CARE: Compiler-Assisted Recovery from Soft Failures

Chao Chen
Georgia Institute of Technology
chao.chen@gatech.edu

Greg Eisenhauer
Georgia Institute of Technology
eisen@cc.gatech.edu

Santosh Pande
Georgia Institute of Technology
santosh.pande@cc.gatech.edu

Qiang Guan
Kent State University
qguan@cs.kent.edu

ABSTRACT

As processors continue to boost the system performance with higher circuit density, shrinking process technology and near-threshold voltage (NTV) operations, they are projected to be more vulnerable to transient faults, which have become one of the major concerns for future extreme-scale HPC systems. Despite being relatively infrequent, crashes due to transient faults are incredibly disruptive, particularly for massively parallel jobs on supercomputers where they potentially kill the entire job, requiring an expensive rerun or restart from a checkpoint.

In this paper, we present **CARE**, a light-weight compiler-assisted technique to repair the (crashed) process on-the-fly when a crash-causing error is detected, allowing applications to continue their executions instead of being simply terminated and restarted. Specifically, **CARE** seeks to repair failures that would result in application crashes due to invalid memory references (segmentation violation). During the compilation of applications, **CARE** constructs a recovery kernel for each crash-prone instruction, and upon an occurrence of an error, **CARE** attempts to repair corrupted state of the process by executing the constructed recovery kernel to recompute the memory reference on-the-fly. We evaluated **CARE** with four scientific workloads. During their normal execution, **CARE** incurs almost **zero** runtime overhead and a fixed **27MB** memory overheads. Meanwhile, **CARE** can recover on an average 83.54% of crash-causing errors within dozens of milliseconds. We also evaluated **CARE** with parallel jobs running on 3072 cores and showed that **CARE** can successfully mask the impact of crash-causing errors by providing almost uninterrupted execution. Finally, We present our preliminary evaluation result for BLAS, which shows that **CARE** is capable of recovering failures in libraries with a very high coverage rate of 83% and negligible overheads. With such an effective recovery mechanism, **CARE** could tremendously mitigate the overheads and resource requirements of the resilience subsystem in future extreme-scale systems.

SC '19, November 17–22, 2019, Denver, CO, USA

ACM ISBN 978-1-4503-6229-0/19/11…$15.00
https://doi.org/10.1145/3295500.3356194

CCS CONCEPTS

• **Computer systems organization** → **Reliability**; • **Software and its engineering** → **Software fault tolerance**.

KEYWORDS

HPC, Resiliency, Reliability, Availability, Transient Fault, Soft Error, SDC, Online Failure Recovery, Online Crash Recovery

ACM Reference Format:
Chao Chen, Greg Eisenhauer, Santosh Pande, and Qiang Guan. 2019. **CARE**: **C**ompiler-**A**ssisted **Re**covery from Soft Failures. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '19), November 17–22, 2019, Denver, CO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3295500.3356194

1 INTRODUCTION

Reliability is a fundamental feature expected from extreme-scale high performance computing (HPC) systems. However, as new processor architectures emerge to boost system performance and energy efficiency with smaller transistor size, higher circuit density, and near-threshold voltage (NTV) operations, the occurrence of transient faults has become one of the major concerns for future extreme-scale HPC systems [1, 4, 13, 21, 31, 35]. In particular, the above technology trends have an undesirable side-effect on the reliability of individual hardware components making them more vulnerable to external noise, e.g., heat fluxes and high energy particle strikes, which could cause temporary bit-flips in circuits and lead to transient faults. Unfortunately, these faults are difficult to mask in a cost-efficient manner [20, 34], and many of them will be exposed to applications. For instance, Oliveira et al. [31] project that a hypothetical exa-scale machine with 190, 000 cutting-edge Xeon Phi processors would experience daily transient errors even though their memory areas are equipped with ECC (Error-Correcting Code) protection. Therefore, efficient application-level resilience techniques are required for future scientific applications [11, 20, 22, 29].

Studies in [3, 9, 16, 23] show that transient faults would either result in incorrect application outputs (Silent Data Corruptions or SDCs) or crashes (referred as **soft failures** in the paper). While there is significant amount of prior work on detecting and correcting SDCs [6–8, 12], less research effort has gone into handling soft failures, with the assumption that standard Checkpoint/Restart (C/R) methods can provide adequate recovery. The C/R technique periodically writes applications' intermediate state (checkpoint) into a stable storage, and loads the latest checkpoint to restart the computation upon a failure. Relying upon this technique for error recovery from soft failures is quite effective, but it is also very

```
1 // memory access statement
2 phitmp[(mzeta + 1) * k] = mzeta * k;
3
4 // the recovery kernel
5 uint64_t recovery_kernel(int *phitmp, int mzeta, ↩
      int k) {
6     return (uint64_t)(phitmp + ( mzeta + 1 ) * k);
7 }
```

Figure 1: A sample recovery kernel.

costly in terms of lost opportunities (batch job slots), lost computation (everything since the last checkpoint) and I/O overheads (repeatedly writing checkpoint files). These costs are particularly significant for massively parallel jobs [13, 15, 30]. Reliance on C/R means that a job which suffers a transient fault will be killed and must be resubmitted, having to wait in the job queue before it can continue execution. Once running, the job must first load the saved state from a checkpoint, which would involve slow I/O operations and then redo calculation that was lost before it can get to the point where the failure occurred. However, since transient faults only impact the hardware temporarily, i.e., for a few cycles, more advanced techniques might mask the fault at the application level, allowing the application to continue the normal operations when their corrupted state is repaired. In this paper, we argue that, while the C/R is still necessary in many circumstances (e.g., power failures, job termination, etc.), soft failures can often be handled with a lightweight resiliency mechanism, which could help to mitigate the overall overheads of the resilience mechanism in HPC systems.

Specifically, we propose **CARE**, a lightweight and compiler-assisted technique to recover processes of scientific applications from soft failures. **CARE** is designed for recovering errors from computing logic units, assuming that memory areas are protected with ECC. Upon a failure, **CARE** will attempt to diagnose and repair the corrupted state for the failing process on-the-fly through replaying related computations, such that applications can continue their executions instead of being simply terminated. **CARE** is inspired by two insights we observed from scientific applications and our study about the manifestation of soft failures (See Section 2):

(1) The majority of soft failures would manifest via hardware traps. Specially, as much as 98.95% of soft failures evidence themselves by causing a *SIGSEGV* because of invalid memory access. This is because many scientific applications contain features like stencil codes, or otherwise involve complex address computations to access neighbor values.
(2) Most soft failures would manifest within a few dynamic instructions. Hence, the original raw data used for array location computations remains uncontaminated, and can be used to recompute the corrupted state.

Based on the above insights and the predominance of soft failures manifesting as invalid memory accesses, **CARE**'s approach is to build a set of *recovery kernels* (one per memory access instruction) that can recompute appropriate addresses for failed dereferences. **CARE** consists of two components: a front-end, **Armor**, for constructing aforementioned *recovery kernels*, and a runtime system, **Safeguard**, for diagnosing the failure and repairing the corrupted architecture state when a soft failure happens. **Armor** is a LLVM pass, and works on LLVM IR representation. For each memory access instruction, **Armor** will construct a *recovery kernel* by strategically extracting instructions related to its address computation. A recovery kernel is simply a standard function which mirrors the address calculation operations of a portion of the application. Upon detection of a memory fault, the runtime system **Safeguard** will execute the recovery kernel with the uncontaminated inputs to recompute the memory references. An example of *recovery kernel* is demonstrated in Figure 1. This kernel computes the address for $phitmp[(mzeta + 1) * k]$, taking *phitmp*, *mzeta* and k as parameters. **CARE** will fetch their unmodified values from the process to execute the kernel thereby leading to the correct address. In short, **CARE** relies on the availability of such values which are typically found in persistent locations such as constant pointers or memory or register values of instructions unmodified by the fault. To minimize memory overheads, recovery kernels for an application are compiled into a stand-alone shared library, which will be loaded dynamically by **Safeguard** when a crash-causing error is detected. **Safeguard** provides recovery services by installing a signal handler for *SIGSEGV*. Upon a failure, **Safeguard** will be invoked to diagnose which instruction caused the invalid memory access, and will disassemble the instruction to determine which operand is referring to a memory address. Based on the address of the instruction, it will then search, load and execute the related recovery kernel to recompute the accessed memory address for the instruction, and update the related operand. **Safeguard** is designed to be as transparent as possible to applications and requires no instrumentation or modification to applications' source code. It is implemented as a shared library that can be automatically loaded through setting the *LD_PRELOAD* environment variable. Because **Safeguard** is not activated unless a crash-causing fault occurs, the small load-time overhead of installing a signal handler and the tiny memory overhead for storing the signal handler are its only impact on an application's execution if a fault does not occur.

This paper makes the following contributions:

- We propose **CARE**, a new failure recovery strategy for scientific applications to survive soft failures. **CARE** exploits hardware detection of memory access violations to repair crashed architecture states on-the-fly by replaying computations that are extracted from applications. **CARE** is lightweight. Except requiring some offline code analysis effort for building recovery kernels, **CARE** incurs almost **zero** runtime overhead and fixed 27MB memory overheads during the normal execution of applications.
- To motivate the design of **CARE**, we studied the manifestation of soft failures in modern scientific applications through empirical instruction-level fault injection experiments. We classified the soft failures based on hardware trap symptoms, and examined their manifestation latency measured in terms of number of dynamic instructions. The results of this empirical study lead to the design of **CARE**.
- We designed and implemented **CARE** based on the LLVM framework and the Linux system. While more engineering work is needed to support -O2/-O3 optimizations, our prototype of **CARE** is a solid step towards a lightweight resilience mechanism for soft failures.

Table 1: Scientific workloads from different scientific domains and implementing different algorithms

Workload	Language	Description
HPCCG	C++	A simple conjugate gradient benchmark code for a 3D chimney domain on an arbitrary number of processors.
CoMD	C	A reference implementation of typical classical molecular dynamics algorithms and workloads as used in materials science.
miniMD	C++	A simple, parallel molecular dynamics (MD) code. It performs parallel molecular dynamics simulation of a Lennard-Jones or a EAM system
miniFE	C++	a Finite Element mini-application which implements a couple of kernels representative of implicit finite-element applications. It assembles a sparse linear-system from the steady-state conduction equation on a brick-shaped problem domain of linear 8-node hex elements. It then solves the linear-system using a simple un-preconditioned conjugate-gradient algorithm
GTC-P	C	A 2D domain decomposition version of the GTC global gyrokinetic PIC code for studying micro-turbulent core transport. It solves the global, nonlinear gyrokinetic equation using the particle-in-cell method.

- We evaluated **CARE** with 4 scientific workloads and with up to 3072 cores. The results show that, on average, **CARE** can recover about 84% of soft failures for the evaluated workloads within dozens of milliseconds, allowing parallel applications to finish their jobs with almost no delays even when crash-causing errors happen during their execution. We also present preliminary evaluation results for *BLAS*, showing that **CARE** can support for failure recoveries in libraries with a high coverage rate and negligible performance hit.

The rest of paper is organized as follows: Section 2 introduces the motivation for **CARE** and explains why it is important for many scientific applications; Section 3 and Section 4 present the design and prototype of **CARE**. Next, evaluation results are presented in Section 5, and the related state-of-the-art studies are discussed in Section 6. Finally, we present our conclusion in Section 7.

2 THE RATIONALE BEHIND CARE

In this section, we will first study how are soft failures typically manifested from transient faults, and present the insights we observed. We then discuss a common feature of scientific applications that motivated the design of **CARE**.

2.1 The Manifestation of Soft Failures

With increasing concerns about transient faults from HPC communities, a solid understanding about the manifestation and propagation of transient faults is key to building an efficient resiliency mechanism. Several recent papers [3, 9, 23] have studied the impact of transient faults on scientific applications leveraging empirical fault injection experiments. While all of these studies point out a need for an efficient application level resilience mechanism against soft failures for scientific applications running on future extreme-scale systems and some, such as [2], examine the propagation of SDCs, none provided quite the insights necessary for devising efficient mechanisms for fault recovery. In their studies, they treat applications as black-boxes. In particular, they do not provide adequate information about how soft failures manifest and propagate inside applications, which is critical for building application level resiliency mechanisms. In this section, however, we focus on exploring how transient faults manifest, propagate and lead to soft failures (crashes). We are specially interested in: 1) determining the major causes/symptoms of soft failures; and 2) the latency of their manifestation in terms of number of instructions executed from the injection point to the crash point. We built a new instruction-level fault injection tool which allows us to track the propagation of faults from instruction to instruction. Our method first injects faults into target operands of randomly selected dynamic instructions. The faults are then allowed to propagate while a trace is captured and analysed. We performed empirical fault injection experiments on five representative scientific workloads (described in Table 1), and analyzed the injections that led to soft failures to find common patterns that can be exploited by a recovery mechanism. These workloads are from different scientific domains e.g., plasma physics, molecular dynamics, etc., and implementing different algorithms, such as Lennard-Jones potential, embedded atom model, and conjugate gradient. For each workload, we performed 10 000 injections based on the single-bit-flip fault model. In the rest of the section, we will detail the methodology of experiments, and the insights we gleaned from this study.

2.1.1 Methodology. We build the fault injection tool with GDB and Python. The tool at runtime attaches itself to the target process randomly, and then injects a fault to the "destination" operand of the instruction at the attachment point. A "destination" operand is one of architecture states, e.g. a register, or a memory cell, that is updated by the instruction. We simulate transient faults from the CPU logic by randomly flipping a bit of the value in the "destination" operand. [1] As done in previous studies [3, 16], we chose a single-bit-flip fault model since it is a conservative way to estimate the causes and the latency for soft failures, considering that multi-bit-flip faults are more likely to incur soft failures with lower latency than single-bit-flip faults [9]. For careful readers who are interested in a multi-bit-flip model, please refer to Appendix Double-bit-flip Model. We utilized capstone[33], an instruction disassembly framework, to disassemble the instruction and get its semantic information for identifying destination operands. The fault is injected at the point right after the instruction is executed, then execution is continued, tracking fault propagation by recording its execution path. For each run of an application, only one injection is performed. The trace of instructions that propagate the fault is then analyzed.

[1] Where the destination operand is implicit, e.g. X86 *idiv %ecx* which divides the value in $\%edx:\%ead$ by $\%ecx$ and store results in $\%eax$ and reminder in $\%edx$, one of the implied destinations, e,g, $\%eax$, is selected.

Table 2: The overall outcomes of fault injections

Workloads	Benign	Soft Failure	SDC	Hang
HPCCG	3118	3409	3472	0
CoMD	6433	2439	1120	8
miniFE	5073	3518	1376	9
miniMD	951	4065	4984	0
GTC-P	6875	1644	1479	2

Table 3: Breakdown of soft failures based on symptoms

	SIGSEGV	SIGBUS	SIGABRT	Other
HPCCG	3322	32	22	33
CoMD	2195	57	41	146
miniFE	3447	51	6	35
miniMD	4028	6	25	6
GTC-P	1196	49	375	24

Table 4: Latency distribution for soft failures

	Latency (Instructions)			
	≤ 10	$11 \sim 50$	$51 \sim 400$	> 400
HPCCG	99.09%	0.482%	0.602%	0.301%
CoMD	64.15%	23.57%	7.43%	4.85%
miniFE	48.03%	37.15%	12.4%	2.407%
miniMD	53.65%	22.09%	0.03%	24.23%
GTC-P	52.68%	28.76%	9.7%	8.86%

Table 5: The percentage of memory access instructions involving multiple computations in their address calculations, and average number of involved operations

	HPCCG	CoMD	miniFE	miniMD	GTC-P
No. Insts	91.49%	94.05%	94.08%	89.22%	86.85%
Avg. No. ops	4.62	5.6	3.04	2.96	3.60

2.1.2 Results and Insights. We categorized the general outcomes of injections into 4 groups: Benign, Soft Failure, SDC, and Hang. A transient fault is benign (or in short vanishes without causing any change in execution) if it doesn't have impact on the application. In such cases, the faulty value could either refer to an incorrect but valid memory location containing the same value to the original memory location, or its effect is masked by a program operation (e.g., min/max operator that masks injections max/min operand, or bit-wise logical operation that suppresses most or least significant bits). Otherwise, it will either kill a process (Soft Failure), lead to incorrect outputs (SDC), or result in a hang state where there is no progress on execution. As presented in Table 2, even though majority of faults are benign, around 30.15% of them manifest as soft failures, and 24.86% of them lead to SDCs. While faults happening in FPU are more likely to cause SDCs, the faults manifested in ALU instructions are more likely to lead to soft failures. There has been a significant amount of work on detecting SDCs but soft failures that cause application crashes have received less attention. Once an application crashes, it needs to be restarted incurring costly recovery operations using check-pointed values.

```
1 for (i = ipsi_in1; i < ipsi_out1+1; i++){
2  for (k = 0; k < mzeta+1; k++) {
3   phitmp[(mzeta + 1) * (igrid[i] - igrid_in) + k] =
4     phitmp[(mzeta + 1) * (igrid[i] + mtheta[i] - ↩
          igrid_in) + k];
5  }
6 }
```

Figure 2: Stencil Code Structure

Table 3 breakdowns the soft failures based on symptoms. It shows that, most (72.75% $\sim$ 98.95%, 91.45% on average) of soft failures manifest as *SIGSEGV*, typically because they corrupt address calculations and lead applications to access invalid memory locations. In addition, Table 4 presents the latency distribution for single-bit-flip model. As it shows, the vast majority of soft failures (> 83%) were manifest within 50 or less dynamic instructions, with more than half of them manifesting within 10 dynamic instructions. We believe such low-latency manifestation implies that the original values (stored in registers or memory) which were involved in the address computation were likely to be intact during this latency window, and that it might be possible to recover the calculation and essentially mask the fault by creating mechanisms to access these original values to recompute the effective address (which is destroyed due to the fault). Based on these two insights, **CARE** is designed specially to protect memory access instructions. During the compilation of applications, it will build a recovery kernel for each memory access instruction by cloning its address computations. This kernel will then be utilized to recompute the address when the instruction is contaminated by a fault.

2.2 Structure of Scientific Applications

CARE is directly motivated by the complex address calculations that exist inside many scientific applications. For example, many scientific applications contain stencil codes, which are a class of iterative kernels. They store scientific data in a set of arrays (or vectors), and update their elements according to some fixed pattern using neighboring array elements. Such stencil pattern of data access is repeated for each element of the array. Due to this access pattern, applications have to maintain several data structures (mainly arrays) for storing the neighbor information. Therefore, to update an element in scientific data arrays, some amount of address calculation is required to access neighbor cells. Figure 2 presents an example code from GTC-P. It involves non-trivial address calculations when accessing an array element in *phitmp*, including 3 or 4 additions, 1 subtraction, and 1 multiplication. For the evaluated scientific workloads, as shown in Table 5, some large percentage of memory accesses (generally 86.85% $\sim$ 94.08%) have multiple binary operations in their address calculations; and each memory access instruction, on average, would involve 2.96 $\sim$ 5.6 binary operations. In addition, inside these applications, some variables used in the address calculation are infrequently updated during their executions,

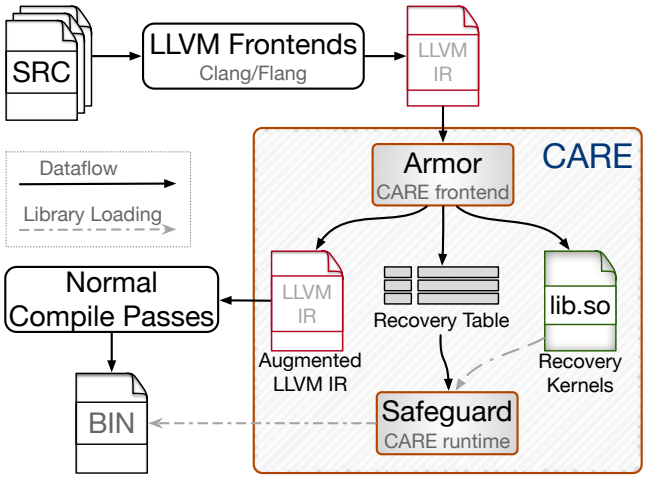

Figure 3: Overall architecture of CARE

```
%idxprom156 = sext i32 %i144.0 to i64
%arrayidx157 = getelementptr inbounds i32, i32* %7,↩
      i64 %idxprom156
%44 = load i32, i32* %arrayidx157, align 4
%arrayidx159 = getelementptr inbounds i32, i32* %8,↩
      i64 %idxprom156
%45 = load i32, i32* %arrayidx159, align 4
%add160 = add nsw i32 %44, %45
%sub161 = sub nsw i32 %add160, %29
%mul162 = mul nsw i32 %add66, %sub161
%add163 = add nsw i32 %mul162, %k.0
%idxprom164 = sext i32 %add163 to i64
%arrayidx165 = getelementptr inbounds double, ↩
     double* %12, i64 %idxprom164
%46 = load double, double* %arrayidx165, align 8
%sub169 = sub nsw i32 %44, %29
%mul170 = mul nsw i32 %add66, %sub169
%add171 = add nsw i32 %mul170, %k.0
%idxprom172 = sext i32 %add171 to i64
%arrayidx173 = getelementptr inbounds double, ↩
     double* %12, i64 %idxprom172
store double %46, double* %arrayidx173, align 8
```

Figure 4: LLVM IR Code for the example code in Figure 2.

once every few time steps or even unchanged during the whole execution. Consider the GTC-P code in Figure 2 for example, *igrid*, *mtheta* are never updated after they are initialized, and *igrid_in*, *mzeta* are unchanged when executing the loop. Because of the complexity of the address computations and the infrequently updated raw data for the computations, the invalid-memory-access errors due to transient faults are recoverable with a high probability for these situations. For example, if a failure manifests when accessing $phitmp[(mzeta+1)*(igrid[i]-igrid\_in)+k]$, the fault could have happened when updating i, k or the rest of computation, such as $mzeta+1$. While the prior cases are likely unrecoverable, the later cases are definitely recoverable. As comparison, a failure which occurs when accessing $phitmp[i]$ is less likely recoverable since the failure would imply a corrupted value of i, and the likelihood of the correct original value for that still existing is potentially quite small. Said another way, a failure manifesting in an address calculation which involves a large number of temporaries is naturally more likely to be recoverable than one which does not.

3 DESIGN AND PROTOTYPE OF CARE

Given the observations above, we designed the **CARE** environment to focus on recovery from *SIGSEGV* faults. Here, we first depict the overall architecture of **CARE**, and then dive into the design details of each component.

3.1 Overview

CARE is a compiler-assisted soft failure recovery mechanism for scientific applications. It is designed to help such applications survive soft failures with negligible overheads. As presented in Figure 3, **CARE** consists of two components: a front-end **Armor** and a runtime system **Safeguard**. **Armor** constructs recovery kernels for memory access instructions during the compilation of applications. To minimize the overhead of **CARE**, recovery kernels are compiled into a stand-alone shared library, and will be loaded by **Safeguard** as needed to repair invalid memory access errors. At the same time, **Armor** also generates a **Recovery Table** which contains information about how to access and execute a recovery kernel. **Safeguard** itself is designed and implemented as a shared library as well, and is automatically loaded by setting the *LD_PRELOAD* environment variable. Upon loading, it will overload the default *SIGSEGV* signal handler of applications to provide recovery service. Besides such initialization work, **Safeguard** is not activated unless a soft failure occurs, therefore it will incur almost negligible overheads during the normal execution of applications. Upon a invalid memory access error, **Safeguard** will be activated to diagnose the failure, find the recovery kernel and execute the kernel to repair the corrupted architecture state. Although the overall idea of **CARE** is straightforward, it comes with several challenges. In the rest of the section, we will present the design details for each component, as well as challenges we met and addressed in detail.

3.2 Armor

Armor is a compiler pass based on the LLVM framework [26]. It works on LLVM IR, a light-weight low-level intermediate representation of programs. There are several existing tools, such as Clang [10], Flang [17], and DragonEgg [14], which can be used to compile applications into LLVM IR codes. Therefore, **CARE** is relatively independent of programming languages, and can support a majority of scientific applications written in C, C++ or FORTRAN.

Figure 4 shows an example LLVM IR code for the stencil code in Figure 2. LLVM IR is in static single assignment (SSA) form. Its syntax is similar to MIPS assembly language, except that LLVM IR has unlimited virtual registers. Each LLVM IR instruction defines a new value which is used by other instructions. In LLVM IR, memory accesses are issued explicitly through either *LoadInst* or *StoreInst* instructions. For these memory access instructions, **Armor** starts from their address operands and works backwardly to identify instructions involved in their address computations. It then clones and organizes these instructions as a recovery kernel, represented as a normal function in LLVM IR code. **Armor** will construct a recovery kernel for each memory access instruction, except those directly loading from (or storing to) an *AllocaInst* (representing a local variable) or a *GlobalVariable*, since they don't involve any

```
bool isExpandable(Value *V, Value *MemAccInst) {
  if (isa<AllocaInst>(V) || isa<GlobalVariable>(V)
      || isa<Argument>(V) || isa<PHINode>(V) || ↩
          isComplexCalls(V))
    return false;

  auto operands = getOperands(V);
  for (op: operands) {
    if (!isLiveAt(op, MemAccInst) && !isExpandable(↩
        op, MemAccInst)) return false;
  }
  return true;
}

void getParamsAndStmts(Value *MemAccInst,
                       vector<Value *> &Params,
                       vector<Value *> &Stmts) {
  vector<Value *> Workspace;
  Value *Addr = getAddrOperand(MemAccInst);
  Workspace.push_back(Addr);
  while (!Workspace.empty()) {
    Value *V = Workspace.back();
    Workspace.pop_back();
    if (isExpandable(V)) {
      Stmts.insert(V);
      auto operands = getOperands(V);
      for(op: operands) {
        if (isa<ConstantData>(Op)) continue;
        Workspace.insert(Workspace.begin(), op);
      }
    } else Params.insert(V);
  }
}
```

Figure 5: Pseudo code for extracting address computations

address computations. Recovery kernels for an application are generated into a separate LLVM module, which is then compiled into a stand-alone shared library.

In general, a recovery kernel can repair transient faults that occur in instructions from which it is cloned, defining the **Coverage Scope** for that particular kernel. However, compiler optimization interacts with coverage scope in complex ways. To make a recovery kernel cover more instructions, **Armor** should clone as many instructions as possible. However, **Armor** cannot aggressively copy all computations. It will not only incur huge overheads, but also could reduce the fault coverage of **CARE** due to code optimizations in modern compilers, which could make values that are arguments for recovery kernels unavailable at the memory access instruction. **Armor** will stop the process of extraction when it meets predefined **Terminal Values**, where the intuition behind **Terminal values** is that they are guaranteed to be found in registers or memory. A formal definition of **Terminal Values** appears further down below. Informally, it is very critical to find correct **Terminal Values** for recovery kernels, since they are inputs to the kernels. When **Safeguard** is activated to repair a fault, these values must be extracted from the process and then provided to the recovery kernel subroutine in order to recreate the correct address. This requires that those values be accessible and not optimized away. However, **Armor** constructs the recovery kernel at LLVM IR level, and some of the LLVM IR values, like many variables in high-level languages, could be optimized away when they are compiled into machine code, particularly when the optimization flag is enabled. To address this challenge, we leveraged liveness analysis during the construction of recovery kernels. For a memory access instruction, the arguments of its recovery kernel should be live at its position. By definition, a variable is live at a particular point in the program if its value at that point will be used along at least one path that originates at the given program point. In particular, we leverage the following observation which is true about lowering of IR into machine code: if a value is live at a memory access instruction and its use is non-local (outside the current basic block), it is not optimized away by machine dependent passes (such as instruction selection etc) when the LLVM IR codes is lowered into the machine code. Therefore such a value is eligible as a parameter to the recovery kernel. Based on this insight, **Armor** leverages the algorithm in Figure 5 to extract address related computations for a memory access instruction. It will stop the process of extracting instructions when it meets one of the following LLVM IR instructions/values, since they imply start-points of the computation:

(1) An *AllocaInst* which represents an variable allocated on the function stack.
(2) A *GlobalVariable* which represents a global variable allocated on the data section of process.
(3) An *Argument* which represents a function parameter.
(4) A *PHINode* that represents a loop induction variable.
(5) A *CallInst* calling a complex function. We treat the *CallInst* differently based on the complexity of the callee. **Armor** will stop the process of extraction if the callee updates global variables, arguments passed to it (including memory regions pointed by arguments), or allocates new memory regions. In contrast, if the callee is a simple math operator, e.g., *sqrt*, it will be treated as a normal binary instruction.
(6) ***Terminal Value***. A Terminal Value for a memory access instruction I, is a LLVM IR instruction/value which is live at I, with at least one of its operands is dead at I and the dead operand cannot be computed from other live instructions/-values. Secondly, as stated earlier, the LLVM value which is live should have a non-local use outside the current basic block(which will ensure that the machine-dependent passes will not eliminate or fold the value making it unavailable). Finally, if every operand of a LLVM instruction/value is live or can be computed from other live values, **Safeguard** can continue the extraction of computations to extend the coverage scope for the kernel.

For illustration, Figure 6 presents a computation-dependency graphs among a set of variables. It shows the address computations for the *LoadInst* in node 1. To build a recovery kernel, **Armor** will start from node 2 and check liveness of *base* and $offset$. Since the $offset$ is live at node 1, and *base* can be computed from a live variable *gtc_input* in node 9, it will continue the extraction and take the instruction in node 2 as a statement in the recovery kernel, and then evaluate node 3 and node 11. The instructions in node 3 and node 11 will be copied as statements too, since both *mul* and *call* are live at node 1, and *densityi* can be computed from node 9. Node 10 is handled similarly to node 11. And node 4 and node 5 are then evaluated respectively. For node 4, its value *mul* will be considered as a parameter of the recovery kernel, since its operand *sub* is not live at node 1, and it cannot be also computed from other live variables. Similarly, for node 5, **Armor** will stop the search

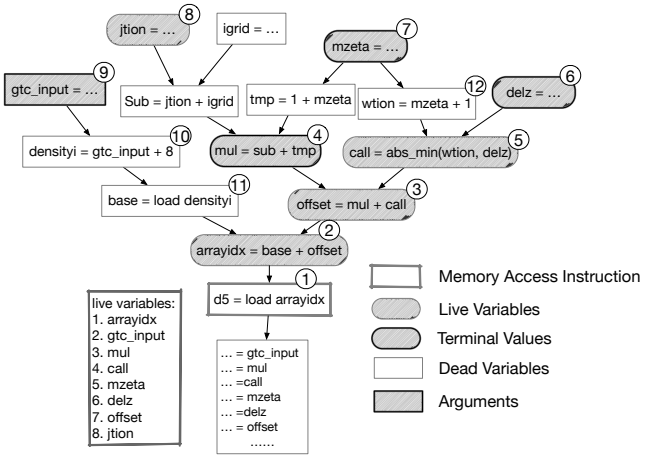

Figure 6: The illustration of constructing recovery kernels. *mzeta* and *delz* are computed from variables dead at node 1

until it meet node 6 and node 7. Finally, the recovery kernel for the memory access instruction in node 1 will clone the instructions in node 2, node 3, node 5, node 10, node 11, and node 12 as statements and take the values in node 4 (*mul*), node 6 (*delz*), node 7 (*mzeta*), and node 9 (*gtc_input*) as parameters of the kernel.

3.3 Recovery Table

In addition to the recovery library containing the recovery kernel subroutines, **Armor** also generates a **Recovery Table** to describe recovery kernels for **Safeguard**. It contains information about how to access a recovery kernel and which are the parameters to the kernel. The **Recovery Table** is a key-value table as shown in Table 6. For each recovery kernel in the recovery library, **Armor** will register an entry for it in the table. The recovery table contains three pieces of information:

(1) **key**, which represents an instruction. Each memory access instruction should be associated with a unique key, which will be used to retrieve the related recovery kernel.
(2) **symbol**, which represents a recovery kernel. The symbol could be simply the function name of the recovery kernel. It will be used to load the actual implementation of the recovery kernel from the recovery library.
(3) **parameters**, which describes the inputs required by the kernel. They are used to retrieve expected input values from the corrupted process.

Because **Armor** and **Safeguard** work on different representations of applications, the **Recovery Table** plays an important role in synchronizing the information between them. **Armor** relies on it to inform **Safeguard** which recovery kernel to use when a particular memory access instruction failed, and the parameters required to execute that kernel.

There are two challenges to be addressed here. First **Armor** and **Safeguard** must agree on the selection of the **key**, which is closely associated with the memory access instruction. For a memory access instruction in LLVM IR and its corresponding assembly instruction in machine code, both **Armor** and **Safeguard** should

Table 6: Recovery table for describing recovery kernels

key	symbol	parameters
key1	care_recovery_k1(int16, int, int)	*a*, *b*, *c*
key2	care_recovery_k2(float, int32)	*m*, *n*
key3	care_recovery_k3(int8, int64)	*d*, *e*

be able to generate the same key to point to the recovery kernel. **Armor** must generate a key at compile-time and associate it with the recovery kernel; meanwhile, **Safeguard** be able to generate the same key using the fault location, and use it to find the recovery kernel. Similarly, parameters are keys to the required input values of the kernel. **Safeguard** relies on them to retrieve input values for the kernel from the process's address space.

Intuitively, the instruction address is a good candidate for the **key**, since each instruction has unique address. It is stable, and easy to get for **Safeguard**. However, it is not available to **Armor**, since it is not generated until the code generation phase. Relying on it would require the modification of the code generation passes in modern compilers. To avoid this complexity, we leveraged the debug information subsystem of modern compilers, which is used to encode source-level program information for machine code. Although **CARE** leveraged this subsystem, it doesn't have to rely on the debug data generated by compiler. In debug data of a program, each instruction is associated with location description, which contains the source file name, the line number and the column number. **CARE** takes the tuple of $(file, line, column)$ as the key to an instruction, since they are accessible both in LLVM IR and in machine code. Specially, **CARE** doesn't require the real debug data of the program, since it won't map instructions to original source-code statements. **CARE** only requires that the debug data is unique for each memory access instruction. **Armor** can generate a fake debug data for each memory access instruction if the debug flag is not enabled. On the other hand, if debug flag is enabled during the compilation, **Armor** needs to resolve the conflicts for some instructions that end up sharing the same debug data. As an additional complexity, since x86_64 assembly supports CISC-style memory access in computations (e.g., "add (%rax, %rcx, 8), %rdx" reads data from memory and adds it to %rdx), some of memory access instructions in LLVM IR might be merged with the related binary instruction during the code generation. Hence, **Armor** also attaches the debug information for memory access instructions to the instructions that directly use their results.

The use of debug mechanism also addresses the second challenge about retrieving arguments for a recovery kernel. For each parameter of a kernel, **Armor** will create a variable description for it by simply assigning a unique name for each parameter. Based on the variable description, the debug information subsystem of the compiler will automatically generate a debug information entry (DIE) to describe the variable in machine code, as shown in Table 7. A DIE contains several attributes which are associated with the variable. An important attribute is the "DW_AT_location", which describes the location for a variable. It contains 2 pieces of information, including address ranges and corresponding location of the variable. For example, item [0] in Table 7 describes that the variable *zion*3 is located in a register if PC address resides in

Table 7: An example of Debug Information for a local variable. It is simplified for ease of reading.

DW_TAG_Variable	
DW_AT_location	<loclist with 2 entries follows> [0] 0x422cd4~0x422d3c: DW_OP_reg11 [1] 0x422d3c~0x422fe4: DW_OP_breg7+4
DW_AT_name	zion3
DW_AT_decl_file	"/path/to/source/file.c"
DW_AT_decl_line	156

Algorithm 1 The pseudo code for repairing a corrupted process

```
function SAFEGUARD(signo, pcontext)
    Addr ← GETADDROFINSTRUCTION(pcontext)
    key ← GETSRCINFO(Addr)
    kname, params ← SEARCHRECOVERYTABLE(key)
    if no kernel found then
        EXIT(signo)
    end if
    lib ← DLOPEN(libRecovery)
    kfunc ← DLSYM(lib, kname)
    pvalues ← GETPARAMSVALUES(params)
    value ← KFUNC(pvalues)
    operand ← GETADDROPERAND(Addr)
    UPDATE(operand, value)
end function
```

$[0x422cd4, 0x422d3c)$, and the item [1] describes that the variable is located on the stack at the offset 4 to the frame point register if PC address is in $[0x422d3c, 0x422fe4)$.

3.4 Safeguard

Safeguard is the runtime system of **CARE**, providing recovery service for applications by setting up and executing recovery kernels constructed by **Armor**. **Safeguard** is built as a shared library, and it is designed to be loaded automatically by setting the *LD_PRELOAD* environment variable. Leveraging the "constructor" attribute in modern compilers, **Safeguard** will add a signal handler for *SIGSEGV* immediately after it is loaded. Except this initialization work, **Safeguard** is not activated until a *SIGSEGV* fault occurs. Therefore, it has almost **zero** runtime overhead during the normal execution of applications. To minimize the memory overhead, **Safeguard** only loads recovery kernels when a crash-causing error is detected, and will immediately release the related memory after the repair. Upon a failure, the steps taken by **Safeguard** are shown in Algorithm 1. It first retrieves the address for the instruction that issued the *SIGSEGV* signal. Based on the address, **Safeguard** will read the line table of the debug data to get the key for the instruction, and then use the key to find the appropriate recovery kernel from the **Recovery Table**. If successful, it will load the recovery library, retrieve the kernel implementation, decode the debug data to find and retrieve values for parameters, and then execute the recovery kernel. Finally, it will disassemble the instruction, to find its address operand, and update that operand with the value computed by the kernel. If the address operand involves both a base register and a index register, e.g. "mov 8(%rbx, %r8, 4), %eax", **Safeguard** will update the index register (%r8) by default, assuming that index register is computed more frequently than base register, and are more likely to experience faults. It will recompute the value for the index register based on the value in base register, and the value returned by recovery kernel. Before making the actual update, **Safeguard** will check whether the kernel-computed address is the same with the invalid address accessed by the instruction. The update is performed only if they are different. Otherwise, it implies that the fault happened to an instruction that is out of the coverage scope of the recovery kernel, and its argument values are contaminated. **CARE** lacks of ability to recover such failures.[2]

4 PROTOTYPE

We implemented a prototype of **CARE** on X86_64 platform and Linux OS. We implemented **Armor** based on LLVM 6.0.1. **Armor** treated some LLVM *CallInst* instructions as a normal binary operators, if they simply call some mathematical kernels, e.g, *sqrt*, or some user-implemented functions that don't update global variables and arguments. But it doesn't clone the implementation of these callee functions, hence, when the recovery kernels are compiled into a shared library, it is necessary to build them with binary source files containing the user-implemented simple functions, and link them with necessary libraries.

On the other hand, **Safeguard** mainly implements a signal handler for *SIGSEGV*. It contains a constructor, which will be executed automatically (by setting *LD_PRELOAD*) to setup/overload the signal handler for processes. The benefit of this design is that **Safeguard** doesn't require source code changes to applications. **Safeguard** computes the address for the failed instruction based on the location where the failure occurs. Failure can happen to either an instruction belonging to the application code or to one belonging to the library code. For a failure occurring on applications' code instruction, it will directly use the *PC* (Program Counter) value corresponding to the failed instruction as the address of the instruction, and for a failure that happened in a shared library, the $offset$, calculated as $PC - base$, is used as the address of the instruction. Here, *base* is the address at which the library is loaded. This design is mainly restricted due to the differences in mechanisms in terms of encoding the debug data for executable binaries versus the shared libraries. Getting the correct address is the key to retrieve the correct recovery kernel, and related parameters. **Safeguard** utilizes *dladdr* to diagnose the location of failures. The failure location also guides the **Safeguard** to access the correct file for the required debug data.

In addition, **Safeguard** relies on the libdwarf [24] library to read the debug data and the libffi [25] library to execute calls to the recovery kernel. Since "ffi_call" takes pointers as arguments, the address of a variable, instead of a value, is retrieved from the process space. Finally, **recovery table** is implemented based on google protobuf-3.6.0 [19], and the MD5 hash of the debug information tuple $(file, line, column)$ is computed with the mhash [28] library and used as the key.

[2]Note that a real segmentation fault resulting from program bug or erroneous input will fall into this category as well. **CARE** will declare it non-recoverable and simply propagate the SIGSEGV.

Table 8: Statistics of recovery kernels

	GTC-P	HPCCG	miniMD	CoMD
Num. of kernels	2786	255	2611	1143
Avg. Instructions (IR)	19.18	2.51	8.2	2.63
Normal Compilation (seconds)	2.322	3.575	4.215	1.486
Armor Overhead (seconds)	226.89	35.538	57.528	14.655

5 EVALUATION

We mainly evaluated **CARE** with 4 scientific workloads including GTC-P, HPCCG, miniMD and CoMD as described in Table 1. We skipped miniFE since it heavily relies on the C++ STL library which is not fully supported in current prototype. For each workload, the compile-time overheads spent on building recovery kernels and the statistical information about the recovery libraries are presented in Table 8. As shown in the table, **Armor** would take tens of seconds for HPCCG, miniMD, and CoMD, or hundreds of seconds for GTC-P, to analyze the program and construct the recovery kernels. More than 90% of the overheads were spent on liveness analysis. Despite this offline compile-time overheads shown, **CARE** incurs almost **zero** runtime overhead during the normal execution of the workloads without faults, since it only calls "sigaction" to register signal handlers, which takes a few microseconds. And the memory overheads of **CARE** is fixed to 27MB (< 1% for evaluated workloads), which is mainly occupied by partial of LLVM and protobuf libraries used by **Safeguard** for encoding/decoding recovery tables. We believe this overhead is negligible for scientific applications that are typically with gigabytes of memory footprints. In this section, we mainly focus on evaluating the fault coverage and recovery time of **CARE**. For these workloads, faults are injected to instructions from applications [3]. In the rest of the section, we will introduce the evaluation methodology and environment, present evaluation results, and discuss the advantages and limitations of **CARE**.

5.1 Methodology

We evaluated **CARE** on an X86_64 platform with up to 64 compute nodes, with each node equipped with 48 cores (3072 total cores) and 128GB of memory. We performed fault injections to emulate transient faults with a methodology similar to that introduced in Section 2, except that we updated the method of randomly selecting a dynamic instruction, such that injections are only performed to instructions from the application itself and not to library code. In the updated tool, we first profiled the number of executions for each static instruction (from applications only) using the Intel Pin tool. Then we randomly select a static instruction for injection based on the numerical distribution of their executions. Finally, we generate a random number based on the executions of the selected instruction to determine the point at which the fault would be injected. In other words, a dynamic instruction is approximately represented by a pair (I, n), which means the fault will be injected to the instruction I after it is executed n times. In all, we examined around 1000 ~ 2000 injections that led to *SIGSEGV* errors. In the discussions below it is important to note that **CARE** is largely insensitive to the exact fault model in use. Different choices for fault models would likely change the relative ratios of fault outcomes (such as provided in Section 2, but if a fault triggers a soft failure, the number of corrupted bits will not impact **CARE**'s operation. Only the actual location of the fault will impact whether or not **CARE** can recover from it.

[3] **CARE** relies on source code to build recovery kernels, and recovery from transient faults that occur in library code requires the recompilation of libraries from their source codes leveraging **CARE**. which is beyond the scope of our current work, IR recovery binary is the key.

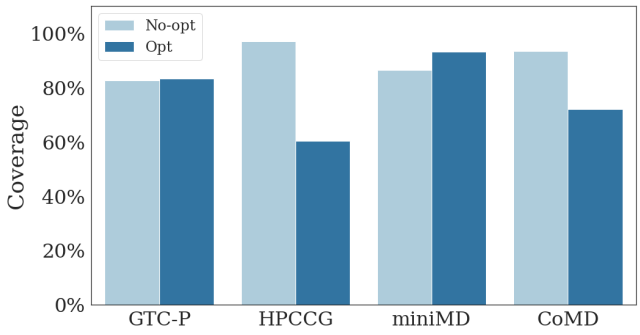

Figure 7: Fault Coverage of CARE.

```
1 int a, b, c, d, *array;
2 array[a+b+c+d];  // case 1
3
4 a += b;
5 c += d;
6 array[a+c];      // case 2
```

Figure 8: Code optimization could help extend the coverage scope for case 2 to the same on for case 1

5.2 Fault Coverage

CARE is a process recovery technique, which aims to recover processes from invalid memory access errors caused by transient faults. In this subsection, we evaluated the performance of **CARE** based on fault-injection experiments on single process. We evaluated **CARE** when applications are compiled with "-O0" (No-opt) or "-O1" (Opt) flags. To support "-O2" and "-O3", which will perform code vectorizations, our current prototype still needs extra engineering work to encode vector-type parameters in recovery tables.

Figure 7 presents the fault coverage of **CARE**. On average, **CARE** can recover 83.54% of injected *SIGSEGV* faults, with up to 96% for HPCCG when it is compiled without optimization. **CARE** achieved such high fault coverage mainly due to the fact that majority of *SIGSEGV* faults manifest quickly, typically within only a few dynamic instructions after they occur. The raw data used in address computations is less likely to get updated during such a short time window, especially in the evaluated workloads where they are infrequently updated at the algorithm level. Therefore, **CARE** has a good chance to recompute the addresses. Despite some variance, code optimization didn't introduce significant reduction for the fault coverage of **CARE**. For miniMD, it improved fault coverage by around 7%. this mainly due to code optimization extending the coverage scope of recovery kernels in the miniMD core. This scenario can be illustrated as follows. Figure 8 shows two memory accesses. When the code is compiled without optimization, **CARE** can recover errors occurring during the computation of $a+b+c+d$

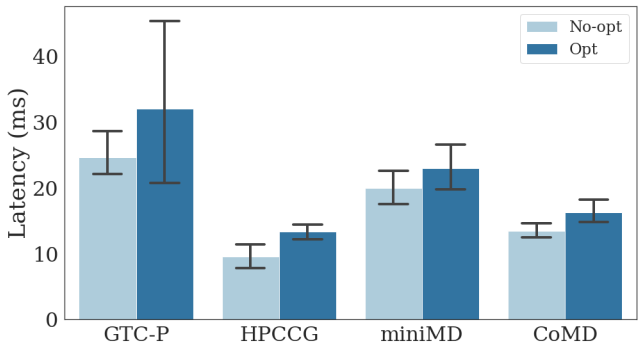

Figure 9: Recovery time of CARE

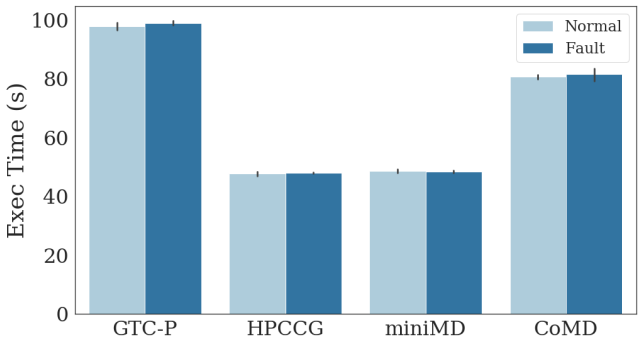

Figure 10: Parallel jobs can finish the computations without delays with SIGSEGV recovered by CARE

for case 1, but can only recovery errors occurring in $a + c$ for case 2 because of the immediate update of a and c in line 6 and 7. Code optimization helps to optimize the case 2 to be like case 1 by optimizing out unnecessary memory updates. As a result, the recovery scope of the kernel is extended. Similarly, there is a slight improvement for GTC-P as well. For HPCCG and CoMD, however, code optimization reduced the coverage by 35% and 21%, respectively. For HPCCG, which is a relative simple kernel, a significant portion of dynamic instructions are involved in updating the loop induction variables after the code optimization, therefore they are more likely to be selected by our tool to inject faults. More importantly, because of the code optimization, loop induction variables will be allocated in registers, and updated in-place. If they are corrupted, **CARE** cannot acquire correct values for related recovery kernels, leaving many faults unrecoverable. For CoMD, however, it is mainly because recovery kernels don't have enough coverage scope due to liveness issues.

It is worth noting that during a recovery of failure, **CARE** will not substitute silent data corruptions (SDCs) for failures as is possible with more heuristic based recovery methods [27]. This is because the computation of a recovery kernel is based on the raw data fetched from the process. If raw data is contaminated by a fault, the recovery kernel will definitely generate a wrong address which is the same as the one accessed by the corrupted instruction. Otherwise, **CARE** is guaranteed to get correct address, since it exactly clones the computation from applications.

5.3 Recovery Time

Recovery time measures the time required by **CARE** to recover from a fault. Clearly a single faulted computation might feed into several memory access instructions. What might not be intuitively obvious is that in this situation, **Safeguard** could be activated several times, recovering the effects of each manifestation of the fault. Figure 9 shows that **CARE** can recover a process from a *SIGSEGV* fault with only a few tens of milliseconds. In fact, only a tiny percentage of that recovery time is spent in the generated recovery kernel. They generally only contain a few instructions related to address computations and while their use is key to **CARE**, their actual portion of the recovery time is negligible. In fact, for each activation, more than 98% of the recovery time is spent on preparing the execution of recovery kernels, including diagnosing the failure, loading recovery table and recovery library, and retrieving arguments from stalled process. When applications are compiled with "-O1", the recovery time will be on average increased by around 8% ∼ 17%. This is because, when code is optimized, a single fault injection is more likely to impact multiple instructions requiring multiple recover kernel executions. In relative terms, HPCCG tends to have a lower recovery time because it has a small code base, hence small recovery table and recovery library.

5.4 Impact on parallel jobs

In this subsection, we examine the impact of **CARE** on parallel jobs. We run the workloads with 512 processes and 6 threads on a cluster with 64 nodes (3072 cores). For each run, we injected a **CARE**-recoverable fault to rank 0 of the job. We wrote a wrapper to *PMPI_Init*, in which an injection process is created and attached to rank 0 using *ptrace*. For a injection point (I, n), the injection process will set a break-point at I, stop the rank 0 after the instruction I is executed n times, and then contaminate the destination operand of the target instruction and continue the execution of process. We performed 100 injections, one injection per-run. Table 10 compares the execution time of parallel jobs when a *SIGSEGV* fault occurred in rank 0 and is repaired by **CARE**. It shows that, despite some execution variance across different runs, **CARE** can almost completely mask the impact of recoverable *SIGSEGV* faults to parallel jobs. It can help parallel jobs to survive invalid memory access errors caused by transient faults. With the protection from **CARE**, parallel jobs when experiencing an invalid memory access error can finish their computations with almost no delays as compared to their normal executions. This is due to the low recovery time of **CARE**. In comparison, even a small run of GTC-P relying upon checkpoint/restart would require at a minimum dozens of seconds (14.367s, 25.946s, or 37.56s on average to recover from a failure if checkpoint is respectively scheduled every 20, 50, or 75 time-steps) to recover from a failure, even if some automatic restart mechanism were available and the new job were to be scheduled immediately.

5.5 Failures in shared library – *BLAS*

BLAS is a popular linear algebra library written in FORTRAN. It prescribes a set of low-level routines for performing common linear algebra operations, and is widely used in many scientific applications

Table 9: Statistics and Performance for *sblat1*/*BLAS*

	# Kernels	Normal Compile time (Sec)	**Armor** Overhead (Sec)	Coverage	Recovery Time
BLAS	10931	6.89	4.973	83.49%	5.7ms
sblat1	187	1.78	1.98		

and machine learning workloads. BLAS routines are categorized into three levels, which correspond to the degree of the polynomial. In this subsection, its preliminary evaluation results are presented.

We rebuilt the *BLAS* from its source code (copied from *LAPACK-3.8.0*) as a shared library ("libblas.so"). The test program sblat1 (in "TESTING/sblat1.f") for REAL level 1 blas, which is provided by the package, was used as the driver to the library. *sblat1* is dynamically linked to "libblas.so". We randomly injected faults into either *BLAS* or *sblat1*. Table 9 briefly presents the statistics of recovery kernels for *sblat1* and *BLAS*, and the performance achieved by **CARE** for them. It shows that **CARE** achieved around 83% coverage with almost negligible overheads. We should point out that *sblat1* only covers 12 REAL level-1 routines, which is a small fraction of procedures provided by *BLAS*. While a detailed evaluation for BLAS is currently underway, the preliminary results in this subsection demonstrate that **CARE** effectively supports the recovery of failures which could occur in libraries with good coverage, small overheads and recovery times.

5.6 Discussion

As shown in previous results, **CARE** is a light-weight and effective failure recovery mechanism for invalid memory access errors triggered by transient faults. Although it is not designed to replace the C/R, its high fault coverage rate, low recovery time, negligible memory overhead and almost **zero** runtime overhead could help to relax the frequent checkpoint to be a relatively infrequent one, thereby reducing the overall overheads of resiliency mechanisms, and speeding up the execution of scientific applications.

We examined the remaining 16% cases for which **CARE** could not provide recovery. In addition to the limited support to C++ STL library, the other two main reasons were inability to recover induction variables and limitations on live ranges. Induction variables always perform in-place updates, and are always put as parameters to recovery kernels. If the in-place update for a induction variable is contaminated by a fault, **CARE** currently has no ability to recover the correct value for the variable. On the other hand, the requirement of live values for parameters strictly restricts the coverage scope of a kernel, and makes the faults that contaminated parameters of recovery kernels unrecoverable. As a part of ongoing work, we are exploring compiler transformations to handle these cases and further improve **CARE**'s coverage.

6 RELATED WORK

Detection and recovery from failures are not new topics in HPC and other environments [3, 6–8, 16], so there is significant prior work to consider. In this section, we present a brief survey of studies most related to **CARE**.

Studies in [3, 9, 23] examined the impact of transient faults on scientific applications. Their results showed that a significant portion

```
1 for (i = 0; i < N; i++) {
2     ... = array[f(i)];
3     ptr++;
4 }
```

Figure 11: An example of exploring equivalent computation for induction variable recovery. If *i* is corrupted, and a SIGSEGV is issued at $array[f(i)]$, the correct value for i can be derived from ptr if the initial values for ptr and i are known.

of transient faults could manifest as soft failures. This motivated us to study how soft failures manifest inside scientific applications and whether there are common features that can be explored to design an efficient resilience mechanism for them, resulting in the design of **CARE**. Georgakoudis et al. [18] and Chang et al. [5] designed and evaluated new fault injection tools for transient faults. While these tools are valuable to the community, they are not a good fit for **CARE** because they either work on high-level intermediate representations (LLVM machine IR) which is inaccurate as compared binary-level injections, or incur high overheads (3× slowdown) making it infeasible to run large scale (~100 000 injections) fault injection experiments.

Besides **CARE**, there are several studies on online recovery from process failures such that applications can continue their normal executions. Rx [32] aims to recover from a process failure by rolling applications back to a previous safe status, and then continuing its execution with a minor modification to its environment. Rx is motivated by the observation that many program bugs are associated with the setup of process environments, so changing the environment setup could avoid the crashes. Its techniques could help handle transient faults by simply replaying the computation *without* changing the environment, however its basic operation requires at least partial application checkpoints which are likely to have significant cost. RCV [27] is another online failure recovery technique for divide-by-zero (*SIGFPE*) and null-dereference (*SIGSEGV*) errors. RCV's approach explores a set of heuristics for recovery. For instance, it returns zero as the default result of the divide for divide-by-zero errors, discards invalid write instructions that accessing near-to-zero addresses and returns 0 for invalid read operations. These techniques are computationally inexpensive and may succeed in getting the application to continue, but are likely to introduce SDCs as a side effect. LetGo [16] shares a similar idea to RCV, and is specially designed for handling soft failures in scientific applications. Its recovery strategy employs a set of heuristics too. Upon a failure, it will reset architecture states to a pre-defined value, and then continue the execution of the application. Obviously such heuristic based method could lead to SDCs, which is another challenge problem in HPC community.

In contrast, **CARE** undertakes a proper recovery process with regards to the maligned address computation by recomputing it as per the program semantics and through the use of in-tainted values by synthesizing a very lightweight function. It develops careful correspondence mechanism to co-relate the recovery handlers to the fault causing instruction at runtime. While **CARE** shares the similar goal and design to RCV and LetGo in that they all aim to help applications to survive failures by replacing the default signal

handler with their own one to provide recovery services, **CARE**'s approach is more accurate than others, and will not introduce SDCs.

7 CONCLUSION AND FUTURE WORK

Resilience is projected to be a critical challenge as HPC systems continue to grow and as individual components continue to shrink feature size and operating voltages. These technology trends would make the system more susceptible to transient faults caused by such things as high-energy particle strikes and heat flux. Transient faults could not only lead scientific applications to generate incorrect outputs, but also crash the execution of a process, killing the entire parallel job as a result. This requires the application to be restarted from a latest checkpoint, and the lost computation to be redone before continuing the execution. The presence of such failures could also necessitate more frequent checkpoints than would otherwise be desirable, leading to significant overheads.

In this paper, we present and evaluate **CARE**, a lightweight and compiler-assisted error-recovery mechanism that allows processes to survive crashes caused by certain transient faults, such that the applications can continue their execution. Based on our experimental studies, we identified *SIGSEGV* as a major symptom to soft failures, and **CARE** is designed for repairing *SIGSEGV* errors. For each memory access instruction that involves complex address computations, **CARE** first builds a recovery kernel by cloning its address computations. At runtime, it maps the fault causing instruction to a failure recovery handler which recomputes the address and masks the fault. We evaluated **CARE** with four scientific workloads. During their normal executions, **CARE** incurs almost **zero** runtime overhead and fixed 27MB memory overheads, and it can recover averagely to 83% *SIGSEGV* faults within a few milliseconds.

We also demonstrated the impact of **CARE** on parallel jobs at the scale of 3072 cores. Due to the low-latency recovery mechanism of **CARE**, the parallel jobs can finish their computations as normal with almost no delays, even if a crash-causing error occurred and repaired by **CARE** during their executions. These results show that **CARE** can allow applications to survive some classes of transient failures with little or no overhead, which is of particular benefit in HPC scenarios where the cost of application failure is high. **CARE** could also improve MTBF and therefore could open a door towards research on relaxation of the checkpoint frequency which could have significant resource and performance implications.

In our future work, we plan to improve the **CARE** in the directions of: 1) exploring equivalent computation in programs to recover induction variables (See Figure 11); 2) extending its runtime to support instructions common in high level optimizations; 3) improving its support for libraries, especially those for which source code might be unavailable, e.g., C++ STL, with inter-procedure analysis and stack unwinding.

A DOUBLE-BIT-FLIP MODEL

For careful readers who are interested in the performance of **CARE** under the multi-bit-flip fault model, this section briefly presents experimental data with the double-bit-flip fault model. We leveraged the same tool from section 2 and section 5, except that, for each injection, we randomly flipped two bits rather than one in the injection target.

Table 10: The overall outcomes (double-bit-flip model)

Workloads	Benign	Soft Failure	SDC	Hang
HPCCG	3020	4244	2710	26
CoMD	5271	4392	337	0
miniFE	4861	3643	1247	208
miniMD	11	4556	5424	9
GTC-P	6539	2474	985	2

Table 11: Breakdown of soft failures (double-bit-flip model)

	SIGSEGV	SIGBUS	SIGABRT	Other
HPCCG	4236	2	1	5
CoMD	4074	53	9	256
miniFE	3511	75	34	23
miniMD	4362	84	32	78
GTC-P	2050	37	362	25

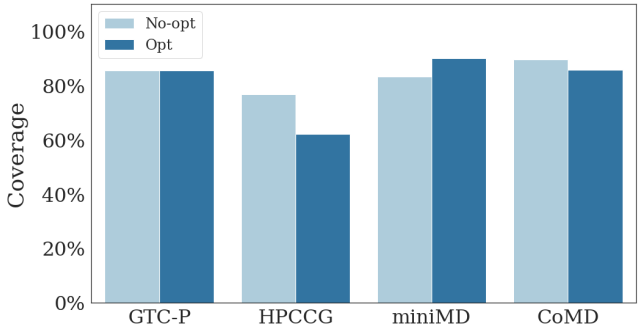

Figure 12: Fault Coverage (double-bit-flip model)

Table 10 and Table 11 present the overall outcomes of injection experiments. On average, 38.49% of transient faults were manifested as soft failures with 82.86% ~ 99.81% of them manifested as *SIGSEGV*, which are a little bit higher than single-bit-flip fault model. In double-bit-flip fault model, soft failures have shorter manifestation latency than in single-bit-flip model, and all of these soft failures are manifested within 10 dynamic instructions.

Finally, Figure 12 presents the fault coverage of **CARE** with the double-bit-flip fault model. On average, it achieved 82.34% coverage for the evaluated workloads. As compared to the single-bit-flip model, the fault coverage for HPCCG(Non-opt) is reduced by ~ 18%. This is mainly because a significant portion of fault injections are performed on loop induction variables, which are unrecoverable by **CARE**. In such case, while single-bit-flip faults could probably lead to SDCs if they are injected to lower bits, multi-bit-flip faults could reduce this possibility, and are more likely to trigger soft failures. For other cases, **CARE** achieved comparable or slightly better coverage under double-bit-flip fault model, which could be mainly due to low manifestation latency.

REFERENCES

[1] Saman Amarasinghe, Dan Campbell, and William Carlson etc. 2009. *Exascale Software Study: Software Challenges in Extreme Scale Systems.* Technical Report. DARPA IPTO, Air Force Research Labs. DOI: http://dx.doi.org/10.1088/1742-6596/180/1/012045

[2] Rizwan A. Ashraf, Roberto Gioiosa, Gokcen Kestor, Ronald F. DeMara, Chen-Yong Cher, and Pradip Bose. 2015. Understanding the Propagation of Transient Errors in HPC Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. ACM, New York, NY, USA, Article 72, 12 pages. DOI: http://dx.doi.org/10.1145/2807591.2807670

[3] Jon Calhoun, Marc Snir, Luke N. Olson, and William D. Gropp. 2017. Towards a More Complete Understanding of SDC Propagation. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '17)*. ACM, New York, NY, USA, 131–142. DOI: http://dx.doi.org/10.1145/3078597.3078617

[4] Franck Cappello, Al Geist, Bill Gropp, Laxmikant Kale, Bill Kramer, and Marc Snir. 2009. Toward Exascale Resilience. *Int. J. High Perform. Comput. Appl.* 23, 4 (Nov. 2009), 374–388. DOI: http://dx.doi.org/10.1177/1094342009347767

[5] Chun-Kai Chang, Sangkug Lym, Nicholas Kelly, Michael B. Sullivan, and Mattan Erez. 2018. Evaluating and Accelerating High-fidelity Error Injection for HPC. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '18)*. IEEE Press, Piscataway, NJ, USA, Article 45, 13 pages. http://dl.acm.org/citation.cfm?id=3291656.3291716

[6] Chao Chen, Greg Eisenhauer, Matthew Wolf, and Santosh Pande. 2018. LADR: Low-cost Application-level Detector for Reducing Silent Output Corruptions. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '18)*. ACM, New York, NY, USA, 156–167. DOI: http://dx.doi.org/10.1145/3208040.3208043

[7] Zizhong Chen. 2011. Algorithm-based Recovery for Iterative Methods Without Checkpointing. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing (HPDC '11)*. ACM, New York, NY, USA, 73–84. DOI: http://dx.doi.org/10.1145/1996130.1996142

[8] Zizhong Chen. 2013. Online-ABFT: An Online Algorithm Based Fault Tolerance Scheme for Soft Error Detection in Iterative Methods. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13)*. ACM, New York, NY, USA, 167–176. DOI: http://dx.doi.org/10.1145/2442516.2442533

[9] Chen-Yong Cher, Meeta S. Gupta, Pradip Bose, and K. Paul Muller. 2014. Understanding Soft Error Resiliency of BlueGene/Q Compute Chip Through Hardware Proton Irradiation and Software Fault Injection. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*. IEEE Press, Piscataway, NJ, USA, 587–596. DOI: http://dx.doi.org/10.1109/SC.2014.53

[10] Clang 2019. Clang. https://clang.llvm.org. (2019).

[11] Majid Dadashi, Layali Rashid, Karthik Pattabiraman, and Sathish Gopalakrishnan. 2014. Hardware-Software Integrated Diagnosis for Intermittent Hardware Faults. In *Proceedings of the International Conference on Dependable Systems and Networks*. IEEE, Atlanta, GA, USA, 363–374. DOI: http://dx.doi.org/10.1109/DSN.2014.1

[12] S. Di and F. Cappello. 2016. Fast Error-Bounded Lossy HPC Data Compression with SZ. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, Chicago, IL, USA, 730–739. DOI: http://dx.doi.org/10.1109/IPDPS.2016.11

[13] Jack Dongarra, Pete Beckman, Terry Moore, Patrick Aerts, Giovanni Aloisio, Jean-Claude Andre, David Barkai, Jean-Yves Berthou, Taisuke Boku, Bertrand Braunschweig, Franck Cappello, Barbara Chapman, Xuebin Chi, Alok Choudhary, Sudip Dosanjh, Thom Dunning, Sandro Fiore, Al Geist, Bill Gropp, Robert Harrison, Mark Hereld, Michael Heroux, Adolfy Hoisie, Koh Hotta, Zhong Jin, Yutaka Ishikawa, Fred Johnson, Sanjay Kale, Richard Kenway, David Keyes, Bill Kramer, Jesus Labarta, Alain Lichnewsky, Thomas Lippert, Bob Lucas, Barney Maccabe, Satoshi Matsuoka, Paul Messina, Peter Michielse, Bernd Mohr, Matthias S. Mueller, Wolfgang E. Nagel, Hiroshi Nakashima, Michael E Papka, Dan Reed, Mitsuhisa Sato, Ed Seidel, John Shalf, David Skinner, Marc Snir, Thomas Sterling, Rick Stevens, Fred Streitz, Bob Sugar, Shinji Sumimoto, William Tang, John Taylor, Rajeev Thakur, Anne Trefethen, Mateo Valero, Aad Van Der Steen, Jeffrey Vetter, Peg Williams, Robert Wisniewski, and Kathy Yelick. 2011. The International Exascale Software Project Roadmap. *Int. J. High Perform. Comput. Appl.* 25, 1 (Feb. 2011), 3–60. DOI: http://dx.doi.org/10.1177/1094342010391989

[14] DragonEgg 2019. DragonEgg. https://dragonegg.llvm.org. (2019).

[15] James Elliott, Kishor Kharbas, David Fiala, Frank Mueller, Kurt Ferreira, and Christian Engelmann. 2012. Combining Partial Redundancy and Checkpointing for HPC. In *Proceedings of the 2012 IEEE 32Nd International Conference on Distributed Computing Systems (ICDCS '12)*. IEEE Computer Society, Washington, DC, USA, 615–626. DOI: http://dx.doi.org/10.1109/ICDCS.2012.56

[16] 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 the 26th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '17)*. ACM, New York, NY, USA, 117–130. DOI: http://dx.doi.org/10.1145/3078597.3078609

[17] Flang 2019. FLANG. https://github.com/flang-compiler/flang. (2019).

[18] Giorgis Georgakoudis, Ignacio Laguna, Dimitrios S. Nikolopoulos, and Martin Schulz. 2017. REFINE: Realistic Fault Injection via Compiler-based Instrumentation for Accuracy, Portability and Speed. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*. ACM, New York, NY, USA, Article 29, 14 pages. DOI: http://dx.doi.org/10.1145/3126908.3126972

[19] gprotobuf 2019. Google Protobuf. https://developers.google.com/protocol-buffers/. (2019).

[20] Michael A. Heroux. 2013. Toward Resilient Algorithms and Applications. In *Proceedings of the 3rd Workshop on Fault-tolerance for HPC at Extreme Scale (FTXS '13)*. ACM, New York, NY, USA, 1–2. DOI: http://dx.doi.org/10.1145/2465813.2465814

[21] Saurabh Hukerikar and Robert F. Lucas. 2016. Rolex: resilience-oriented language extensions for extreme-scale systems. *The Journal of Supercomputing* 72, 12 (01 Dec 2016), 4662–4695. DOI: http://dx.doi.org/10.1007/s11227-016-1752-5

[22] Dmitrii Kuvaiskii, Rasha Faqeh, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2016. HAFT: Hardware-assisted Fault Tolerance. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. ACM, New York, NY, USA, Article 25, 17 pages. DOI: http://dx.doi.org/10.1145/2901318.2901339

[23] Dong Li, Jeffrey S. Vetter, and Weikuan Yu. 2012. Classifying Soft Error Vulnerabilities in Extreme-scale Scientific Applications Using a Binary Instrumentation Tool. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*. IEEE Computer Society Press, Los Alamitos, CA, USA, Article 57, 11 pages. http://dl.acm.org/citation.cfm?id=2388996.2389074

[24] libdwarf 2019. libdwarf. https://www.prevanders.net/dwarf.html. (2019).

[25] libffi 2019. libffi. https://sourceware.org/libffi/. (2019).

[26] llvm 2019. LLVM. https://llvm.org. (2019).

[27] Fan Long, Stelios Sidiroglou-Douskos, and Martin Rinard. 2014. Automatic Runtime Error Repair and Containment via Recovery Shepherding. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 227–238. DOI: http://dx.doi.org/10.1145/2594291.2594337

[28] mhash 2019. mhash. http://mhash.sourceforge.net/. (2019).

[29] S. Mitra, P. Bose, E. Cheng, C. Cher, H. Cho, R. Joshi, Y. M. Kim, C. R. Lefurgy, Y. Li, K. P. Rodbell, K. Skadron, J. Stathis, and L. Szafaryn. 2014. The resilience wall: Cross-layer solution strategies. In *Proceedings of Technical Program - 2014 International Symposium on VLSI Technology, Systems and Application (VLSI-TSA)*. IEEE Press, Hsinchu, Taiwan, 1–11. DOI: http://dx.doi.org/10.1109/VLSI-TSA.2014.6839639

[30] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R. de Supinski. 2010. Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*. IEEE Computer Society, Washington, DC, USA, 1–11. DOI: http://dx.doi.org/10.1109/SC.2010.18

[31] Daniel Oliveira, Laércio Pilla, Nathan DeBardeleben, Sean Blanchard, Heather Quinn, Israel Koren, Philippe Navaux, and Paolo Rech. 2017. Experimental and Analytical Study of Xeon Phi Reliability. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*. ACM, New York, NY, USA, Article 28, 12 pages. DOI: http://dx.doi.org/10.1145/3126908.3126960

[32] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. 2005. Rx: Treating Bugs As Allergies—a Safe Method to Survive Software Failures. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles (SOSP '05)*. ACM, New York, NY, USA, 235–248. DOI: http://dx.doi.org/10.1145/1095810.1095833

[33] Nguyen Anh Quynh. 2014. Capstone: Next-Gen Disassembly Framework. (2014).

[34] Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt B. Ferreira, Jon Stearley, John Shalf, and Sudhanva Gurumurthi. 2015. Memory Errors in Modern Systems: The Good, The Bad, and The Ugly. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, New York, NY, USA, 297–310. DOI: http://dx.doi.org/10.1145/2694344.2694348

[35] Margaret H. Wright and Al. 2010. The opportunities and challenges of exascale computing. (2010). https://science.energy.gov/~/media/ascr/ascac/pdf/reports/Exascale_subcommittee_report.pdf

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

For single process evaluation, we ran the benchmarks mentioned in the paper on a desktop equipped with AMD Ryzen 2700X and 16GB memory, and for evaluation of parallel jobs, we ran the benchmarks on a custom built cluster equipped with 64 haswell compute nodes, each with 48 cores and 128GB memory.

ARTIFACT AVAILABILITY

Software Artifact Availability: All author-created software artifacts are maintained in a public repository under an OSI-approved license.

Hardware Artifact Availability: There are no author-created hardware artifacts.

Data Artifact Availability: All author-created data artifacts are maintained in a public repository under an OSI-approved license.

Proprietary Artifacts: No author-created artifacts are proprietary.

List of URLs and/or DOIs where artifacts are available:

`https://github.com/kandycs/CARE.git`
`https://github.gatech.edu/cchen435/CAREExpr.git`

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: ChameleonCloud, Intel haswell, AMD Ryzen 2700x

Operating systems and versions: Ubuntu 16.04/Ubuntu 18.04

Compilers and versions: Clang 6.0.1

Applications and versions: miniMD, GTC-P, CoMD, HPCCG

Libraries and versions: openmpi-4.0.0, llvm-6.0.1, protobuf-3.6.0, udis86-1.7.2 capstone-4.0-rc, adios-1.13.1, libffi-3.2.1, libdwarf-20180809, pin-3.7-97619

Output from scripts that gathers execution environment information.

```
# For AMD Ryzen desktop
SUDO_GID=1000
MAIL=/var/mail/USER
LANGUAGE=en_US
USER=USER
HOME=/home/cchen
LC_CTYPE=en_US.UTF-8
COLORTERM=truecolor
SUDO_UID=1000
LOGNAME=USER
TERM=xterm-256color
USERNAME=USER
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/b⌋
↪ in:/sbin:/bin:/snap/bin
DISPLAY=:0
LANG=en_US.UTF-8
XAUTHORITY=/home/cchen/.Xauthority
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=0⌋
↪ 1;35:do=01;35:bd=40;33;01:cd=40;33;01:or=40;31;0⌋
↪ 1:mi=00:su=37;41:sg=30;43:ca=30;41:tw=30;42:ow=3⌋
↪ 4;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=01;31:*⌋
↪ .arc=01;31:*.arj=01;31:*.taz=01;31:*.lha=01;31:*⌋
↪ .lz4=01;31:*.lzh=01;31:*.lzma=01;31:*.tlz=01;31:⌋
↪ *.txz=01;31:*.tzo=01;31:*.t7z=01;31:*.zip=01;31:⌋
↪ *.z=01;31:*.Z=01;31:*.dz=01;31:*.gz=01;31:*.lrz=⌋
↪ 01;31:*.lz=01;31:*.lzo=01;31:*.xz=01;31:*.zst=01⌋
↪ ;31:*.tzst=01;31:*.bz2=01;31:*.bz=01;31:*.tbz=01⌋
↪ ;31:*.tbz2=01;31:*.tz=01;31:*.deb=01;31:*.rpm=01⌋
↪ ;31:*.jar=01;31:*.war=01;31:*.ear=01;31:*.sar=01⌋
↪ ;31:*.rar=01;31:*.alz=01;31:*.ace=01;31:*.zoo=01⌋
↪ ;31:*.cpio=01;31:*.7z=01;31:*.rz=01;31:*.cab=01;⌋
↪ 31:*.wim=01;31:*.swm=01;31:*.dwm=01;31:*.esd=01;⌋
↪ 31:*.jpg=01;35:*.jpeg=01;35:*.mjpg=01;35:*.mjpeg⌋
↪ =01;35:*.gif=01;35:*.bmp=01;35:*.pbm=01;35:*.pgm⌋
↪ =01;35:*.ppm=01;35:*.tga=01;35:*.xbm=01;35:*.xpm⌋
↪ =01;35:*.tif=01;35:*.tiff=01;35:*.png=01;35:*.sv⌋
↪ g=01;35:*.svgz=01;35:*.mng=01;35:*.pcx=01;35:*.m⌋
↪ ov=01;35:*.mpg=01;35:*.mpeg=01;35:*.m2v=01;35:*.⌋
↪ mkv=01;35:*.webm=01;35:*.ogm=01;35:*.mp4=01;35:*⌋
↪ .m4v=01;35:*.mp4v=01;35:*.vob=01;35:*.qt=01;35:*⌋
↪ .nuv=01;35:*.wmv=01;35:*.asf=01;35:*.rm=01;35:*.⌋
↪ rmvb=01;35:*.flc=01;35:*.avi=01;35:*.fli=01;35:*⌋
↪ .flv=01;35:*.gl=01;35:*.dl=01;35:*.xcf=01;35:*.x⌋
↪ wd=01;35:*.yuv=01;35:*.cgm=01;35:*.emf=01;35:*.o⌋
↪ gv=01;35:*.ogx=01;35:*.aac=00;36:*.au=00;36:*.fl⌋
↪ ac=00;36:*.m4a=00;36:*.mid=00;36:*.midi=00;36:*.⌋
↪ mka=00;36:*.mp3=00;36:*.mpc=00;36:*.ogg=00;36:*.⌋
↪ ra=00;36:*.wav=00;36:*.oga=00;36:*.opus=00;36:*.⌋
↪ spx=00;36:*.xspf=00;36:
SUDO_COMMAND=/bin/sh ./collect_environment.sh
SHELL=/usr/bin/zsh
SUDO_USER=cchen
PWD=/home/cchen/Documents/SC
+ lsb_release -a
No LSB modules are available.
Distributor ID:        Ubuntu
Description:        Ubuntu 18.04.2 LTS
Release:        18.04
Codename:        bionic
+ uname -a
Linux YZ 4.15.0-47-generic #50-Ubuntu SMP Wed Mar 13
↪ 10:44:52 UTC 2019 x86_64 x86_64 x86_64 GNU/Linux
+ lscpu
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              16
```

```
On-line CPU(s) list: 0-15
Thread(s) per core:  2
Core(s) per socket:  8
Socket(s):           1
NUMA node(s):        1
Vendor ID:           AuthenticAMD
CPU family:          23
Model:               8
Model name:          AMD Ryzen 7 2700X Eight-Core
↪  Processor
Stepping:            2
CPU MHz:             3170.957
CPU max MHz:         3700.0000
CPU min MHz:         2200.0000
BogoMIPS:            7385.39
Virtualization:      AMD-V
L1d cache:           32K
L1i cache:           64K
L2 cache:            512K
L3 cache:            8192K
NUMA node0 CPU(s):   0-15
Flags:               fpu vme de pse tsc msr pae mce
↪  cx8 apic sep mtrr pge mca cmov pat pse36 clflush
↪  mmx fxsr sse sse2 ht syscall nx mmxext fxsr_opt
↪  pdpe1gb rdtscp lm constant_tsc rep_good nopl
↪  nonstop_tsc cpuid extd_apicid aperfmperf pni
↪  pclmulqdq monitor ssse3 fma cx16 sse4_1 sse4_2
↪  movbe popcnt aes xsave avx f16c rdrand lahf_lm
↪  cmp_legacy svm extapic cr8_legacy abm sse4a
↪  misalignsse 3dnowprefetch osvw skinit wdt tce
↪  topoext perfctr_core perfctr_nb bpext perfctr_llc
↪  mwaitx cpb hw_pstate sme ssbd ibpb vmmcall
↪  fsgsbase bmi1 avx2 smep bmi2 rdseed adx smap
↪  clflushopt sha_ni xsaveopt xsavec xgetbv1 xsaves
↪  clzero irperf xsaveerptr arat npt lbrv svm_lock
↪  nrip_save tsc_scale vmcb_clean flushbyasid
↪  decodeassists pausefilter pfthreshold avic
↪  v_vmsave_vmload vgif overflow_recov succor smca
+ cat /proc/meminfo
MemTotal:       16418420 kB
MemFree:         1098732 kB
MemAvailable:   10380424 kB
Buffers:          728856 kB
Cached:          8138008 kB
SwapCached:        23968 kB
Active:         11272040 kB
Inactive:        2835476 kB
Active(anon):    4997620 kB
Inactive(anon):   308336 kB
Active(file):    6274420 kB
Inactive(file):  2527140 kB
Unevictable:          48 kB
Mlocked:              48 kB
SwapTotal:      49998844 kB
SwapFree:       49393916 kB
Dirty:              580 kB
Writeback:            0 kB
AnonPages:       5234516 kB
Mapped:           538280 kB
Shmem:             65292 kB
Slab:             927296 kB
SReclaimable:     816356 kB
SUnreclaim:       110940 kB
KernelStack:       20704 kB
PageTables:        89604 kB
NFS_Unstable:         0 kB
Bounce:               0 kB
WritebackTmp:         0 kB
CommitLimit:    58208052 kB
Committed_AS:   17245400 kB
VmallocTotal:   34359738367 kB
VmallocUsed:          0 kB
VmallocChunk:         0 kB
HardwareCorrupted:    0 kB
AnonHugePages:        0 kB
ShmemHugePages:       0 kB
ShmemPmdMapped:       0 kB
CmaTotal:             0 kB
CmaFree:              0 kB
HugePages_Total:      0
HugePages_Free:       0
HugePages_Rsvd:       0
HugePages_Surp:       0
Hugepagesize:       2048 kB
DirectMap4k:      686392 kB
DirectMap2M:    16019456 kB
DirectMap1G:     1048576 kB
+ inxi -F -c0
./collect_environment.sh: 14:
↪  ./collect_environment.sh: inxi: not found
+ lsblk -a
NAME        MAJ:MIN RM   SIZE RO TYPE MOUNTPOINT
loop0         7:0    0 138.1M  1 loop /snap/lepton/7
loop1         7:1    0  16.6M  1 loop
↪  /snap/ubuntu-budgie-welcome/124
loop2         7:2    0 138.9M  1 loop /snap/lepton/8
loop3         7:3    0         1 loop
loop4         7:4    0  91.1M  1 loop /snap/core/6531
loop5         7:5    0 174.5M  1 loop /snap/spotify/32
loop6         7:6    0  16.6M  1 loop
↪  /snap/ubuntu-budgie-welcome/120
loop7         7:7    0   174M  1 loop /snap/spotify/34
loop8         7:8    0 113.5M  1 loop /snap/lepton/1
loop9         7:9    0 108.9M  1 loop
↪  /snap/odrive-unofficial/2
loop10        7:10   0  16.6M  1 loop
↪  /snap/ubuntu-budgie-welcome/122
loop11        7:11   0  89.3M  1 loop /snap/core/6673
loop12        7:12   0  89.4M  1 loop /snap/core/6818
loop13        7:13   0 180.2M  1 loop /snap/spotify/35
sda           8:0    0 894.3G  0 disk
└─sda1        8:1    0 894.3G  0 part /home
```

```
nvme0n1     259:0    0 238.5G  0 disk
├─nvme0n1p1 259:1    0 190.8G  0 part /
├─nvme0n1p2 259:2    0     1K  0 part
└─nvme0n1p5 259:3    0  47.7G  0 part [SWAP]
+ lsscsi -s
./collect_environment.sh: 16:
↪  ./collect_environment.sh: lsscsi: not found
+ module list
./collect_environment.sh: 17:
↪  ./collect_environment.sh: module: not found
+ nvidia-smi
./collect_environment.sh: 18:
↪  ./collect_environment.sh: nvidia-smi: not found
+ lshw -short -quiet -sanitize
+ cat
H/W path              Device     Class
↪  Description
=====================================================┘
↪  =======
                                  system         To Be
                                  ↪  Filled By O.E.M.
                                  ↪  (To Be Filled By
                                  ↪  O.E.M.)
/0                                bus            X470
↪  Master SLI/ac
/0/0                           memory        64KiB BIOS
/0/10                             memory         16GiB
↪  System Memory
/0/10/0                           memory        [empty]
/0/10/1                           memory         8GiB
↪  DIMM DDR4 Synchronous Unbuffered (Unregistered)
↪  2666 MHz (0.4 ns)
/0/10/2                           memory        [empty]
/0/10/3                           memory         8GiB
↪  DIMM DDR4 Synchronous Unbuffered (Unregistered)
↪  2666 MHz (0.4 ns)
/0/12                             memory         768KiB
↪  L1 cache
/0/13                             memory        4MiB L2
↪  cache
/0/14                             memory         16MiB
↪  L3 cache
/0/15                             processor      AMD
↪  Ryzen 7 2700X Eight-Core Processor
/0/100                            bridge         Family
↪  17h (Models 00h-0fh) Root Complex
/0/100/0.2                        generic        Family
↪  17h (Models 00h-0fh) I/O Memory Management Unit
/0/100/1.3                        bridge         Family
↪  17h (Models 00h-0fh) PCIe GPP Bridge
/0/100/1.3/0                      bus
↪  Advanced Micro Devices, Inc. [AMD]
/0/100/1.3/0/0        usb1        bus            xHCI
↪  Host Controller
/0/100/1.3/0/0/9                  communication
↪  Bluetooth wireless interface
/0/100/1.3/0/1        usb2        bus            xHCI
↪  Host Controller
/0/100/1.3/0.1                    storage
↪  Advanced Micro Devices, Inc. [AMD]
/0/100/1.3/0.2                    bridge
↪  Advanced Micro Devices, Inc. [AMD]
/0/100/1.3/0.2/0                  bridge
↪  Advanced Micro Devices, Inc. [AMD]
/0/100/1.3/0.2/0/0    wlp30s0     network        Dual
↪  Band Wireless-AC 3168NGW [Stone Peak]
/0/100/1.3/0.2/1                  bridge
↪  Advanced Micro Devices, Inc. [AMD]
/0/100/1.3/0.2/1/0    enp31s0     network        I211
↪  Gigabit Network Connection
/0/100/1.3/0.2/2                  bridge
↪  Advanced Micro Devices, Inc. [AMD]
/0/100/1.3/0.2/3                  bridge
↪  Advanced Micro Devices, Inc. [AMD]
/0/100/1.3/0.2/4                  bridge
↪  Advanced Micro Devices, Inc. [AMD]
/0/100/1.3/0.2/4/0                storage
↪  Realtek Semiconductor Co., Ltd.
/0/100/1.3/0.2/6                  bridge
↪  Advanced Micro Devices, Inc. [AMD]
/0/100/1.3/0.2/7                  bridge
↪  Advanced Micro Devices, Inc. [AMD]
/0/100/3.1                        bridge         Family
↪  17h (Models 00h-0fh) PCIe GPP Bridge
/0/100/3.1/0                      display        GP106
↪  [GeForce GTX 1060 6GB]
/0/100/3.1/0.1                    multimedia     GP106
↪  High Definition Audio Controller
/0/100/7.1                        bridge         Family
↪  17h (Models 00h-0fh) Internal PCIe GPP Bridge 0 to
↪  Bus B
/0/100/7.1/0                      generic
↪  Advanced Micro Devices, Inc. [AMD]
/0/100/7.1/0.2                    generic        Family
↪  17h (Models 00h-0fh) Platform Security Processor
/0/100/7.1/0.3                    bus            USB 3.0
↪  Host controller
/0/100/7.1/0.3/0      usb3        bus            xHCI
↪  Host Controller
/0/100/7.1/0.3/0/2                bus
↪  HighSpeed Hub
/0/100/7.1/0.3/0/2/1              input          HHKB
↪  Professional
/0/100/7.1/0.3/0/2/3              input          USB
↪  Receiver
/0/100/7.1/0.3/0/4                input          USB
↪  Receiver
```

```
/0/100/7.1/0.3/1      usb4       bus            xHCI
↪  Host Controller
/0/100/8.1                       bridge         Family
↪  17h (Models 00h-0fh) Internal PCIe GPP Bridge 0 to
↪  Bus B
/0/100/8.1/0                     generic
↪  Advanced Micro Devices, Inc. [AMD]
/0/100/8.1/0.2                   storage        FCH
↪  SATA Controller [AHCI mode]
/0/100/8.1/0.3                   multimedia     Family
↪  17h (Models 00h-0fh) HD Audio Controller
/0/100/14                        bus            FCH
↪  SMBus Controller
/0/100/14.3                      bridge         FCH LPC
↪  Bridge
/0/101                           bridge         Family
↪  17h (Models 00h-0fh) PCIe Dummy Host Bridge
/0/102                           bridge         Family
↪  17h (Models 00h-0fh) PCIe Dummy Host Bridge
/0/103                           bridge         Family
↪  17h (Models 00h-0fh) PCIe Dummy Host Bridge
/0/104                           bridge         Family
↪  17h (Models 00h-0fh) PCIe Dummy Host Bridge
/0/105                           bridge         Family
↪  17h (Models 00h-0fh) PCIe Dummy Host Bridge
/0/106                           bridge         Family
↪  17h (Models 00h-0fh) PCIe Dummy Host Bridge
/0/107                           bridge         Family
↪  17h (Models 00h-0fh) Data Fabric: Device 18h;
↪  Function 0
/0/108                           bridge         Family
↪  17h (Models 00h-0fh) Data Fabric: Device 18h;
↪  Function 1
/0/109                           bridge         Family
↪  17h (Models 00h-0fh) Data Fabric: Device 18h;
↪  Function 2
/0/10a                           bridge         Family
↪  17h (Models 00h-0fh) Data Fabric: Device 18h;
↪  Function 3
/0/10b                           bridge         Family
↪  17h (Models 00h-0fh) Data Fabric: Device 18h;
↪  Function 4
/0/10c                           bridge         Family
↪  17h (Models 00h-0fh) Data Fabric: Device 18h;
↪  Function 5
/0/10d                           bridge         Family
↪  17h (Models 00h-0fh) Data Fabric: Device 18h;
↪  Function 6
/0/10e                           bridge         Family
↪  17h (Models 00h-0fh) Data Fabric: Device 18h;
↪  Function 7
/0/1                 scsi0       storage
/0/1/0.0.0           /dev/sda    disk           960GB
↪  ADATA SU650
/0/1/0.0.0/1         /dev/sda1   volume         894GiB
↪  EXT4 volume

# For Cluster
./colloect.sh: 1: ./colloect.sh: nux: not found
./colloect.sh: 1: ./colloect.sh: adjust: not found
SUDO_GID=1011
MAIL=/var/mail/USER
USER=USER
HOME=/home/cc
LC_CTYPE=en_US.UTF-8
SUDO_UID=1000
LOGNAME=USER
TERM=xterm-256color
USERNAME=USER
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/b
↪  in:/sbin:/bin:/snap/bin
LANG=en_US.UTF-8
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=0
↪  1;35:do=01;35:bd=40;33;01:cd=40;33;01:or=40;31;0
↪  1:mi=00:su=37;41:sg=30;43:ca=30;41:tw=30;42:ow=3
↪  4;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=01;31:*
↪  .arc=01;31:*.arj=01;31:*.taz=01;31:*.lha=01;31:*
↪  .lz4=01;31:*.lzh=01;31:*.lzma=01;31:*.tlz=01;31:
↪  *.txz=01;31:*.tzo=01;31:*.t7z=01;31:*.zip=01;31:
↪  *.z=01;31:*.Z=01;31:*.dz=01;31:*.gz=01;31:*.lrz=
↪  01;31:*.lz=01;31:*.lzo=01;31:*.xz=01;31:*.bz2=01
↪  ;31:*.bz=01;31:*.tbz=01;31:*.tbz2=01;31:*.tz=01;
↪  31:*.deb=01;31:*.rpm=01;31:*.jar=01;31:*.war=01;
↪  31:*.ear=01;31:*.sar=01;31:*.rar=01;31:*.alz=01;
↪  31:*.ace=01;31:*.zoo=01;31:*.cpio=01;31:*.7z=01;
↪  31:*.rz=01;31:*.cab=01;31:*.jpg=01;35:*.jpeg=01;
↪  35:*.gif=01;35:*.bmp=01;35:*.pbm=01;35:*.pgm=01;
↪  35:*.ppm=01;35:*.tga=01;35:*.xbm=01;35:*.xpm=01;
↪  35:*.tif=01;35:*.tiff=01;35:*.png=01;35:*.svg=01
↪  ;35:*.svgz=01;35:*.mng=01;35:*.pcx=01;35:*.mov=0
↪  1;35:*.mpg=01;35:*.mpeg=01;35:*.m2v=01;35:*.mkv=
↪  01;35:*.webm=01;35:*.ogm=01;35:*.mp4=01;35:*.m4v
↪  =01;35:*.mp4v=01;35:*.vob=01;35:*.qt=01;35:*.nuv
↪  =01;35:*.wmv=01;35:*.asf=01;35:*.rm=01;35:*.rmvb
↪  =01;35:*.flc=01;35:*.avi=01;35:*.fli=01;35:*.flv
↪  =01;35:*.gl=01;35:*.dl=01;35:*.xcf=01;35:*.xwd=0
↪  1;35:*.yuv=01;35:*.cgm=01;35:*.emf=01;35:*.ogv=0
↪  1;35:*.ogx=01;35:*.aac=00;36:*.au=00;36:*.flac=0
↪  0;36:*.m4a=00;36:*.mid=00;36:*.midi=00;36:*.mka=
↪  00;36:*.mp3=00;36:*.mpc=00;36:*.ogg=00;36:*.ra=0
↪  0;36:*.wav=00;36:*.oga=00;36:*.opus=00;36:*.spx=
↪  00;36:*.xspf=00;36:
SUDO_COMMAND=/bin/sh ./colloect.sh
SHELL=/bin/bash
SUDO_USER=cc
PWD=/home/cc
+ lsb_release -a
No LSB modules are available.
Distributor ID:        Ubuntu
Description:        Ubuntu 16.04.6 LTS
```

```
Release:        16.04
Codename:        xenial
+ uname -a
Linux fm 4.4.0-143-generic #169-Ubuntu SMP Thu Feb 7
↪  07:56:38 UTC 2019 x86_64 x86_64 x86_64 GNU/Linux
+ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                48
On-line CPU(s) list:   0-47
Thread(s) per core:    2
Core(s) per socket:    12
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 63
Model name:            Intel(R) Xeon(R) CPU E5-2670 v3
↪  @ 2.30GHz
Stepping:              2
CPU MHz:               1201.750
CPU max MHz:           3100.0000
CPU min MHz:           1200.0000
BogoMIPS:              4601.25
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              30720K
NUMA node0 CPU(s):     0,2,4,6,8,10,12,14,16,18,20,22」
↪  ,24,26,28,30,32,34,36,38,40,42,44,46
NUMA node1 CPU(s):     1,3,5,7,9,11,13,15,17,19,21,23」
↪  ,25,27,29,31,33,35,37,39,41,43,45,47
Flags:                 fpu vme de pse tsc msr pae mce
↪  cx8 apic sep mtrr pge mca cmov pat pse36 clflush
↪  dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx
↪  pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs
↪  bts rep_good nopl xtopology nonstop_tsc
↪  aperfmperf pni pclmulqdq dtes64 monitor ds_cpl
↪  vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid
↪  dca sse4_1 sse4_2 x2apic movbe popcnt
↪  tsc_deadline_timer aes xsave avx f16c rdrand
↪  lahf_lm abm epb invpcid_single ssbd ibrs ibpb
↪  stibp kaiser tpr_shadow vnmi flexpriority ept
↪  vpid fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms
↪  invpcid cqm xsaveopt cqm_llc cqm_occup_llc dtherm
↪  ida arat pln pts flush_l1d
+ cat /proc/meminfo
MemTotal:       131784524 kB
MemFree:        64850008 kB
MemAvailable:   84679732 kB
Buffers:          864084 kB
Cached:         19670556 kB
SwapCached:            0 kB
Active:         61611340 kB
Inactive:        2967620 kB
Active(anon):   44879004 kB
Inactive(anon):   562436 kB
Active(file):   16732336 kB
Inactive(file):  2405184 kB
Unevictable:        3652 kB
Mlocked:            3652 kB
SwapTotal:             0 kB
SwapFree:              0 kB
Dirty:               236 kB
Writeback:             0 kB
AnonPages:      44047552 kB
Mapped:           491952 kB
Shmem:           1394692 kB
Slab:            1578644 kB
SReclaimable:    1318284 kB
SUnreclaim:       260360 kB
KernelStack:       53664 kB
PageTables:       159380 kB
NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:          0 kB
CommitLimit:    65892260 kB
Committed_AS:   146129368 kB
VmallocTotal:   34359738367 kB
VmallocUsed:           0 kB
VmallocChunk:          0 kB
HardwareCorrupted:     0 kB
AnonHugePages:         0 kB
CmaTotal:              0 kB
CmaFree:               0 kB
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
DirectMap4k:      201424 kB
DirectMap2M:     3766272 kB
DirectMap1G:    132120576 kB
+ inxi -F -c0
./colloect.sh: 12: ./colloect.sh: inxi: not found
+ lsblk -a
NAME   MAJ:MIN RM   SIZE RO TYPE MOUNTPOINT
sda      8:0    0 232.9G  0 disk
└─sda1   8:1    0 232.9G  0 part /
sr0     11:0    1  1024M  0 rom
loop0    7:0    0         0 loop
loop1    7:1    0         0 loop
loop2    7:2    0         0 loop
loop3    7:3    0         0 loop
loop4    7:4    0         0 loop
loop5    7:5    0         0 loop
loop6    7:6    0         0 loop
loop7    7:7    0         0 loop
+ lsscsi -s
```

```
[0:0:0:0]    disk    ATA      ST9250610NS      AA65
↪  /dev/sda    250GB
[10:0:0:0]   cd/dvd  HL-DT-ST DVD+-RW GU90N    A3B0
↪  /dev/sr0        -
+ module list
./colloect.sh: 15: ./colloect.sh: module: not found
+ nvidia-smi
./colloect.sh: 16: ./colloect.sh: nvidia-smi: not
↪  found
+ lshw -short -quiet -sanitize
+ cat
H/W path             Device      Class
↪  Description
=================================================┘
↪  =======
                               system        PowerEdge
                                  ↪  R630 (SKU=NotPr┘
                                  ↪  ovided;ModelNam┘
                                  ↪  e=PowerEdge
                                  ↪  R630)
/0                                bus            0CNCJW
/0/0                             memory        64KiB BIOS
/0/400                            processor
↪  Intel(R) Xeon(R) CPU E5-2670 v3 @ 2.30GHz
/0/400/700                        memory         768KiB
↪  L1 cache
/0/400/701                       memory         3MiB L2
↪  cache
/0/400/702                        memory         30MiB
↪  L3 cache
/0/401                            processor
↪  Intel(R) Xeon(R) CPU E5-2670 v3 @ 2.30GHz
/0/401/703                        memory         768KiB
↪  L1 cache
/0/401/704                       memory         3MiB L2
↪  cache
/0/401/705                        memory         30MiB
↪  L3 cache
/0/1000                           memory         128GiB
↪  System Memory
/0/1000/0                         memory         16GiB
↪  DIMM Synchronous 2133 MHz (0.5 ns)
/0/1000/1                         memory         16GiB
↪  DIMM Synchronous 2133 MHz (0.5 ns)
/0/1000/2                         memory         16GiB
↪  DIMM Synchronous 2133 MHz (0.5 ns)
/0/1000/3                         memory         16GiB
↪  DIMM Synchronous 2133 MHz (0.5 ns)
/0/1000/4                        memory         [empty]
/0/1000/5                        memory         [empty]
/0/1000/6                        memory         [empty]
/0/1000/7                        memory         [empty]
/0/1000/8                        memory         [empty]
/0/1000/9                        memory         [empty]
/0/1000/a                        memory         [empty]
/0/1000/b                        memory         [empty]
/0/1000/c                         memory         16GiB
↪  DIMM Synchronous 2133 MHz (0.5 ns)
/0/1000/d                         memory         16GiB
↪  DIMM Synchronous 2133 MHz (0.5 ns)
/0/1000/e                         memory         16GiB
↪  DIMM Synchronous 2133 MHz (0.5 ns)
/0/1000/f                         memory         16GiB
↪  DIMM Synchronous 2133 MHz (0.5 ns)
/0/1000/10                       memory         [empty]
/0/1000/11                       memory         [empty]
/0/1000/12                       memory         [empty]
/0/1000/13                       memory         [empty]
/0/1000/14                       memory         [empty]
/0/1000/15                       memory         [empty]
/0/1000/16                       memory         [empty]
/0/1000/17                       memory         [empty]
/0/100                           bridge         Xeon E7
↪  v3/Xeon E5 v3/Core i7 DMI2
/0/100/1                         bridge         Xeon E7
↪  v3/Xeon E5 v3/Core i7 PCI Express Root Port 1
/0/100/1/0           scsi0       storage
↪  MegaRAID SAS-3 3008 [Fury]
/0/100/1/0/0.0.0     /dev/sda    disk           250GB
↪  ST9250610NS
/0/100/1/0/0.0.0/1   /dev/sda1   volume         232GiB
↪  EXT4 volume
/0/100/2                         bridge         Xeon E7
↪  v3/Xeon E5 v3/Core i7 PCI Express Root Port 2
/0/100/3                         bridge         Xeon E7
↪  v3/Xeon E5 v3/Core i7 PCI Express Root Port 3
/0/100/3/0           eno1        network
↪  NetXtreme II BCM57800 1/10 Gigabit Ethernet
/0/100/3/0.1         eno2        network
↪  NetXtreme II BCM57800 1/10 Gigabit Ethernet
/0/100/3/0.2         eno3        network
↪  NetXtreme II BCM57800 1/10 Gigabit Ethernet
/0/100/3/0.3         eno4        network
↪  NetXtreme II BCM57800 1/10 Gigabit Ethernet
/0/100/3.2                       bridge         Xeon E7
↪  v3/Xeon E5 v3/Core i7 PCI Express Root Port 3
/0/100/5                         generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Address Map, VTd_Misc,
↪  System Management
/0/100/5.1                       generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Hot Plug
/0/100/5.2                       generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 RAS, Control Status and
↪  Global Errors
/0/100/5.4                       generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 I/O APIC
/0/100/11                         generic
↪  C610/X99 series chipset SPSR
/0/100/11.4                     storage        C610/X99
↪  series chipset sSATA Controller [AHCI mode]
```

```
/0/100/16                       communication
↪  C610/X99 series chipset MEI Controller #1
/0/100/16.1                     communication
↪  C610/X99 series chipset MEI Controller #2
/0/100/1a                      bus           C610/X99
↪  series chipset USB Enhanced Host Controller #2
/0/100/1a/1          usb1       bus            EHCI
↪  Host Controller
/0/100/1a/1/1                   bus           USB hub
/0/100/1a/1/1/6                 bus            Gadget
↪  USB HUB
/0/100/1c                       bridge
↪  C610/X99 series chipset PCI Express Root Port #1
/0/100/1c.7                     bridge
↪  C610/X99 series chipset PCI Express Root Port #8
/0/100/1c.7/0                   bridge         SH7758
↪  PCIe Switch [PS]
/0/100/1c.7/0/0                 bridge         SH7758
↪  PCIe Switch [PS]
/0/100/1c.7/0/0/0               bridge         SH7758
↪  PCIe-PCI Bridge [PPB]
/0/100/1c.7/0/0/0/0             display        G200eR2
/0/100/1d                      bus           C610/X99
↪  series chipset USB Enhanced Host Controller #1
/0/100/1d/1          usb2       bus            EHCI
↪  Host Controller
/0/100/1d/1/1                   bus           USB hub
/0/100/1f                       bridge
↪  C610/X99 series chipset LPC Controller
/0/100/1f.2                    storage       C610/X99
↪  series chipset 6-Port SATA Controller [AHCI mode]
/0/2                            generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 QPI Link 0
/0/4                            generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 QPI Link 0
/0/6                            generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 QPI Link 0
/0/7                            generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 QPI Link 1
/0/8                            generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 QPI Link 1
/0/9                            generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 QPI Link 1
/0/a                            generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 R3 QPI Link 0 & 1 Monitoring
/0/b                            generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 R3 QPI Link 0 & 1 Monitoring
/0/c                            generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 R3 QPI Link 0 & 1 Monitoring
/0/d                            generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Unicast Registers
/0/e                            generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Unicast Registers
/0/f                            generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Unicast Registers
/0/10                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Unicast Registers
/0/11                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Unicast Registers
/0/12                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Unicast Registers
/0/13                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Unicast Registers
/0/14                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Unicast Registers
/0/15                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Unicast Registers
/0/16                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Unicast Registers
/0/17                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Unicast Registers
/0/18                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Unicast Registers
/0/19                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Buffered Ring Agent
/0/1a                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Buffered Ring Agent
/0/1b                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Buffered Ring Agent
/0/1c                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Buffered Ring Agent
/0/1d                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 System Address Decoder &
↪  Broadcast Registers
/0/1e                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 System Address Decoder &
↪  Broadcast Registers
/0/1f                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 System Address Decoder &
↪  Broadcast Registers
/0/20                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 PCIe Ring Interface
/0/21                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 PCIe Ring Interface
/0/22                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Scratchpad & Semaphore
↪  Registers
/0/23                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Scratchpad & Semaphore
↪  Registers
/0/24                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Scratchpad & Semaphore
↪  Registers
/0/25                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Home Agent 0
/0/26                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Home Agent 0
```

```
/0/27                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Home Agent 0 Debug
/0/28                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Home Agent 1
/0/29                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Home Agent 1
/0/2a                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Home Agent 1 Debug
/0/2b                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Integrated Memory Controller
↪  0 Target Address, Thermal & RAS Registers
/0/2c                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Integrated Memory Controller
↪  0 Target Address, Thermal & RAS Registers
/0/2d                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Integrated Memory
↪  Controller 0 Channel Target Address Decoder
/0/2e                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Integrated Memory
↪  Controller 0 Channel Target Address Decoder
/0/2f                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 DDRIO Channel 0/1 Broadcast
/0/30                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 DDRIO Global Broadcast
/0/31                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Integrated Memory
↪  Controller 0 Channel 0 Thermal Control
/0/32                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Integrated Memory
↪  Controller 0 Channel 1 Thermal Control
/0/33                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Integrated Memory
↪  Controller 0 Channel 0 ERROR Registers
/0/34                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Integrated Memory
↪  Controller 0 Channel 1 ERROR Registers
/0/35                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 DDRIO (VMSE) 0 & 1
/0/36                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 DDRIO (VMSE) 0 & 1
/0/37                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 DDRIO (VMSE) 0 & 1
/0/38                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 DDRIO (VMSE) 0 & 1
/0/39                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Integrated Memory Controller
↪  1 Target Address, Thermal & RAS Registers
/0/3a                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Integrated Memory Controller
↪  1 Target Address, Thermal & RAS Registers
/0/3b                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Integrated Memory
↪  Controller 1 Channel Target Address Decoder
/0/3c                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Integrated Memory
↪  Controller 1 Channel Target Address Decoder
/0/3d                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 DDRIO Channel 2/3 Broadcast
/0/3e                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 DDRIO Global Broadcast
/0/3f                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Integrated Memory
↪  Controller 1 Channel 0 Thermal Control
/0/40                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Integrated Memory
↪  Controller 1 Channel 1 Thermal Control
/0/41                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Integrated Memory
↪  Controller 1 Channel 0 ERROR Registers
/0/42                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Integrated Memory
↪  Controller 1 Channel 1 ERROR Registers
/0/43                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 DDRIO (VMSE) 2 & 3
/0/44                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 DDRIO (VMSE) 2 & 3
/0/45                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 DDRIO (VMSE) 2 & 3
/0/46                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 DDRIO (VMSE) 2 & 3
/0/47                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Power Control Unit
/0/48                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Power Control Unit
/0/49                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Power Control Unit
/0/4a                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Power Control Unit
/0/4b                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Power Control Unit
/0/4c                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 VCU
/0/4d                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 VCU
/0/1                            bridge         Xeon E7
↪  v3/Xeon E5 v3/Core i7 PCI Express Root Port 1
/0/3                            bridge         Xeon E7
↪  v3/Xeon E5 v3/Core i7 PCI Express Root Port 3
/0/5                            generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Address Map, VTd_Misc,
↪  System Management
/0/5.1                          generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Hot Plug
/0/5.2                          generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 RAS, Control Status and
↪  Global Errors
/0/5.4                          generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 I/O APIC
/0/4e                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 QPI Link 0
```

```
/0/4f                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 QPI Link 0
/0/50                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 QPI Link 0
/0/51                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 QPI Link 1
/0/52                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 QPI Link 1
/0/53                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 QPI Link 1
/0/54                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 R3 QPI Link 0 & 1 Monitoring
/0/55                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 R3 QPI Link 0 & 1 Monitoring
/0/56                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 R3 QPI Link 0 & 1 Monitoring
/0/57                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Unicast Registers
/0/58                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Unicast Registers
/0/59                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Unicast Registers
/0/5a                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Unicast Registers
/0/5b                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Unicast Registers
/0/5c                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Unicast Registers
/0/5d                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Unicast Registers
/0/5e                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Unicast Registers
/0/5f                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Unicast Registers
/0/60                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Unicast Registers
/0/61                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Unicast Registers
/0/62                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Unicast Registers
/0/63                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Buffered Ring Agent
/0/64                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Buffered Ring Agent
/0/65                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Buffered Ring Agent
/0/66                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Buffered Ring Agent
/0/67                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 System Address Decoder &
↪  Broadcast Registers
/0/68                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 System Address Decoder &
↪  Broadcast Registers
/0/69                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 System Address Decoder &
↪  Broadcast Registers
/0/6a                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 PCIe Ring Interface
/0/6b                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 PCIe Ring Interface
/0/6c                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Scratchpad & Semaphore
↪  Registers
/0/6d                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Scratchpad & Semaphore
↪  Registers
/0/6e                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Scratchpad & Semaphore
↪  Registers
/0/6f                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Home Agent 0
/0/70                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Home Agent 0
/0/71                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Home Agent 0 Debug
/0/72                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Home Agent 1
/0/73                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Home Agent 1
/0/74                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Home Agent 1 Debug
/0/75                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Integrated Memory Controller
↪  0 Target Address, Thermal & RAS Registers
/0/76                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Integrated Memory Controller
↪  0 Target Address, Thermal & RAS Registers
/0/77                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Integrated Memory
↪  Controller 0 Channel Target Address Decoder
/0/78                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Integrated Memory
↪  Controller 0 Channel Target Address Decoder
/0/79                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 DDRIO Channel 0/1 Broadcast
/0/7a                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 DDRIO Global Broadcast
/0/7b                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Integrated Memory
↪  Controller 0 Channel 0 Thermal Control
/0/7c                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Integrated Memory
↪  Controller 0 Channel 1 Thermal Control
/0/7d                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Integrated Memory
↪  Controller 0 Channel 0 ERROR Registers
/0/7e                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Integrated Memory
↪  Controller 0 Channel 1 ERROR Registers
```

```
/0/7f                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 DDRIO (VMSE) 0 & 1
/0/80                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 DDRIO (VMSE) 0 & 1
/0/81                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 DDRIO (VMSE) 0 & 1
/0/82                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 DDRIO (VMSE) 0 & 1
/0/83                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Integrated Memory Controller
↪  1 Target Address, Thermal & RAS Registers
/0/84                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Integrated Memory Controller
↪  1 Target Address, Thermal & RAS Registers
/0/85                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Integrated Memory
↪  Controller 1 Channel Target Address Decoder
/0/86                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Integrated Memory
↪  Controller 1 Channel Target Address Decoder
/0/87                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 DDRIO Channel 2/3 Broadcast
/0/88                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 DDRIO Global Broadcast
/0/89                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Integrated Memory
↪  Controller 1 Channel 0 Thermal Control
/0/8a                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Integrated Memory
↪  Controller 1 Channel 1 Thermal Control
/0/8b                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Integrated Memory
↪  Controller 1 Channel 0 ERROR Registers
/0/8c                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Integrated Memory
↪  Controller 1 Channel 1 ERROR Registers
/0/8d                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 DDRIO (VMSE) 2 & 3
/0/8e                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 DDRIO (VMSE) 2 & 3
/0/8f                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 DDRIO (VMSE) 2 & 3
/0/90                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 DDRIO (VMSE) 2 & 3
/0/91                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Power Control Unit
/0/92                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Power Control Unit
/0/93                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Power Control Unit
/0/94                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Power Control Unit
/0/95                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 Power Control Unit
/0/96                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 VCU
/0/97                           generic        Xeon E7
↪  v3/Xeon E5 v3/Core i7 VCU
/0/98                scsi10     storage
/0/98/0.0.0          /dev/cdrom disk
↪  DVD+-RW GU90N
/1                              power          0G6W6KA00
/2                              power          0G6W6KA00
```

ARTIFACT EVALUATION

Accuracy and precision of timings: average numbers were taken for timing measurements

Quantified the sensitivity of results to initial conditions and/or parameters of the computational environment: could be sensitive