults on Programs. *IEEE Transactions on Reliability* 64, 1 (March 2015), 297–310. DOI: https://doi.org/10.1109/TR.2014.2363152
[49] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M Roy, and Tudor Leu. 2004. A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors). In *Computer Security Applications Conference, 2004. 20th Annual*. IEEE, 82–90.
[50] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee, Jr. Enhancing Server Availability and Security Through Failure-oblivious Computing *(OSDI'04)*. USENIX Association. http://dl.acm.org/citation.cfm?id=1251254.1251275

[51] Marc et al. Snir. 2014. Addressing failures in exascale computing. *International Journal of High Performance Computing Applications* (2014).

[52] Noah Spurrier and contributors. 2013. Pexpect is a Pure Python Expect-like module. (2013). https://pexpect.readthedocs.io/en/stable/index.html

[53] V. Sridharan, N. DeBardeleben, and K. Ferreira S. Blanchard, J. Stearley, J. Shalf, and S. Gurumurthi. 2015. Memory Errors in Modern Systems: The Good, The Bad, and the Ugly. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*.

[54] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott. 2010. Hybrid Checkpointing for MPI Jobs in HPC Environments. In *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*. 524–533. DOI: https://doi.org/10.1109/ICPADS.2010.48

[55] L. Wang, Z. Kalbarczyk, R. K. Iyer, and A. Iyengar. 2015. VM-ĬijCheckpoint: Design, Modeling, and Assessment of Lightweight In-Memory VM Checkpointing. *IEEE Transactions on Dependable and Secure Computing* (March 2015).

[56] Panruo Wu, Qiang Guan, Nathan DeBardeleben, Sean Blanchard, Dingwen Tao, Xin Liang, Jieyang Chen, and Zizhong Chen. 2016. Towards Practical Algorithm Based Fault Tolerance in Dense Linear Algebra. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC '16)*. ACM, New York, NY, USA, 31–42. DOI: https://doi.org/10.1145/2907294.2907315

[57] Z. Yang, A. Jain, J. Liang, J. Han, and F. Lombardi. 2013. Approximate XOR/XNOR-based adders for inexact computing. In *Nanotechnology (IEEE-NANO)*. DOI: https://doi.org/10.1109/NANO.2013.6720793

[58] John W. Young. 1974. A First Order Approximation to the Optimum Checkpoint Interval. *Commun. ACM* 17, 9 (Sept. 1974), 530–531. DOI: https://doi.org/10.1145/361147.361115