BonVoision: Leveraging Spatial Data Smoothness for Recovery from Memory Soft Errors

Bo Fang, Hassan Halawa, Karthik Pattabiraman, Matei Ripeanu, Sriram Krishnamoorthy*
Electrical and Computer Engineering, University of British Columbia
*High Performance Computing Group, Pacific Northwest National Laboratory
{bof,hhalawa, karthikp, matei}@ece.ubc.ca, {sriram}@pnnl.gov

Abstract

Detectable but Uncorrectable Errors (DUEs) in the memory subsystem are becoming increasingly frequent. Today, upon encountering a DUE, applications crash, and the recovery methods used incur significant performance, storage, and energy overheads. To mitigate the impact of these errors, we start from two high-level observations that apply to some classes of HPC applications (e.g., stencil computations on regular grids or irregular meshes): first, these applications, display a property we dub *spatial data smoothness*: i.e., data items that are nearby in the application's logical space are relatively similar. Second, since these data items are generally used together, programmers go to great lengths to place them in nearby memory locations to improve application's performance by improving access locality. Based on these observations we explore the feasibility of a roll-forward recovery scheme that leverages spatial data smoothness to repair the memory location corrupted by a DUE and continues the application execution. We present BonVoision, a run-time system that intercepts DUE events, analyzes the application binary at runtime to identify the data elements in the neighborhood of the memory location that generates a DUE, and uses them to fix the corrupted data. Our evaluation demonstrates that BonVoision is: *(i) efficient* - it incurs negligible overhead, *(ii) effective* - it is frequently successful in continuing the application with benign outcomes, and *(iii) user friendly* - as it does not require programmer input to expose the data layout or access to source code. We demonstrate that using BonVoision can lead to significant savings in the context of a checkpointing/restart schemes by enabling longer checkpoint intervals.

1 Introduction

Shrinking feature sizes, reduced supply voltages, lower refresh rates, and the overarching need to reduce the energy footprint, make memory devices subject to increasing error rates [1, 2]. This scenario is exacerbated in a HPC context, where a large application memory footprint and system scale make it more likely that the system experiences memory errors. For example, Martino et al. [38] report that on Blue Waters, on average 250 memory errors per hour are observed across half of the total Blue Waters nodes. A recent study [51] predicts that the uncorrected error rate for a future exascale system could be 3.6–69.9× the uncorrected error rate currently observed in existing supercomputers.

The most common kind of error correcting codes (ECC) used to protect against these errors are single-error correction double-error detection (SECDED): they correct single-bit flips, but can only detect (and not correct) double-bit flips. More advanced ECC schemes such as chipkill can extend the capacity of the correctable bits in memory [15, 34], but they generally incur high cost in terms of energy and performance. Therefore, a portion of the memory errors remain detected but not corrected. These Detectable but Uncorrectable Errors are known as DUEs. While DUEs occur less frequently than correctable errors (e.g., Martino et al. [38] observe that 29.8% of memory errors are DUEs without chipkill, while with chipkill this percentage drops to less than 1%), they can lead to critical consequences for long-running applications. when a DUE occurs, the CPU raises a machine check exception (MCE) to the Operating System (OS), which typically crashes the application. Higher crash likelihood directly impacts time-to-solution and the cost of mitigating strategies (e.g., checkpoint frequency).

Prior studies [3, 39, 55] have observed that many applications inherently mask some errors, and still produce acceptable results. This observation motivated us to first investigate whether the fail-stop model overprotects the application in the context of HPC applications. We asked whether *is it feasible to simply ignore DUEs* and continue operating. Unfortunately, our fault injection experiments (Section 3.1) revealed that while not all DUEs lead to failures, the rate of subsequent failures is too high for this technique to be of practical use. Therefore, we develop heuristics to repair the state affected by the DUE and enable forward application progress.

The key insight that guides the design of our repair mechanism is that some classes of HPC applications (e.g., stencil applications) are likely to have spatial data smoothness [4]. This is because the physical phenomena often modeled by HPC applications exhibit inherent continuities in their modeled physical space e.g., temperature, pressure, or velocity for a climate modelling application. Further, the data structures used by these applications tend to keep physically close data points close together in memory as well to exploit locality. We leverage these two observations to propose a repair scheme for DUEs in HPC applications that is: *(i) effective* - it is frequently successful in continuing the application with benign outcomes, *(ii) efficient* - it incurs negligible performance overhead, *(iii) automated* - it does not require additional programmer involvement to expose the data layout, and *(iv) practical* as it neither requires access to the application's source code, nor runtime instrumentation.

While conceptually elegant, there are two challenges in designing a repair technique based on the above intuition and offering the above characteristics. The first challenge is to identify the application-level data *element* corresponding to an observed DUE

ICS '19, June 26–28,2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6079-1/19/06…$15.00
https://doi.org/10.1145/3330345.3330388

(recall that the ECC memory would only identify the faulty memory *word* in case of a DUE), as HPC applications are deployed as binaries on the target system, and the source code is not available at runtime when faults occur. Even if the source code was available, it is not straightforward to infer the data layout and the nearby data elements in the application's physical space without relying on either programmer annotations (which would imply a significant burden), or on runtime instrumentation (which would slow down the execution). After identifying the application-level data element corresponding to the fault, the second challenge is to decide on the value to use for repair by finding uncorrupted elements that are spatially close to the location of the fault.

We present BonVoision, a lightweight and automated approach for DUE repair. BonVoision requires neither programmer annotations nor instrumentation of the application code. To address the first challenge, we propose a set of heuristics to infer the neighboring application-level data elements of a faulty word, based on runtime analysis of the application's binary. To address the second challenge, we propose a number of repair heuristics, and experimentally evaluate them.

BonVoision can be used in conjunction with classic checkpoint-restart (C/R) schemes typically deployed on HPC systems. On a DUE, a C/R system rolls-back the application and recovers it from a previously collected checkpoint. With BonVoision, the system will first attempt to reconstruct the application state and roll-forward (i.e., continue execution). Only in case of an another failure (either a crash or incorrect state detected by application-level checks), will it attempt to recover from the checkpoint. From the perspective of a C/R scheme, the impact of using BonVoision is similar to increasing the mean time between failures (MTBFs), leading to lower checkpointing overheads and faster execution times.

This paper makes the following contributions:

- Introduces BonVoision, a software methodology that leverages data spatial similarity to repair memory DUEs. (Sections 3 and 4)
- Evaluates and compares the end-to-end application outcomes for three repair schemes: *(i)* using BonVoision's stride-based values, *(ii)* 0s, and *(iii)* random values. Results show that BonVoision repairs lead to 0.8 - 2.5x more benign outcomes, and only marginal increases in Silent Data Corruption (SDC) outcomes compared to using 0 values and random values for the repairs. (Section 6.1)
- Proposes an enhancement to BonVoision to use an online classifier, to predict the outcome of a individual repair. We demonstrate that the optimized BonVoision (called BonVoision-E) leads to a significant improvement in repair effectiveness (on average 23% higher rate of benign-only outcomes). (Section 6.2)
- Evaluates the overall BonVoision efficiency with different program input sizes and number of MPI processes. We find that BonVoision incurs a negligible overhead (about 6 milliseconds on average per corrected DUE). (Section 6.3)
- Demonstrates the performance gains enabled by BonVoision-E when used in the context of a C/R scheme. (Section 6.4)

2 Background

ECC Memory. Dynamic random-access memory (DRAM) may incur single- or multi-bit flips due to, for example, alpha particle strikes, background radiation, or neutrons from cosmic rays [43]. ECC uses extra memory bits to detect and correct bit-flip errors. SECDED, the most widely-used ECC scheme, uses 8 redundant bits for every 64 bits. More powerful schemes such as chipkill ECCs can correct more errors at a higher cost. While generally applicable, we demonstrate BonVoision in the context of the SECDED scheme. A memory location is checked for an error when an application reads it or during memory scrubbing, which periodically performs memory reads and writes. The memory controller computes the checksum for the data on every read, and compares it with the stored ECC bits. For SECDED, when more than one bit is flipped, the controller logs the event as a DUE and passes it to the processor.

MCE for memory DUEs. On x86 processors, the most common mechanism to deal with a memory DUE is to generate a machine check exception (MCE). In the case of DUEs, MCE encapsulates the DUE-specific hardware information (e.g., socket, channel, DIMM, etc). In some scenarios, the computed ECC syndrome and the value of the corrupted data are available, but we do not assume that this information is available. MCEs are generally directly handled by the OS. For example, Linux provides an "MCE tolerance level" option for users to determine how the OS should react to MCEs. By default the kernel would either "panic" or deliver a SIGBUS exception to the applications on uncorrected errors, and log the corrected errors. We assume that only the application experiencing the DUE receives the SIGBUS and terminates (this covers most cases). In user space, mcelog [29] is used for Linux to decode and consolidate MCE messages.

Checkpoint/Restart. C/R schemes [18, 54] are frequently used to recover from crash (i.e., fail-stop) failures. A C/R scheme consists of two main operations: *(i)* it periodically saves program state to non-volatile storage (i.e., it 'checkpoints' the data necessary for the application to recover); *(ii)* if a crash failure occurs, it retrieves one of the checkpoints (often the latest one) to re-create the current state, and continues the application. Many studies have focused on determining the "optimal" check-pointing interval, which optimizes application's expected runtime. Young et al. [59] and Daly et al. [14] present analytic models to find the optimal checkpoint interval as a function of *(i)* the time to generate the checkpoint (i.e., the checkpoint overhead), and *(ii)* the mean time between failure (MTBF). EI-Sayed et al. [16] empirically show that checkpointing guided by Young's formula achieves almost identical performance as more sophisticated schemes, based on exhaustive observations on production systems. BonVoision essentially attempts to convert the fail-stop DUE failures into continued and successful executions. Since the application fails less often, its MTBF is increased, reducing checkpointing overheads when BonVoision is used.

3 Motivating Studies

Our approach to repair memory soft errors that result in DUEs is based on two observations: *(i)* simply ignoring the DUEs does not work for HPC applications, and *(ii)* some classes of HPC applications exhibit spatial data smoothness that can be leveraged for efficient DUE repairs. We describe the experiments based on which we make the two observations. The benchmarks used and the other experimental details are described in Section 5.

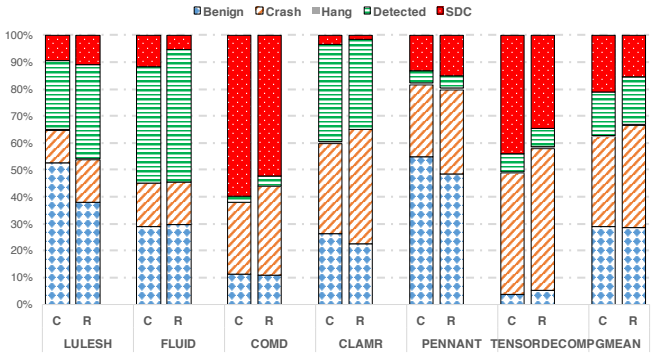


Figure 1: Error injection results for six HPC applications. C means consecutive bits, while R means random bits.

3.1 Does Ignoring DUEs Work?

A typical application's virtual memory space is organized as segments such as stack, heap, etc. Typically, the heap segments consumes the largest size of the virtual memory space, and hence if a DUE occurs in an application's memory, it most likely occurs in the heap segment. To verify this, we have profiled the use of stack and heap segments for the HPC applications in our benchmark and found that the heap segments indeed dominate the memory usage.

We inject DUE errors into the heap segment, and observe their impact on application outcomes. We perform the following steps in the error injection campaign: *(i)* we profile each application to find out how many dynamic memory read instructions access the heap segment – these are our *fault injection sites*. We pick memory read instructions only as they are the sole sources that trigger DUEs considered in this paper [1]; *(ii)* we randomly (uniform probability distribution) select an instruction from the set of all error injection sites, and flip[2] two bits in the memory data accessed by the selected instruction before the instruction executes. We consider two error modes: the flipped bits are consecutive or at random; and *(iii)* we allow the program to complete execution and study its outcome.

The outcomes are classified into five categories: (1) benign: correct output, identical to that of a 'golden run' - a run with no fault injected, (2) detected: the application's own correctness checks detects some violation of the properties in the results (see section Section 5 for details on correctness checks), (3) Silent Data Corruption (SDC): the output data is different from that of the 'golden run', (4) crash: exceptions, and (5) hang: the program does not stop for a long period of time. We perform 5,000 double-bit fault injection runs for each application to obtain a statistically significant estimate of outcome probabilities: the 95% confidence interval yields error bars of around 1%.

Figure 1 shows the results of the error injection experiments. On average, when injecting into consecutive bits (left bars), 30.5% of the injected errors lead to benigns, 26.4% to crashes, 19.3% to detected cases, 0.3% to hangs and 23.5% to SDCs. The errors injected into random bits (right bars) cause slightly fewer benigns (25.4%) and SDCs (20%) but higher crashes (32.2%) and detected cases (22%). Overall, the results are independent of how we choose the 2 bits. We also observe that not a single benchmark exhibits a high benign rate combined with a low SDC rate. Thus it is not practical for these applications to just ignore memory DUEs.

[1]Memory scrubbing is likely to occur during idle period to prevent decreasing performance, so we do not consider it as the source of the DUEs in this study

[2]We modify PINFI [56], a PIN-based instruction-level fault injection tool, to inject 2-bit-flip errors into memory on the memory instructions that read from the heap.

3.2 Exploring Spatial Data Smoothness

This paper makes two assumptions about smoothness : *(i)* that it exists in the application's logical space; and *(ii)* that data structures preserve it when mapped to actual memory. To verify these assumptions together, we quantitatively estimate the spatial smoothness of an application's core data structures (i.e., the data structures that model the physical space the application works over) once they are mapped to memory. If spatial smoothness exists, then the standard deviation of the differences between consecutive elements in the memory layout of the data structure is small. Therefore, given a data structure and a sets of elements S, we form a new set $diff(S) = \{ d_i | d_i = s_{i+1} - s_i, \forall s_i \in S \mid \}$ by first computing the difference between successive elements, and then computing the standard deviation over $diff(S)$ - the smaller the standard deviation, the smoother is the space. To study how spatially far from an anchor element smoothness extends, we vary the size of the set S. More concretely, we pick in S the data elements at distance (in stride) $\pm 2^i$ for $i = 1..K$ around an anchor element.

To estimate smoothness for our benchmark applications, we build a custom profiler that pauses the application execution at multiple random time points and, at each pause, randomly selects a number of anchor elements from the core data structures of the application. It then constructs the set S and applies the metric described above. Note that since the profiler works directly at the data structures, it makes no mistake in finding the neighbouring elements of that data structure. In our experimental settings, the profiler pauses each application 10 times and picks 1,000 random anchor elements during each pause.

Figure 2 shows the empirical cumulative distribution function (ECDF) of standard deviations (SD) under different step sizes (values for K above). We present the ECDFs sampled from one of the core data structures of each application as an example. We make two main observations: *(i) generally the values for SDs are low, indicating smoothness.* Around 80% to 100% SDs are close to 0 in LULESH, FLUID and PENNANT; 80% of SDs are less than 0.3 in TENSORDECOMP, and nearly 100% of SDs are less than 0.5 in CLAMR; for COMD, around 75% of SDs are close to 0, but the remaining 25% vary in a large range. *(ii) comparing the SDs across different step sizes, step size 1 always offers the smallest SDs for all applications.* This indicates that BonVoision should use the closest neighbours for repair. In summary, we observe that significant spatial data smoothness exists across the elements of the core data structures.

4 Design and Implementation of BonVoision

Overview. Section 3.2 shows that application-level data structures exhibit spatial data smoothness. Armed with this information, we explore DUE repair strategies that harness smoothness and use the data values from nearby locations to propose values used for repair. BonVoision embodies the results of this exploration.

BonVoision consists of three main components (Figure 3): *(i)* information parsing, *(ii)* stride speculation, and *(iii)* repair write-back. BonVoision is invoked upon a DUE signal. The information parsing component parses the information from the DUE signal

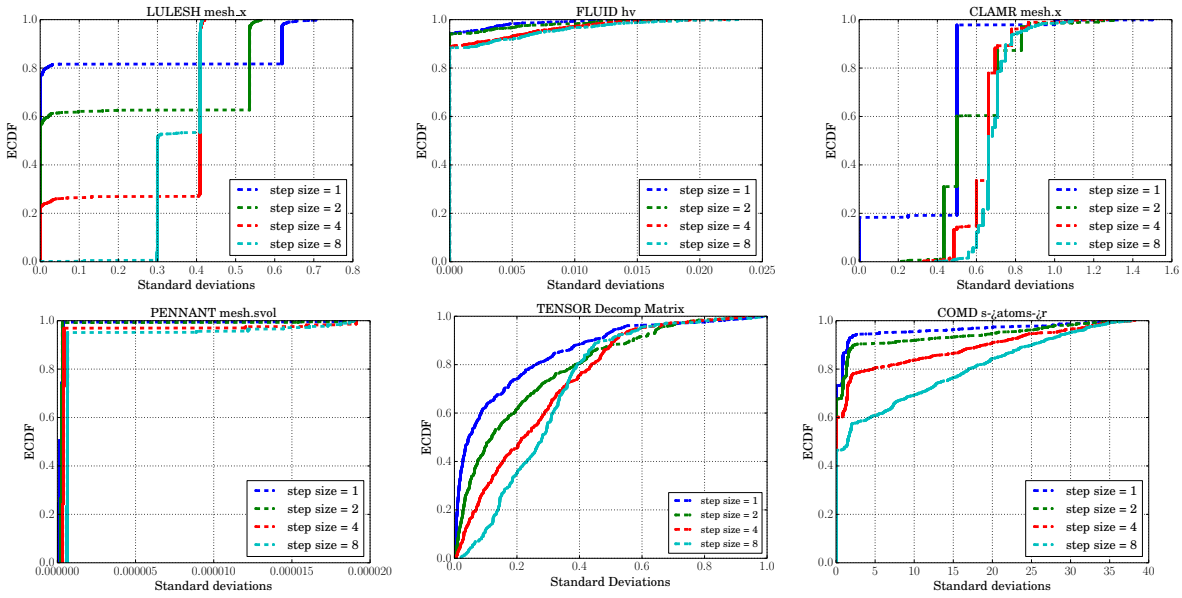


Figure 2: Examples of the ECDF of the standard deviations for six benchmarks (LULESH, FLUID, CLAMR (top-row), PENNANT, TENSORDECOMP, and COMD (bottom row). The corresponding data structures are indicated on the top of the figures.

handler and feeds it to the stride speculation component. This component runs a set of heuristics and outputs the possible stride. Finally, the repair write-back component uses the stride to access the neighboring elements, computes their average value, and writes that value back to the corrupted memory location. The rest of this section focuses mainly on the stride speculation component, as it is the most challenging part. We first explain the challenges of stride speculation, followed by a synthetic study we designed to devise heuristics. Finally, we present the heuristics for stride speculation.

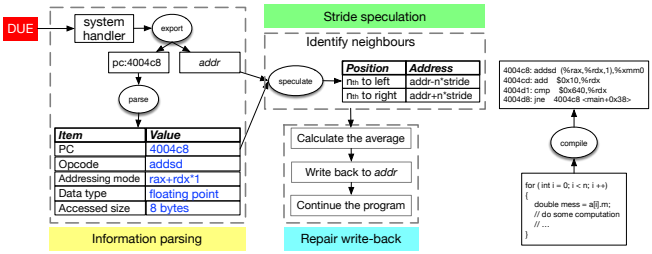


Figure 3: Illustration of BonVoision components, the information available upon a DUE and why this information is insufficient to determine the a data structure's stride.

Stride speculation challenges. We explain the challenge for stride speculation with an example: the program iterates over the array and reads its elements for the later computation. Figure 3 shows the C code and corresponding x86 instructions (compiled with gcc and -O3). Suppose a memory DUE occurs when the memory read instruction (*4004c8*) is executed, and a recovery handler is in place. For the recovery strategy to exploit data smoothness and repair the corrupted location using information from nearby elements, it needs to identify the neighbours of the data residing in *addr*. At the source code level, this task is trivial: knowing the address of an element &a[k].m = *addr*, its neighbours should be addressed as &a[k-1].m = *addr* - sizeof(a[k]) and &a[k+1].m = *addr* + sizeof(a[k]), where in our example the sizeof(a[k]) = 2*sizeof(double). However, it is challenging to infer this at the assembly code level, with no support from the programmer or other program instrumentation. This is because the only information available to the recovery handler is the assembly instruction itself (referred to as *pc* in Figure 3) and the corrupted memory location *addr*. We need to speculate the stride based on this information.

4.1 Synthetic Study

We conduct a synthetic study that explores several typical scenarios to find common patterns for inferring the stride.

Addressing modes. The addressing modes in x86 assembly consist of several segments including base, index, displacement and scale, where the base and index are x86 registers, and displacement and scale are immediate numbers (scale can be 1,2,4 or 8). Depending on the compilation options, program semantics and other factors, the compiler needs to choose a particular addressing mode for accessing the elements within a data structure. For example, the instruction *4004c8* in Figure 3 contains no displacement segment, while other memory read instructions may contain no index or scale segments, no base segments, etc.

Observations. We made a preliminary observation on the assembly code in Figure 3: the register *rdx* is the index register of instruction *4004c8*, and gets added by 0x10 (which is the size of the data structure *Atom*) every time the instruction *4004cd* gets executed. This observation leads to two insights: 1) there is an observable update on the instruction's register to locate different elements in the data structure - we call this register the **stride register**; 2) the value of the stride can be inferred based on the modification to the stride register. Therefore, it is possible to infer the stride from the corresponding address calculation operations on the memory instruction.

For a more comprehensive understanding of how a program calculates the addresses for different elements in a data structure, we study several typical data structures in C/C++: vector, array, array

Table 1: The patterns and stride observed for accessing to the elements of the different data structures. Reading elements from Foo3 and Foo5 results in the same addressing mode as Foo2.

Data structure	Compilation option	Accessed elements	Addressing mode	Register for address shifting	Instruction(s) modify the register	Stride
Foo1	O3	bar2	(base)	rdx	add $(0x8), %rdx	8
Foo2	O3	bar3	displacement(base)	rdx	add $(0x10), %rdx	16
Foo4	O3	bar1.bar2	displacement(base,index,scale)	rdx*4 (scale)	add $0x1, %rdx	4
Foo4	O0	bar1.bar2	displacement(base)	rax, rdx	shl $0x4, %rdx add $rdx, %rax	16

of structures, structure of arrays, namely foo: vector<double>; foo1: {int bar1; double bar2}; foo2:{float bar1; float bar2;}; foo3:{foo2 *bar1; int bar2;}; foo4:{foo2 bar1[n]; int bar2[n];}; [3]. We implement a simple program that consecutively accesses elements of each data structure, and examine the program's assembly code for the instructions associated with the address calculations. Table 1 summaries our findings. There are two main insights: *(i)* both the base and the index registers can participate in the element's address calculations, *(ii)* while the compilation options (i.e., O0 and O3) generate distinctive addressing modes, the stride information can still be inferred from the corresponding address calculation instructions.

4.2 Heuristics

Based on the insights from the above study, we design 2 heuristics to speculate the stride information from the program's assembly code.

Basic heuristic. We observed that many core data structures are comprised of basic-type elements such as double, float, integer etc. For those data structures, the stride information can be directly inferred by the accessed size from the memory read instruction. We call this heuristic the size-based approach.

Advanced heuristic. For more complicated cases, we leverage the insight from the above study, that the stride information may be found by tracing the modification to the stride register, i.e., either the base or the index register used for accessing different elements in a data structure. Thus, we propose the following heuristic to look for the instruction/instructions that write to the stride register, and extract the value in the operands as the indicator of how consecutive elements are accessed. The heuristic contains two main processes:

Backward and forward slicing. The goal of this process is to traverse all the instructions that use the stride register as the destination register at the assembly level, and find which one(s) contain the information of accessing to the neighbouring elements. As observed in the the synthetic study, these instructions can be either before or after the memory instruction, so both directions need to be searched. In practice, when a instruction modifies the register with an intermediate number, or a *lea* (load effective address) instruction modifies the register, the stride is inferred from the instruction(s), and the process terminates. The process also terminates when all the static instructions in the same function are iterated, or there is a memory read instruction that overwrites the register with the memory data. In these cases, BonVoision shifts to the basic heuristic and uses the accessed size as the stride.

Handling transitive relations. There are cases where the data dependency propagates to another register: when the backward or the forward slicing process identify it as (i.e. the transitive register) being used to update the root register of the current slices, the modification on the new transitive register needs to be considered. When such relation is captured, the current slice needs to be discarded, and new backward and forward slicing processes based on the transitive register are launched. It might be possible for BonVoision to encounter one or more levels of transitive relations. Therefore, BonVoision sets the limit to the depth of the transitivity, based on the observations on various applications' assembly codes.

Algorithm 2 shows how the two heuristics are implemented in BonVoision. It takes the instruction that triggers the DUE as the input, determines its addressing mode, and calls the backward and forward slicing procedures respectively. Algorithm 1 describes how the backward slice is computed. The forward slicing process is identical to the backward slicing process except that the $prev(inst\_b)$ needs to be replaced with $next(inst\_b)$.

Algorithm 1 Backward instruction slicing

```
1:  procedure BACKWARD_SLICING(reg, inst)
2:      inst_b ← inst
3:      while inst_b ≠ first inst in the function do
4:          ▷ the function contains the inst
5:          inst_b ← prev(inst_b)
6:          if reg == dest_reg(inst_b) then
7:              ▷ dest_reg outputs the destination register of a inst
8:              if has_immediate(inst_b)or is_lea(inst_b) then
9:                  if is_lea(inst_b) then
10:                     scale ← parse_lea(inst_b)
11:                     if scale == 0||scale == 1 then
12:                         index_lea = parse(inst_b)
13:                         ▷ goes one more level of transitivity
14:                         backward_slicing(index_lea)
15:                         forward_slicing(index_lea)
16:                     else
17:                         stride ← scale
18:                 else
19:                     stride = handle_immediate(inst_b)
20:             else
21:                 ▷ Transitive relation begins
22:                 backward_slicing(source_reg(inst_b))
23:                 forward_slicing(source_reg(inst_b))
24:     return stride
```

Algorithm 2 The main procedure for stride speculation

```
1:  procedure SPECULATE(inst)                          ▷ the inst that triggers the DUE
2:      stride ← 0
3:      (base, index, size) ← parse(inst)
4:      ▷ size means the accessed size
5:      if index ≠ "" then                             ▷ the addressing mode includes index
6:          stride ← backward_slicing(base, inst)
7:          if stride == 0 then
8:              stride ← forward_slicing(base, inst)
9:      else
10:         stride ← backward_slicing(index, inst)
11:         if stride == 0 then
12:             stride ← forward_slicing(index, inst)
13:     if stride == 0 then
14:         stride ← size                              ▷ takes the basic heuristic
15:     return stride
```

Impact on the run-time overhead. Prior studies [13, 22] show that conducting backward/forward slicing processes requires building dynamic data dependence graphs, which could be extremely

[3]The allocation, initialization, access, and free operations, implemented in the driver code, are not shown here due to space limits.

Table 2: Benchmark description, including SDC determination approach and application correctness check criteria

Application	Domain	Main data struct	LOC	# dynamic mem read inst (10^9)	Application data used for acceptance check	Criteria for acceptance check
LULESH	Hydrodynamics	Struct mesh	2.9k	1.0	Mesh points	Same # of iterations, Energy conservation, Measures of symmetry: smaller than 10^{-8}
CLAMR	Hydrodynamics	Adaptive mesh	60.1k	6.4	Mesh points	Threshold for the mass change per iteration
PENNANT	Hydrodynamics	Unstruct. mesh	5.0k	9.3	Mesh points	Energy conservation
COMD	Molec. dynamics	Grid	5.6k	8.6	Atom properties	Energy conservation
FLUID	Fluid dynamics	Newtonian particle grid	5.7k	1.8	Particle properties	Particles' positions, velocities, bounding boxes with absolute tolerance
TENSORDECOMP	Sparse tensor decomposition	Multi-D Array	10.2k	0.2	Output matrices	Decomposition fit > 0.65 and delta < 10^{-3}

time-consuming for large-scale applications. This problem, however, has been largely mitigated in BonVoision's stride speculation heuristic, due to the following reasons: *(i).* the heuristic runs over static instructions, so it is unlikely to cause state explosion; *(ii).* it looks for the dependence chains of the index (and/or transitive index) register(s), *(iii).* it is limited to the scope of the function the DUE-causing instruction belongs to (i.e., it is intra-procedural).

4.3 Computing and writing-back the repair value

Based on the results of the motivating study presented in Section 3.2, the closest elements offer the smallest standard deviations across all applications. BonVoision operates as follows: once the stride is determined, BonVoision uses the stride to chooses the closest neighbours located at (*addr*-1*stride, *addr*+1*stride) and obtains the data (l,r) from those elements correspondingly (assume the DUE occurs at *addr*). It then computes the arithmetic mean of l and r, and writes the mean back to the address *addr* as the repaired value. After this, BonVoision lets the program continue the execution. Because a typical word is 8 bytes, if the inferred stride is smaller than 8 bytes (e.g. 4 bytes) , BonVoision needs to repair the entire memory word. For each unit (e.g. 4 bytes) in the word that needs to be repaired, BonVoision attempts to locate the closest elements beyond the scope of the same memory word, and use the average of those elements to write back to the unit.

Handling vector instructions Modern CPUs provide direct support for vector operations where a single instruction is applied to multiple data (i.e. SIMD). For example, MOVAPD in SSE2 instruction sets loads 16 bytes of data from the memory to a 128-bit register like XMM. If an ECC exception is triggered when executing such instructions, BonVoision assumes that the corrupted bits are in the first 8 bytes of the memory and repairs them correspondingly. Repairs that cover neighbouring memory word will be explored in the future work.

Implementation-level mechanisms. For the system to adopt BonVoision, the following mechanisms should be in place. Firstly, the applications should not react (i.e., terminate) to the SIGBUS signal. This can be implemented by overwriting the signal handler from the application, or by reconfiguring the application's signal-handling actions [21]. Secondly, when performing the repair, the application should bypass the cache and directly overwrite the corrupted data in the memory [4]. This can be implemented with a class of intrinsics (i.e. _mm_stream_si32, _mm_stream_si128, etc) supported by compilers such as *gcc* or the Intel compiler, or the binary can be instrumented directly with the corresponding instructions such as MOVNTI from Intel SSE2. Thirdly, the OS kernel needs to perform a translation from the physical memory address to the virtual memory address for the DUE and pass the virtual memory address to the run-time. We assume that the DUE is triggered by an application read and not by scrubbing, since memory scrubbing is likely to occur during the idle periods.

Limitations. There are a few cases that adversely affect BonVoision's performance. First, if a program implements loop unrolling on some loops, then the elements are accessed in a user-defined fashion which makes the stride speculation much more difficult from the corresponding assembly code. Second, the advanced heuristics assume that frequently accessed elements on the heap are likely to be iterated over in a loop so that the stride can be speculated from the operations on certain registers during the address calculation. Our assumptions are based on two common practices in scientific applications: *(i)* many simulation problems tend to access nearby data iteratively, *(ii)* applications tend to exploit locality of reference.

5 Experimental Methodology

We conduct large-scale *fault injection* campaign to evaluate BonVoision's effectiveness. We follow the methodology described in Section 3.1, with the slight difference that in each experimental trial, we do not explicitly "inject" a two-bit error into the memory as the corrupted data in the location will be completely overwritten by the repair anyways.

Benchmarks. We use (1) four representative DOE mini-applications, namely, LULESH [19], CLAMR [42], PENNANT [31] and COMD [11], (2) two scientific applications: FLUID [41] from the PARSEC benchmark suite [5] and, (3) a state-of-the-art CANDECOMP/PARAFAC decomposition implementation (denoted as TENSORDECOMP) [33]. The details of these applications are presented in Table 2. Note that our benchmarks use representative internal data structures (structured grids / meshes, adaptive meshes, etc.)

Application-specific correctness checks. For many scientific applications, developers write acceptance checks that increase confidence that the result was not impacted by a numerical error, an SDC, or simply a software bug. These acceptance checks are often based on energy conservation or numeric tolerance for result approximation. In practice, the check is typically placed at the end of the execution of the application [19], or if the application uses a C/R scheme, at the end of each checkpoint interval [42]. For *CLAMR*, *FLUID*, and *PENNANT*, we use the built-in acceptance checks (written by the developers). For *LULESH*, *COMD*, we wrote the checks ourselves, based on application verification specifications: Section 6.2 in [28] for *LULESH*, "Verification correctness" section in [12] for *COMD*. For *TENSORDECOMP*, we consulted with the benchmark's

[4] Otherwise there could be recurrent ECC errors due to the mismatched ECC. A memory write can trigger the computation of the new ECC for the memory data

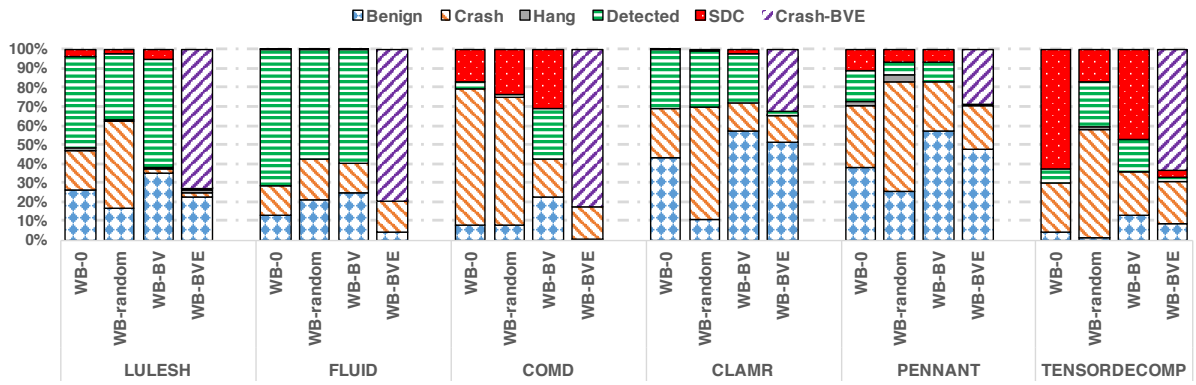


Figure 4: Application outcome ratios when DUEs are repaired by WB-0 (left), WB-random (center-left), BonVoision (center-right), and BonVoision-E (right). Crash-BVE class denotes that BonVoision-E predicts the repair will lead to detected or SDC outcome, a situation similar to a crash in the context of a C/R, as applications would continue from a checkpoint. For BonVoision-E, the crash and SDC rates are low or zero for most benchmarks, as a result they are not visible in some plots.

developer and wrote the check that examines the decomposition fit rate at the end of each iteration for convergence. Table 2 describes the the criteria used in the acceptance checks for each benchmark.

Outcome classification. A first decision is made based on whether the application appears to complete its run successfully after a repair. If not, the application either receives an OS signal and terminates before it finishes (a *crash*), or the application does not finish in an expected time (a *hang*) and is terminated. If yes, then the output of the final application state is checked. If the application's results do not pass the acceptance check, the outcome is labelled as *detected*, while if the application passes the check, the core program output is compared bit-wise with the output from a fault-free run: if they differ then the outcome is *SDC*, otherwise the outcome is *benign*. Note that this is conservative estimate as we do not consider application-specific semantics in interpreting the output.

Implementation and experimental setup. BonVoision is implemented using Intel Pin tool [44] 3.5. All experiments are run on a server with 40 Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz processors running Linux 4.13.0. *We perform* 5,000 *injections per application to obtain tight error bounds of 0.2% - 1.3% at the 95% confidence interval.* All benchmarks are compiled with gcc 8.2 using O3, which auto-generates X86-64 SSE2 instructions.

6 Evaluation Results

We first evaluate the effectiveness of BonVoision in repairing memory errors that result in DUEs (Section 6.1 and Section 6.2). We then measure BonVoision's performance overheads while scaling up the application (Section 6.3). Finally, we demonstrate the use of BonVoision in the context of a typical C/R scheme and evaluate its impact on reducing C/R overheads (Section 6.4).

6.1 Q1: How effective is BonVoision?

Comparison baselines. We introduce two repair strategies other than BonVoision, namely write-back with 0 (labeled WB-0 in plots) and write-back with random (WB-random), both attempting to repair the corrupted memory data and continue the execution of an application. Based on past related work [21, 37], both strategies are viable, and are hence reasonable as baselines.

Overall effectiveness. To measure BonVoision's effectiveness we focus on the benign and SDC outcomes of a repair: the more benigns and the fewer SDCs BonVoision converts DUEs into, the more effective are its repairs. Figure 4 (left three bars for each benchmark) shows the outcome of our error injection campaign after repairing DUEs with BonVoision (indicated as WB-BV), WB-0 and WB-random. For each benchmark, we present the percentage of each type of outcome under different repair techniques. *Overall, applications repaired by BonVoision lead to highest benign rates among three repair strategies across all benchmarks.*

In particular, the improvements of benign rates range from 5% to 20% compared to WB-0, and from 5% to 47% comparing WB-random (all expressed as absolute value). On average, BonVoision achieves on average 12% and 21% improvements over WB-0 and WB-random, respectively. We observe, on average, 0.8× and 2.5× improvement in benign rates when comparing WB-BV with WB-0 and WB-random, respectively (expressed as relative values).

For LULESH, FLUID, CLAMR and PENNANT, the SDC rates are less than 7% across all repair strategies; although BonVoision leads to higher SDC rates, the difference is rather marginal, ranging from 0.1% to 2% (absolute). For COMD and TENSORDECOMP, we observe relatively high SDC rates from three strategies: 23%(WB-0), 27%(WB-random) and 30%(WB-BV) for COMD, 63%(WB-0),16%(WB-random) and 45%(WB-BV) for TENSORDECOMP.

WB-BV outperforms the strategy that ignores DUEs: Comparing the results of BonVoision with the results of ignoring DUEs in Figure 1, BonVoision is able to reduce the SDC rates from 22% to 14%, and increase the benign rates from 26% to 36% on average.

WB-BV offers more effective repairs than WB-random Although WB-random offers the lowest SDC rates, it performs poorly for converting DUEs to benign outcomes. Specifically, the repairs lead to the highest crash and detected rates among the three strategies on average, significantly degrading the benefit of the repairs in the context of the C/R: the application either crashes or violates the application's checks (if any), and rolls back to the checkpoint.

To summarize, BonVoision outperforms both WB-0 and WB-random in offering the most effective repairs: BonVoision results in the highest benign rates for all benchmarks, and only incurs marginal increases in SDC rates in four benchmarks. The results

were similar when we tested with different data inputs for LULESH, CLAMR, and PENNANT. For COMD and TENSORDECOMP, none of the three techniques: WB-BV, WB-0, and WB-random, offers effective repairs in terms of minimizing the SDC rates.

6.2 Q2: Can machine learning help improve BonVoision?

One of the main uses for BonVoision is in the context of a C/R scheme, thus, the following discussion will be guided by this context. When using BonVoision, upon a memory DUE, the application can roll forward instead of rolling backward since the corrupted data likely gets repaired by BonVoision. However, as shown in Figure 4, the chance that the repair is followed by a crash, detected, or SDC is not negligible, which raises concerns[5] when using BonVoision with a C/R system. We elaborate these concerns and explain why they need to be addressed for BonVoision to run with C/R: *(i)* most importantly SDCs are the worst case as the application produces incorrect results without notifying the users; *(ii)* the detected errors result in the waste of resources: as the correctness checks are typically placed at the end of the checkpoint interval, a BonVoision repair that triggers an application correctness check error at the end of the interval is costlier than a crash triggered by the DUE; and finally *(iii)* crashes are similar to detected though they incur lower cost, as a crash would generally re-occur before the end of the checkpoint interval [6]. Therefore, we consider the crashes to be not harmful in this study and focus on the other outcomes.

One way to alleviate these concerns is to predict the outcome of a BonVoision repair, and just recover from a checkpoint if the predicted outcome is SDC, detected or crash. To this end, we explore the effectiveness of an online classifier to predict the outcome of BonVoision repairs. The classifier uses as features our estimators of space smoothness around the repair site and is trained based on ground truth obtained during the error injection campaigns.

The classifier is built using the standard supervised machine learning process: we collect the standard deviation calculated from each trial of the fault injection experiments for each benchmark, and assign the label to each standard deviation based on the outcome class of the trial. We follow standard practice and split the dataset 80/20 as the training and testing set, train the classifier on the training set with a couple of classification algorithms (e.g., linear-regression and decision-tree), and test the models on the testing set to determine the best-performing model.

Model tuning. We prioritize avoiding primarily SDCs, and, secondly detected outcomes. Table 3 shows the corresponding confusion matrix. We note that, depending on the operational context, the classifier can be tuned to prioritize other success metrics.

Classifier performance. Table 4 shows the classifier performance on each benchmark. Overall, the decision-tree classifier provides the best results: depending on application, the accuracy of the classifiers are from 70% to 92.4%, offering good accuracy to separating benign and not benign outcomes (SDC and detected). In particular, as the classifier is optimized for a conservative prediction on benigns when the actual class of the repairs are also benigns, the recalls (i.e., the true positive rates) for the benchmarks can be relatively low, while the selectivity (i.e., the true negative rate) soars from 88% to 100%, meaning that when the classifier predicts that a repair would not lead to a benign outcome, it is likely that the prediction leads to detected or SDC. Thus, a C/R scheme is confident to crash the application for those cases. It is observed that, for TENSORDECOMP, the mis-classification rate of detected cases can be a little higher than 10%, and mis-classification rate of SDCs is 8%. In fact, for this case, the absolute number of SDCs and detected cases is relatively small [7].

Table 3: The confusion matrix for the classifier, associated with mis-classification costs. Low cost for misclassifying a repair that leads to a benign outcome (in these cases the C/R scheme will restart from a checkpoint similarly to when BonVoision is not used), high cost if SDCs or detected are predicted as benigns.

Predicted \ Actual	Benign	Detected or SDC (not benign)
Benign	True Positive Cost: low	False Positive Cost: high
Detected or SDC (not benign)	False Negative Cost: not high	True Negative Cost: low

Comparing with vanilla BonVoision. Figure 4 highlights the impact of using the online classifier: the outcomes of using the online predictor are labelled BonVoision-E(nhanced) - right bars in the plots. The ratio of predicted SDC or crash outcomes is identified separately. We assume that the crash rate stays constant between BonVoision and BonVoision-E, so all the types of outcomes are adjusted proportionally, which leads to a conservative estimation The key takeaway is that BonVoision-E is capable of eliminating most SDC and detected cases for all the benchmarks and for some benchmarks it completely eliminates SDCs.

6.3 Q3: What is BonVoision's Efficiency?

BonVoision runs in two phases: the dynamic instrumentation phase for stride speculation on the program's assembly code, and the dynamic execution phase that performs the actual repairs. We aim to estimate the overhead of each of these phases, and how overheads scale when the application scales up in either input size or the number of processes. We present the results for only a single application CLAMR, though similar results were observed for all benchmarks. We measure the execution time of each phase under different input sizes and numbers of MPI processes with CLAMR, and compare those times with the time spent for each iteration by CLAMR. For each configuration, we run the application over 100 times, and measure the average of the execution time and the iteration time. We use CLAMR in this experiment because it has a basic C/R feature enabled, and the C/R runs at the end of each iteration as used in real world.

Figure 5a shows that the execution times of both phases remain approximately stable across different input sizes, while the iteration time increases much faster. Figure 5b shows a similar trend in the executions times of BonVoision while the number of MPI processes increases. The dynamic instrumentation time and the

[5]The similar concerns are also discussed in the recent study [20]

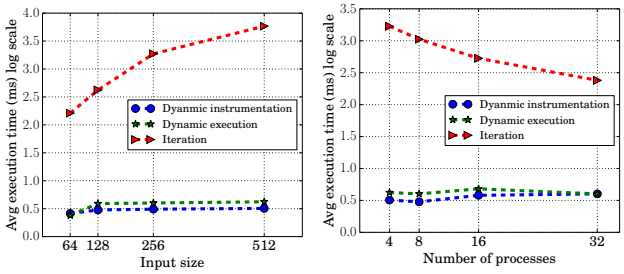
[6]We measure the interval between the time the application gets continued after the repair and the time the crash occurs, and observe that all the crashes occur nearly right after the repair.

[7]We applied this process on LULESH and PENNANT for additional inputs (scale and values), and we observed qualitatively similar results.

Table 4: Classifier quality. The mis-classification rate of SDC shows the fraction of predicted benigns but the true class are SDCs divided by the total number of SDCs in the testing set. Similarly for the misclassification rate of detected.

Benchmark	Accuracy	Recall	Precision	Selectivity	False Positive Rate	Misclassification Rate of SDC	Misclassification Rate of Detected
LULESH	88.5%	55.4%	99.0%	99.7%	0.2%	3.0%	0%
FLUID	85.7%	22.0%	84.2%	98.8%	1.1%	0	1.0%
COMD	70.0%	5.8%	100.0%	100.0%	0	0	0
CLAMR	91.2%	89.2%	96.6%	94.8%	5.0%	0	6.0%
PENNANT	92.4%	90.0%	99.2%	98.3%	1.6%	5.0%	0.0%
TENSOR-D.	85.3%	48.5%	54.0%	88.4%	2.5%	8.0%	13.0%

dynamic execution time adds up to a total execution time around 6 milliseconds overall for BonVoision. The overheads for the other benchmarks are in the same order of magnitude as well (0.5 to 2 ms). This level of overhead is negligible for most HPC applications.



a Execution time for each BonVoision phase and the iteration time (including checkpointing) while scaling input size

b Execution time for each BonVoision phase and the iteration time while scaling number of processes

Figure 5: The performance overhead incurred by BonVoision scales well and is overall negligible while the application scales up.

6.4 Q4: What is the impact in the context of C/R?

For long-running scientific applications, the ability to tolerate memory DUEs is essential to make progress to completion - thus they usually employ C/R schemes. To evaluate the impact of BonVoision-E in this context we have developed we use a finite state machine to model the applications that run with a typical C/R system with and without BonVoision. We conduct experiments using an in-house continuous-time event simulator [21] on the applications in both scenarios to estimate resource usage efficiency (i.e., the ratio between the time the application does useful work and the total application runtime), a metric widely used for C/R's efficiency [6, 14, 59]).

We make the following assumptions to simplify the model. First, all crashes are due to memory DUEs, other memory soft errors are corrected by the ECC/chipkill, and no DUEs occur during the checkpointing process. Second, no other fault tolerance mechanisms than C/R are used. Third, we assume the applications take synchronous coordinated checkpoints. Thus, when a crash occurs on one node, all the nodes used by the application have to roll back to the last checkpoint and re-execute from that point. These are standard assumptions [14, 21, 59].

Parameter description. *Checkpoint interval.* The checkpoint interval defines the frequency to take checkpoints. Young's formula [59] indicates that the optimal checkpoint interval is determined by the MTBF and the checkpointing overhead.

MTBF. We extrapolate the MTBF based on the error rates occuring on the Blue Waters supercomputer. Martino et al. [38] report that during the measurement period, i.e. roughly 6,177 hours, 1,031,886 memory errors are observed on 12,721 Blue waters nodes, and 70.01% of them involve 1 bit error, 29.98% of them have 2-8 consecutive bits errors, only 28 errors are not correctable by SECDED ECC/chipkill. Therefore, if there is only SECDED ECC deployed, the DUE MTBF rate is 1.2s, and with x8 Chipkill, the DUE MTBF is 794,185s. The MTBF of the system is inversely proportional to the size of the system: if the system scales up by an order of magnitude, the MTBF decreases by an order of magnitude [6], assuming that similar device technology is used.

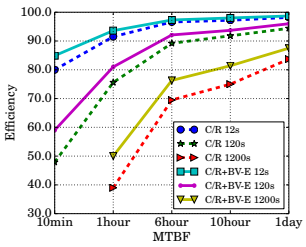
Checkpoint overhead. For system level checkpointing techniques, normally the entire memory (typically 32GB to 128GB on modern system node) is backed up to persistent storage. We assume three types of systems: a well-provisioned system with burst-buffer implemented with SSD (I/O bandwidth: average 1GB/s, peak 6GB/s), an average-provisioned system that has burst-buffer and spinning disk, and an under-provisioned system that only deploys spinning disks (I/O bandwidth 50MB/s to 500MB/s), and extrapolate the checkpointing overhead to be 12, 120 and 1200 seconds respectively.

Other parameters. We optimistically assume that the recovery overhead equals the checkpointing overhead (i.e., equal storage read and write speeds). We also assume application correctness checks at the end of checkpoint interval take 1% of checkpoint overhead, and that the application needs to spend 10% of checkpointing overhead to synchronize across all nodes as we assume synchronized checkpointing.

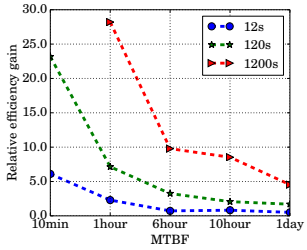
Results Figure 6 shows the efficiency comparison between the vanilla C/R system and the C/R system using BonVoision-E. Left plots present absolute application efficiency in various configurations while right plots present the relative improvement offered by BonVoision-E. We vary the system size in combination with different checkpointing overheads. We consider systems with 50k, 60k, 100k, 600k, and 3600k nodes - with MTBF scaled from data inferred from Martino et al. [38]. These have MTBFs of 1 day, 10 hours, 6 hours, 1 hour and 10 mins respectively. Overall, C/R+BonVoision-E achieves higher efficiency than C/R across all configurations, ranging from 76% to 1%, and in particular on average 8% for LULESH, and 15% for CLAMR, 15% for PENNANT, 3% for TENSORDECOMP and FLUID, and a marginal improvement for COMD.

We also find that there is a clear trend for the efficiency gain offered by BonVoision-E: when MTBF decreases, the gains accelerate across all configurations. We speculate that this enables an opportunity for future HPC systems to reduce energy consumption: the system could intentionally reduce the DRAM refresh rate [36, 47, 53], resulting in increasing memory error rates. Applications running with C/R+BonVoision-E can potentially maintain the same level of efficiency e.g., as shown in Figure 6, the efficiency of C/R+BonVoision-E's is about 90% at 1 hour MTBF and 12 seconds checkpointing overhead, while C/R's efficiency is 91% at 10

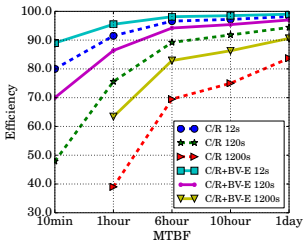
minutes MTBF and the same checkpointing overhead. This trend is consistent for other configurations across all benchmarks.



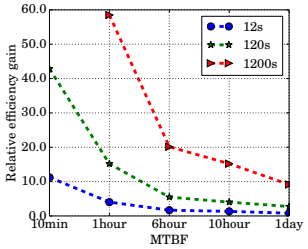
a Efficiency in the context of a typical C/R and C/R with BonVoision-E for LULESH



b Relative efficiency gain of C/R+BonVoision-E over typical C/R for LULESH



c Efficiency in the context of C/R and C/R with BonVoision-E for CLAMR



d Relative efficiency gain of C/R+BonVoision-E over typical C/R for CLAMR

Figure 6: Efficiency comparison between C/R and C/R+BonVoision-E for simulated 10-years of application time for LULESH and CLAMR. The MTBF decreases as the system scales up.

7 Related work

Levy et al. [32] propose a methodology that leverages the data similarity between memory pages to perform compression on memory, and use the compressed memory pages to repair the ones that are impacted by DUEs. At a high level, their goal is similar to BonVoision, while there main three differences: *(i)* BonVoision leverages the spatial data smoothness across data elements of a program, which leads to a more efficient implementation in terms of performance and storage overhead; *(ii)*, as stated in their paper, the memory compression may not be always a viable operation, as they can only protect pages that are read-only at the beginning of each protection interval. *(iii)*, when a DUE occurs and the corrupted page is eligible to recover, they essentially roll that corrupted page back to its previous state, and which can cause a cascading recovery.

Gottscho et al. [24] propose software-defined ECC (SDECC) that uses side information to estimate the original message by filtering and ranking possible candidate codewords. The SDECC technique [23], is shown to be able to successfully recover most of DUEs for integer applications. Along this direction, Schoeny et al. [48] focus on the coding side of SDECC and apriori designate specific messages to protect the system from errors with a stronger confidence. BonVoision differs from these two papers in two ways. First, they assume that the ECC syndrome is available for investigation, while BonVoision takes a conservative position and does not assume the availability of the syndrome bits. In fact, if the syndrome is available to BonVoision, then the search space for possible repairs would be further narrowed. Second, both papers [24, 48] need hardware-level support and modifications, which is not possible for most HPC systems. Poulos et al. [17] leverage both the candidate code-words and the application-specific contextual information to attempt to repair DUEs without hardware modification. However, they require application-specific information to guide the selection of the codeword making the approach less general than BonVoision.

One work that is closely related to BonVoision is *LetGo* [21] which takes advantage of the "temporal" redundancy in a program by skipping the instructions that cause exceptions due to soft errors in the processor. There are two differences between *LetGo* and BonVoision. First, *LetGo* focuses on soft errors that occur in the processor's computational data-path and register file, while BonVoision focuses on memory errors - which can be more damaging to overall application state. Second, *LetGo* uses fairly simply heuristics to repair erroneous registers (e.g., replacing them with a 0) while BonVoision leverages the "spatial" redundancy and the data smoothness to find similar data to repair DUEs. As we have shown (Section 6), the simple heuristics used by *LetGo* do not work for repairing memory errors. Following LetGO, Hamada et al. [25] attempt to reactively repair floating point NaN errors in registers and memory. However, their paper does not explain how a NaN should be repaired and defers the solution to future work. Though, BonVoision focuses on different fault model, our observations on spatial data smoothness can also be leveraged by their technique.

Algorithm based fault tolerance (ABFT) exploits mathematical properties of some problems to check for data violations and aid error recovery [7–9, 27, 35, 49, 50, 57, 58]. They are generally restricted to specific application areas such as linear algebra and certain types of fault models, and they require great effort to implement, as opposed to BonVoision that requires no program level information or source code change.

Failure oblivious computing and roll-forward recovery: Rinard et al. [45, 46] propose failure oblivious computing, an approach that continues application's execution when memory-related errors occur during execution. For invalid memory loads, the implementation manufactures a value to feed the register, and for invalid memory writes. it simply discards the written value. BonVoision focuses on memory DUEs, and show that the simple repair strategy may not be efficient to fix memory soft errors resulting in DUEs. Chien et al. [10] propose a global view resilience (GVR) framework that allows applications to store and compute an approximation from the current and versioned application data for a forward recovery. BonVoision is free from the heavy program instrumentation's, and gets applied only when there is a DUE occured.

Binary analysis has been an active research area for decades, aiming to automatically analyze/improve the program for performance optimization [26, 52], validating the security risk [40], and fixing software bugs [30], etc. while BonVoision is in line with the binary analysis approach, it has two major differences from prior studies: i). BonVoision does not rely on either source-code or debug information for data structure speculation, 2). BonVoision requires no run-time instrumentation, hence no runtime overhead is incurred.

8 Discussion

While this paper demonstrates that BonVoision and BonVoision-E can effectively and efficiently recover applications from memory

DUEs using the stride-based speculation heuristic that leverages space data smoothness, there are three issues that should be discussed, including *(i).* is the proposed heuristic superior to other repair approaches? *(ii).* can one use BonVoision on other architectures like GPUs? *(iii)*, how well would BonVoision work with more advanced ECC schemes like chipkill?

(i). Compare the stride-based approach with others The heuristic proposed by BonVoision assumes no information from the application and speculates the stride of the data structure with a lightweight dynamic analysis. While we show its usefulness, it is worth investigating if this approach works better than other possible solutions for devising the repairs for memory DUEs. Below, we discuss our perspective on this comparison:

- *speculating the stride differently.* A straightforward approach to speculate the stride of the data structure is to find the immediate neighbours using the size-based approach. To Obtain the size one only needs to parse the DUE-causing instruction at run-time, so it incurs less overhead than BonVoision's stride-based heuristic. However, as many HPC applications employ complex data structures to represent the physical space, the immediate neighbours may not necessarily reflect the spatial data smoothness.
- *estimating the impact of potential data similarity.* The program data might be similar across a large chunk of memory. In this case, the proposed heuristics might output incorrect strides while still offering appropriate repairs. To rule out this factor, we propose two approaches: *random with stride*, which reads elements from two random locations in the range of (i.e. addr-128*stride, addr+128*stride), and computes their average as the repair value; and similarly *random with size*, which reads elements from two random locations in the range of (i.e. addr-128*size, addr+128*size) and compute their average as the repair value.

We have executed a fault injection experiment similar to the one presented in Section 6.1. For each error injection trial that leads to benign, SDC or detected case (we exclude crashes as applications terminate in these cases hence the repairs have no impact on the output's correctness), we compute the relative difference between the original data in the corrupted memory location and the repair values computed based on each of the approaches above. We find that, the stride-based approach (i.e. BonVoision's heuristic) outperforms other approaches significantly. In particular, BonVoision offers on average 41% more values that differ from the original value by less than 5%.

(ii). Portability The current implementation of BonVoision mainly focuses on supporting well-adopted architectures (i.e. X86-64) in HPC systems. As the trend in HPC moves towards heterogeneous computing, it is worth understanding the BonVoision's feasibility for diverse architectures like GPUs and ARM CPUs. The two fundamental ideas of BonVoision: leveraging spatial data smoothness in the application' data for repair, and attempting to infer the stride of the data structure based on how the index register is adjusted, are independent of the characteristics of a specific architecture. While extending BonVoision for diverse architectures is promising, there are certain issues that need attention. For example, BonVoision requires the just-in-time compiler tool for dynamic instrumentation, which may not be easily accessible on other platforms; and in particular to GPUs, additional complexity results from less mature interrupt functionality on current GPUs and GPU-specific optimization techniques like coalesced memory access.

(iii). Advanced ECC support Advanced ECC schemes like chipkill can largely improve the memory error resilience, and have been deployed in some HPC systems. For example, Blue Waters [38] feature both x4 (i.e. single symbol error correction and dual-symbol error detection, with 4 bit/symbol), and x8 chipkill ECC DRAM. It is interesting to understand how would BonVoision work with such schemes. In Section 4.3 we introduce the BonVoision's strategy for writing the repair back to the memory: BonVoision's repair overwrites the entire memory word that contains the error bits. Therefore, as long as the detected error bits (i.e. by chipkill) are in the same memory word, BonVoision should be able to handle it.

9 Conclusions

Memory soft errors are projected to be more frequent in future computing systems due to technology scaling and the large sites. One of the concerns is that memory soft errors might increasingly result in detectable but uncorrectable errors (DUEs), causing application failure. In this paper, we presented BonVoision, a lightweight, automatic approach that leverages spatial data smoothness present in HPC applications to repair the corrupted memory values that cause DUEs, with estimates based on neighboring data elements. We find that when including our most evolved techniques, on average, BonVoision-E is able to continue to completion 30% of the cases while decreasing the application's SDC to almost zero.

BonVoision creates the opportunity to trade off confidence in results for efficiency (time-to-solution or energy-to-solution). Certainly, for some applications - or for some operational situations - confidence in results is the user's primary concern, and BonVoision will not be used. We believe, however, that there are many situations that make this tradeoff attractive: Firstly, since Silent Data Corruptions (SDC) can occur anyway (due to bit-flips, regardless of whether BonVoision is used), HPC users are already taking the risk of getting incorrect results, and have developed techniques to validate their results. For example, application-specific checks to diminish this risk are an active area of research [27] and BonVoision will benefit from all these efforts. Secondly, for some applications BonVoision performs extremely well (e.g., for CLAMR all faults that would lead to crashes can be elided by BonVoision, without any additional SDCs). In these cases, BonVoision certainly represents an appealing solution. Finally, note that it is trivial to collect information on whether a particular run has benefited from BonVoision repair heuristics and offer users additional information, based on which one can reason about the confidence in results.

Acknowledgement

This work was supported in part by the U.S. Department of Energy's (DOE) Office of Science, the National Sciences and Engineering Research Council of Canada (NSERC), Office of Advanced Scientific Computing Research, under award 66905. Pacific Northwest National Laboratory is operated by Battelle for DOE under Contract DE-AC05-76RL01830.

References

[1] [n. d.]. International Technology Roadmap for Semiconductors./ Model for Assessment of CMOS Technologies and Roadmaps (MASTAR). http://www.itrs.net. Semiconductor Industries Association.

[2] [n. d.]. Samsung Electronics Develops WorldâĂŹs First Eight-Die Multi-Chip Package for Multimedia Cell Phones. http://www.samsung.com. Samsung Electronics Corporation.

[3] Rizwan Ashraf, Roberto Gioiosa, Gokcen Kestor, Ronald F DeMara, Chen-Yong Cher, and Pradip Bose. [n. d.]. Understanding the propagation of transient errors in HPC applications. *SCâĂŸ15* ([n. d.]).

[4] L. Bautista-Gomez and F. Cappello. 2015. Exploiting Spatial Smoothness in HPC Applications to Detect Silent Data Corruption. In *2015 IEEE 17th International Conference on High Performance Computing and Communications*.

[5] Christian Bienia. 2011. *Benchmarking Modern Multiprocessors*. Ph.D. Dissertation. Princeton University.

[6] George Bosilca, Aurélien Bouteiller, Elisabeth Brunet, Franck Cappello, Jack Dongarra, Amina Guermouche, Thomas Herault, Yves Robert, Frédéric Vivien, and Dounia Zaidouni. 2014. Unified Model for Assessing Checkpointing Protocols at Extreme-scale. *Concurr. Comput. : Pract. Exper.* 26, 17 (Dec. 2014), 2772–2791. https://doi.org/10.1002/cpe.3173

[7] Jieyang Chen, Hongbo Li, Sihuan Li, Xin Liang, Panruo Wu, Dingwen Tao, Kaiming Ouyang, Yuanlai Liu, Kai Zhao, Qiang Guan, et al. 2018. Fault tolerant one-sided matrix decompositions on heterogeneous systems with GPUs. In *SC'18*.

[8] J. Chen, S. Li, and Z. Chen. 2016. GPU-ABFT: Optimizing Algorithm-Based Fault Tolerance for Heterogeneous Systems with GPUs. In *NAS*.

[9] J. Chen, X. Liang, and Z. Chen. [n. d.]. Online Algorithm-Based Fault Tolerance for Cholesky Decomposition on Heterogeneous Systems with GPUs. In *2016 IPDPS*.

[10] A. et al. Chien. 2015. Versioned Distributed Arrays for Resilience in Scientific Applications. *Procedia Comput. Sci.* (2015).

[11] P. Cicotti, S. M. Mniszewski, and L. Carrington. [n. d.]. An Evaluation of Threaded Models for a Classical MD Proxy Application. In *Hardware-Software Co-Design for High Performance Computing (Co-HPC), 2014.*

[12] P. Cicotti, S. M. Mniszewski, and L. Carrington. 2013. CoMD: A Classical Molecular Dynamics Mini-app. http://exmatex.github.io/CoMD/doxygen-mpi/index.html

[13] M. Dadashi, L. Rashid, K. Pattabiraman, and S. Gopalakrishnan. 2014. Hardware-Software Integrated Diagnosis for Intermittent Hardware Faults. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 363–374. https://doi.org/10.1109/DSN.2014.1

[14] J. T. Daly. 2006. A Higher Order Estimate of the Optimum Checkpoint Interval for Restart Dumps. *Future Gener. Comput. Syst.* (2006).

[15] Timothy J. Dell. 1997. A White Paper on the Benefits of Chipkill-Correct ECC for PC Server Main Memory by.

[16] N. El-Sayed and B. Schroeder. 2014. Checkpoint/restart in practice: When simple is better;. In *2014 Cluster*.

[17] Alexandra Poulos et al. 2018. Improving Application Resilience by Extending Error Correction with Contextual Information. In *IEEE/ACM 8th FTXS@SC 2018*.

[18] G. Bosilca et al. [n. d.]. MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes. In *Supercomputing, ACM/IEEE 2002 Conference*.

[19] Ian Karlin et al. [n. d.]. *LULESH Programming Model and Performance Ports Overview*. Technical Report.

[20] B. Fang, J. Chen, K. Pattabiraman, M. Ripeanu, and S. Krishnamoorthy. [n. d.]. Towards Predicting the Impact of Roll-Forward Failure Recovery for HPC Applications. In *DSN 2019 (Fast-abstract)*. IEEE.

[21] Bo Fang, Qiang Guan, Nathan Debardeleben, Karthik Pattabiraman, and Matei Ripeanu. 2017. LetGo: A Lightweight Continuous Framework for HPC Applications Under Failures *(HPDC '17)*.

[22] B. Fang, Q. Lu, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi. 2016. ePVF: An Enhanced Program Vulnerability Factor Methodology for Cross-Layer Resilience Analysis. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 168–179. https://doi.org/10.1109/DSN.2016.24

[23] Mark Gottscho. [n. d.]. Opportunistic Memory Systems in Presence of Hardware Variability. ([n. d.]).

[24] M. Gottscho, C. Schoeny, L. Dolecek, and P. Gupta. [n. d.]. Software-Defined Error-Correcting Codes. In *2016 DSN-W*.

[25] Shinsuke Hamada, Soramichi Akiyama, and Mitaro Namiki. [n. d.]. Reactive NaN Repair for Applying Approximate Memory to Numerical Applications. *CoRR* ([n. d.]).

[26] Laune C. Harris and Barton P. Miller. 2005. Practical Analysis of Stripped Binary Code. *SIGARCH Comput. Archit. News* 33, 5 (Dec. 2005), 63–68. https://doi.org/10.1145/1127577.1127590

[27] Kuang-Hua Huang and J. A. Abraham. 1984. Algorithm-Based Fault Tolerance for Matrix Operations. *IEEE Trans. Comput.* (1984).

[28] I.Karlin. 2012. LULESH Programming Model and Performance Ports Overview. https://codesign.llnl.gov/pdfs/lulesh_Ports.pdf

[29] Andi Kleen. [n. d.]. mcelog: memory error handling in user space.

[30] Michael Kling, Sasa Misailovic, Michael Carbin, and Martin Rinard. 2012. Bolt: On-demand Infinite Loop Escape in Unmodified Binaries. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '12)*. ACM, New York, NY, USA, 431–450. https://doi.org/10.1145/2384616.2384648

[31] Los Alamos National Laboratory. 2016. The PENNANT Mini-App v0.9. https://github.com/losalamos/PENNANT

[32] Scott Levy, Kurt B. Ferreira, and Patrick G. Bridges. 2016. Improving Application Resilience to Memory Errors with Lightweight Compression. In *SC '16*.

[33] Jiajia Li, Jimeng Sun, and Richard Vuduc. [n. d.]. HiCOO: Hierarchical Storage of Sparse Tensors. In *SC '18*.

[34] S. Li, K. Chen, M. Hsieh, N. Muralimanohar, C. D. Kersey, J. B. Brockman, A. F. Rodrigues, and N. P. Jouppi. 2011. System implications of memory reliability in exascale computing. In *SC '11*.

[35] Xin Liang, Jieyang Chen, Dingwen Tao, Sihuan Li, Panruo Wu, Hongbo Li, Kaiming Ouyang, Yuanlai Liu, Fengguang Song, and Zizhong Chen. 2017. Correcting Soft Errors Online in Fast Fourier Transform *(SC)*.

[36] Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G. Zorn. 2011. Flikker: Saving DRAM Refresh-power Through Critical Data Partitioning. *SIGPLAN Not.* 46, 3 (March 2011), 213–224. https://doi.org/10.1145/1961296.1950391

[37] Fan Long, Stelios Sidiroglou-Douskos, and Martin Rinard. [n. d.]. Automatic Runtime Error Repair and Containment via Recovery Shepherding. *SIGPLAN Not.* ([n. d.]).

[38] C. D. Martino, Z. Kalbarczyk, R. K. Iyer, F. Baccanico, J. Fullop, and W. Kramer. [n. d.]. Lessons Learned from the Analysis of System Failures at Petascale: The Case of Blue Waters. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*.

[39] Sarah E. et al. Michalak. 2014. Correctness Field Testing of Production and Decommissioned High Performance Computing Platforms at Los Alamos National Laboratory *(SC '14)*.

[40] Jiang Ming, Dinghao Wu, Gaoyao Xiao, Jun Wang, and Peng Liu. 2015. TaintPipe: Pipelined Symbolic Taint Analysis. In *Proceedings of the 24th USENIX Conference on Security Symposium (SEC'15)*. USENIX Association, Berkeley, CA, USA, 65–80. http://dl.acm.org/citation.cfm?id=2831143.2831148

[41] Matthias Müller, David Charypar, and Markus Gross. [n. d.]. Particle-based Fluid Simulation for Interactive Applications. In *SCA '03*.

[42] D. Nicholaeff, N. Davis, D. Trujillo, and R. W. Robey. [n. d.]. Cell-Based Adaptive Mesh Refinement Implemented with General Purpose Graphics Processing Units. ([n. d.]).

[43] E. Normand. 1996. Single event upset at ground level. *IEEE Transactions on Nuclear Science* (1996).

[44] H. et al. Patil. [n. d.]. Pinpointing Representative Portions of Large Intel Itanium Programs with Dynamic Instrumentation. In *MICRO-37*.

[45] Martin et al. Rinard. [n. d.]. A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors). In *Computer Security Applications Conference, 2004. 20th Annual*.

[46] Martin et al. Rinard. [n. d.]. Enhancing Server Availability and Security Through Failure-oblivious Computing *(OSDI'04)*.

[47] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. 2011. EnerJ: Approximate Data Types for Safe and General Low-power Computation. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 164–174. https://doi.org/10.1145/1993498.1993518

[48] C. Schoeny, F. Sala, M. Gottscho, I. Alam, P. Gupta, and L. Dolecek. 2017. Context-aware resiliency: Unequal message protection for random-access memories. In *ITW'17*.

[49] Manu Shantharam, Sowmyalatha Srinivasmurthy, and Padma Raghavan. 2012. Fault Tolerant Preconditioned Conjugate Gradient for Sparse Linear System Solution *(ICS)*.

[50] J. Sloan, R. Kumar, and G. Bronevetsky. 2012. Algorithmic approaches to low overhead fault detection for sparse linear algebra. In *DSN*.

[51] 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. *ASPLOS* (2015).

[52] Nathan R. Tallent, John M. Mellor-Crummey, and Michael W. Fagan. 2009. Binary Analysis for Measurement and Attribution of Program Performance. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 441–452. https://doi.org/10.1145/1542476.1542526

[53] Li Tan, Shuaiwen Leon Song, Panruo Wu, Zizhong Chen, Rong Ge, and Darren J. Kerbyson. 2015. Investigating the Interplay between Energy Efficiency and Resilience in High Performance Computing. In *Proceedings of the 29th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 786–796. https://doi.org/xpl/articleDetails.jsp?arnumber=7161565

[54] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott. 2010. Hybrid Checkpointing for MPI Jobs in HPC Environments. In *2010 IEEE 16th International Conference on Parallel and Distributed Systems*.

[55] Nicholas Wang, Michael Fertig, and Sanjay Patel. 2003. Y-branches: when you come to a fork in the road, take it. *Parallel Architectures and Compilation Techniques, 2003. PACT 2003. Proceedings. 12th International Conference on* (2003).
[56] J. Wei, A. Thomas, G. Li, and K. Pattabiraman. 2014. Quantifying the Accuracy of High-Level Fault Injection Techniques for Hardware Faults. In *2014 DSN*.
[57] Panruo Wu, Nathan DeBardeleben, Qiang Guan, Sean Blanchard, Jieyang Chen, Dingwen Tao, Xin Liang, Kaiming Ouyang, and Zizhong Chen. 2017. Silent Data Corruption Resilient Two-sided Matrix Factorizations *(PPoPP)*.
[58] Panruo Wu, Qiang Guan, Nathan DeBardeleben, Sean Blanchard, Dingwen Tao, Xin Liang, Jieyang Chen, and Zizhong Chen. 2016. Towards Practical Algorithm Based Fault Tolerance in Dense Linear Algebra. In *HPDC*.
[59] John W. Young. [n. d.]. A First Order Approximation to the Optimum Checkpoint Interval. *Commun. ACM* ([n. d.]).