FailAmp: Relativization Transformation for Soft Error Detection in Structured Address Generation

IAN BRIGGS, ARNAB DAS, and MARK BARANOWSKI, University of Utah
VISHAL SHARMA, Microsoft
SRIRAM KRISHNAMOORTHY, Pacific Northwest National Laboratory
ZVONIMIR RAKAMARIĆ and GANESH GOPALAKRISHNAN, University of Utah

We present FailAmp, a novel LLVM program transformation algorithm that makes programs employing structured index calculations more robust against soft-errors. Without FailAmp, an offset error can go undetected; with FailAmp, all subsequent offsets are relativized, building on the faulty one. FailAmp can exploit ISAs such as ARM to further reduce overheads. We verify correctness properties of FailAMP using an SMT solver, and present a thorough evaluation using many HPC benchmarks under a fault injection campaign. FailAmp provides full soft-error detection for address calculation while incurring an average overhead of around 5%.

CCS Concepts: • **Hardware → Error detection and error correction**; • **Computer systems organization** → *Reliability*; • **Software and its engineering** → *Software verification.*

Additional Key Words and Phrases: Soft error detection, failure amplification, structured address generation, LLVM transformation

ACM Reference Format:
Ian Briggs, Arnab Das, Mark Baranowski, Vishal Sharma, Sriram Krishnamoorthy, Zvonimir Rakamarić, and Ganesh Gopalakrishnan. 2020. FailAmp: Relativization Transformation for Soft Error Detection in Structured Address Generation. 1, 1 (February 2020), 22 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

High performance computing (HPC) is central to science and engineering: from critical scientific calculations such as climate simulation to advanced manufacturing methods for airframes and cars. While software bugs continue to be a weak link in the trustworthiness of HPC software [10], another weak link associated with the end of Moore's law is the propensity of simulations to be affected by *soft errors*—transient faults occurring within the silicon that cause erroneous values in the application state. In addition to high energy particles causing these faults, manufacturing variability, heat, and aging, coupled with internal noise and lower voltage margins also contribute to these *silent data corruptions*, affecting the integrity of long-running simulations [12, 20, 32].

Many types of software-based soft error detectors have been proposed: those that check for aberrations in the time series of the computational data [8], those that check that the loaded data matches what was stored [35], and detectors that fit a machine learning model around the "normal profile" of the computational data [30]. Unfortunately, all these

Authors' addresses: Ian Briggs, ianbriggsutah@gmail.com; Arnab Das, arnab.das@utah.edu; Mark Baranowski, mark.s.baranowski@gmail.com, University of Utah; Vishal Sharma, Microsoft, vishalcsharma.cse@gmail.com; Sriram Krishnamoorthy, Pacific Northwest National Laboratory, sriram@pnnl.gov; Zvonimir Rakamarić, zvonimir@cs.utah.edu; Ganesh Gopalakrishnan, ganesh@cs.utah.edu, University of Utah.

detection schemes introduce unacceptable computational overheads—30% for some, and much more for others. A 30% overhead in sequential performance clearly sets us back a few generations in terms of Moore's law. The detailed study of resilience solutions and their overheads provided in [5] emphasizes some of these points. Another significant drawback of these detection schemes is that they have false positive rates that are much higher than the rates at which faults themselves occur, thus potentially causing unnecessary recomputations. Last but not least, inserting error detectors into an application's code-base can have non-trivial software engineering challenges that are not often explicitly addressed.

One reason for this high detection overhead is that the detector is invoked very frequently—almost every time that key data values are generated. The second reason is that the computed data itself, or some key aspect thereof, gets checked. The FailAmp approach presented in this paper makes some key departures: (1) FailAmp is focused on a narrow but important aspect of the overall computation, namely *address generation*. Consequently, FailAmp has very low overhead (an average of 5%). (2) FailAmp does the error checking quite infrequently, and in the interim keeps "amplifying" (propagating forward) each fault.[1] (3) FailAmp is 100% precise—no false positives or negatives for the covered aspect of the fault. (4) The process of inserting detectors is completely automated, as FailAmp is nothing but an LLVM transformation of the given code into a new code that integrates detector deployment within it.

Accessing individual elements in values of aggregate types such as arrays and structures require structured address generation. These are compute-intensive parts of an HPC code's execution which are known to be important vulnerability windows that must be preferentially protected. In fact, virtually all modern microprocessors and DSP units employ a dedicated hardware unit called the *Address Generation Unit* (AGU) to offload address generation from the main execution unit [21]. Thus, our work is relevant to practice, and can actually be viewed as a well-targeted resilience solution, guarding bit-flips in the AGU.

```
double sum(size_t n,
           double *A) {
  double sum = 0.0;
  for (size_t i=0; i<n; i++) {
    sum += A[i];
  }
  return sum;
}
```

```
double sum(size_t n,
           double *A) {
  double sum = 0.0;
  for (size_t i=0; i<n; i++) {
    double* A_at_i = A+i;
    sum += *A_at_i;
  }
  return sum;
}
```

```
double sum(size_t n,
           double *A) {
  double sum = 0.0;
  double* relative_A = A;
  int relative_i = 0
  for (size_t i=0; i<n; i++) {
    int delta_i = i − relative_i;
    relative_A = relative_A + delta_i;
    relative_i = i;
    double* A_at_i = relative_A;
    sum += *A_at_i;
  }
  assert(&A[n] == relative_A);
  return sum;
}
```

Fig. 1. Summation Code with the Argument Pointer Transformed According to FailAmp

A simple example that illustrates our approach is in Figure 1. Let us consider the well accepted error model (e.g., [1, 4]) of a single bit-flip that affects the computation of index `i` occurs somewhere within the entire computation, causing one summand to come from an alternate location `A[i']` (left-hand code block). If `i'` strays outside the legal address space, the computation will very likely crash, thus obtaining a "free" detection, in effect. However, in today's large address spaces, this is becoming less likely, and the final **sum** will carry the difference between `A[i']` and `A[i]`. In a sense, the address generated, **A_at_i**, is computed with respect to the fixed base address of **A** and a moving offset **i** that

[1]We could also call FailAmp as "Break-fast" i.e., "fails faster, and fails visibly."

suffers one corruption (middle column code block). The rightmost column of Figure 1 expresses the high level idea behind relativization (our implementation is actually much more general, as we shall explain momentarily). Notice that we set **relative_A** to A, and set **relative_i** to **0**. In the loop, we generate **relative_A** in terms of its own *previous* value augmented by a **delta_i** value. Thus, once a **relative_A** is corrupted by a bit-flip at any point, *all subsequent address calculations based off it are also corrupted*, and the **assert** will fail.

The FailAmp approach generalizes what we described in the following ways:

- It handles objects of type **T** which are arbitrary nests of arrays and structs, i.e.,
  T ::= primitive | T* | { f_i : T_i }
- FailAmp handles addressing changes caused by vectorization.
- We have conducted extensive empirical validation of FailAmp on many examples, demonstrating the performance of the FailAmp-transformed code.
- Using an LLVM-level fault injector targeting address calculations, we empirically demonstrate that FailAmp obtains 100% coverage of all address generation faults.
- We also applied a symbolic formal verification tool to FailAmp's LLVM transformation, and caught a bug that resulted in a revision of our translation scheme. This bug was a rare corner case that arose only when we applied a compiler vectorization flag, the compiler chose to vectorize, *and* the runtime pointer was not aligned.
- FailAmp allows users to effect a trade-off between detection latency and detector overhead. By delaying the **assert** checks to the point just before a relativized pointer is lost, we can obtain the least overhead. In the absence of bit flips, FailAmp guarantees that all the generated addresses (and hence the data that are fetched) are equal in a bit-exact sense—thus preserving the intended program semantics.
- We demonstrate that the error detection efficacy of FailAmp is not reduced by source-to-source loop optimizations such as polyhedral transformations and tiling, which are very important for programs in this class.
- When used with an x86 code generator, FailAmp introduces an average overhead of approximately 5% for realistically sized examples (smaller examples often have higher overheads, as the relativization costs are not adequately amortized).
- We have developed a relativization approach that can exploit the ARM ISA to significantly reduce the overhead in many cases.

2 BACKGROUND

We focus on soft errors—transient bit flips—that affect instruction execution. These errors can impact combinational or sequential logic, and can affect one or more bits. A bit flip in the microarchitecture can either be masked (if the state affected is not required for architecturally correct execution [19]) or propagate to the architectural state. An error that reaches the architectural state might be benign (masked by the application), lead to a program crash, be detected by the application, or escape detection altogether and lead to silent data corruption. The goal of our work presented in this paper is to eliminate silent data corruptions caused by address-generation faults errors.

Beam-injection studies [26] have shown that there are many don't-cares at the circuit, logic, and architectural levels that mask a majority of the faults. It was reported in this paper that only a negligible percentage (about 0.03%) of the injected faults actually turned into *silent data corruptions*. In a follow-on study [25], however, it was pointed out that due to technology scaling trends, this number is bound to rise significantly. In addition to particle strikes, one now has to worry about the relatively higher variability in transistor switching characteristics, increasing levels of system noise, and component aging. Now with the "end of Moore's law" resulting in sub 10-nanometer feature sizes, and the immense pressure to reduce energy consumption, there is now a general level of consensus that some degree of adoption of system resilience solutions [3] is inevitable in all scientific computing platforms of the future. Even given all this, there is still widespread reluctance in adopting system resilience solutions in HPC. Our work shows that if resilience solutions are focused to narrow, but critical aspects of a computation, and if more failure-amplifying methods are developed, the overall detection costs can be brought down significantly.

When it comes to practical system resilience solutions, software-based solutions are, understandably, far more popular nowadays than hardware-based solutions. The main reason is that the extra hardware involved can prove to be a constant speed impediment, while sitting idle and not firing most of the time. Virtually all software-based solutions depend on redundant computations [3]. Also, a recovery method is often invoked following fault detection.

2.1 Closely Related Work

The idea of introducing relativization in structured address generation for detecting soft errors was first introduced by Sharma et. al. as part of their PRESAGE algorithm[28]. While the PRESAGE algorithm successfully demonstrated the feasibility of relativization scheme, the algorithm could only handle one-dimensional arrays. Our FailAmp work is a significant improvement over the PRESAGE algorithm as FailAmp can handle values of aggregate types of any arbitrary shape and size such as multi-dimensional arrays, etc. Also, unlike FailAmp, PRESAGE algorithm was not subjected to any formal analysis to verify its correctness which we believe is of paramount importance for any compiler pass which performs code transformation. Finally, in addition to the aforementioned differences, our work also introduces a new approach for exploiting ISAs such as ARM to further reduce error detection overheads.

3 THE TRANSFORMATION ALGORITHM

The FailAmp transformation algorithm takes in a stream of LLVM instructions and an effective address chain beginning at an object of the allowed type (arrays of structs) to be transformed. The power of FailAmp stems from the fact that compilers that generate the LLVM intermediate form (e.g., the Clang compiler) exploit the versatility of a single powerful instruction called the "get element pointer," or `getelementptr` (GEP) for structured address computation. FailAmp emits a transformed stream of LLVM instructions where the incoming GEP instructions acting on the effective address chain are replaced by *relativized counterparts*. Non-GEP instructions are simply copied unchanged into the output stream.

3.1 LLVM background

We now introduce the requisite background on LLVM. The versatility of GEP is illustrated by an example from [18]. Consider the C declaration `struct ST *s;` where `ST` is of type.

```
struct ST {int X; double Y; int Z; };
```

Given this, `&s[1].z` turns into a *single* GEP instruction

```
%arrayidx = getelementptr inbounds %struct.ST, %struct.ST* %s, i64 1, i32 2
```

A GEP instruction indexes by item, not by byte. The instruction takes the address `%s` and indexes the array to get item 1, then takes that address and indexes the struct to get item 2 which is field `Z`. Note that multi-argument GEPs (such as in our illustration) can be broken down into a sequence of single-argument GEPs that successively consumes each argument. GEPs are also employed when handling vectorization where the data layout is interpreted modulo the indexing steps needed by the vector instruction. Essentially, another GEP instruction for the vectorized output is generated.

3.2 An Example

We now present the FailAmp transformation on a simple example involving four GEPs in sequence:

$$e_2 = g(e_1, i_1);\ e_3 = g(e_2, i_2);\ e_4 = g(e_2, i_3);\ e_5 = g(A, i_4)$$

We present a GEP as $g(e, i)$ where e is an incoming effective address and i is an index (offset) with respect to e. This illustration covers all the corner cases of our formal transformation presented in Section 3.3. Figure 2 details this example. We consider an array A that is the target of relativization. Each GEP instruction carries as an argument an effective address and a displacement with respect to it, and computes a new effective address. For example, e_2 is obtained by adding e_1 and i_1, and so on. The effective addresses generated in this example are e_2, e_3, e_4, and e_5.

Consider the translation of $e_2 = g(e_1, i_1)$. For this, we obtain the displacement (from the array base) of e_1, via $pm[e_1]$, and add i_1 to it, thus obtaining d_{i_2}. Δ_2 is now the difference between the displacements, i.e., $d_{i_2} - d_{i_1}$. We now generate a GEP, namely $r_{e_2} = g(r_{e_1}, \Delta_2)$, thus generating the same address as e_2, albeit via the relativized chain. We finally update the pointer map at location e_{i_2} with the displacement d_{i_2}. The rest of the transformations may be understood in the same manner.

Role of a Pointer Map: To turn the current relativized address to the next relativized address, all we need is to add an "address delta." Address-deltas are easily calculated if we knew the displacement of each new effective address from the base of the array. These displacements are denoted by d_{i_x}. To compute these displacements, we employ a hashmap called the *pointer map* (*pm*) to keep an association between effective addresses and their displacement from the array base. We initialize $pm[A] = 0$; also, for our example, we already have a *pm* initialization coming in, namely $pm[e_1] = d_{i_1}$, corresponding to the incoming effective address e_1. As will be clear in Section 3.3, *pm* is held by our LLVM transformation system (not the runtime). Overall, we show how each GEP instruction is turned into three replacement instructions, and a *pm* update.

Each GEP in the source program will correspond to exactly one GEP in the transformed program that will compute the same address under fault free executions. The crucial observation here is that the relativized chain of addresses rp_x propagates bit-flips forward along the address calculation chain when faults are considered. At every juncture, under the absence of bit-flips, each effective address e_x matches the corresponding relativized address rp_x. However, under a bit-flip, they will not match.

We now study the various cases involved, without and with shortcuts that exploit special cases.

No shortcut: The standard approach for relativization involves an addition and a subtraction. This case is involved in handling the instruction $e_2 = g(e_1, i_1)$. Since we are inserting new instructions into the computation to detect bit flips, we need to be sure that our inserted instructions themselves do not incur a bit flip that goes undetected. Corruption to the add instruction will corrupt $\%d'$, which will flow through $\%\Delta$ and $\%r'$. This will leave the relative pointer and relative index in a self consistent state which will go unnoticed by our system. A possible mitigation for this would be to repeat the add and compare the results. This unoptimized case is very infrequent (see detailed evaluations in

$pm[A] = 0,\ pm[e_1] = d_{i_1}$
$e_2 = g(e_1, i_1)$:
– $d_{i_2} = pm[e_1] + i_1$
– $\Delta_2 = d_{i_2} - d_{i_1}$
– $r_{e_2} = g(r_{e_1}, \Delta_2)$
– $pm[e_{i_2} \leftarrow d_{i_2}]$
$e_3 = g(e_2, i_2)$:
– $d_{i_3} = pm[e_2] + i_2$
– $\Delta_3 = d_{i_3} - d_{i_2}$
– $r_{e_3} = g(r_{e_2}, \Delta_3)$
– $pm[e_{i_3} \leftarrow d_{i_3}]$

$e_4 = g(e_2, i_3)$:
– $d_{i4} = pm[e_2] + i_3$
– $\Delta_4 = d_{i_4} - d_{i_3}$
– $r_{e_4} = g(r_{e_3}, \Delta_4)$
– $pm[e_{i_4} \leftarrow d_{i_4}]$
$e_5 = g(A, i_4)$:
– $d_{i5} = pm[A] + i_4$
– $\Delta_5 = d_{i_5} - d_{i_4}$
– $r_{e_5} = g(r_{e_4}, \Delta_5)$
– $pm[e_{i_5} \leftarrow d_{i_5}]$

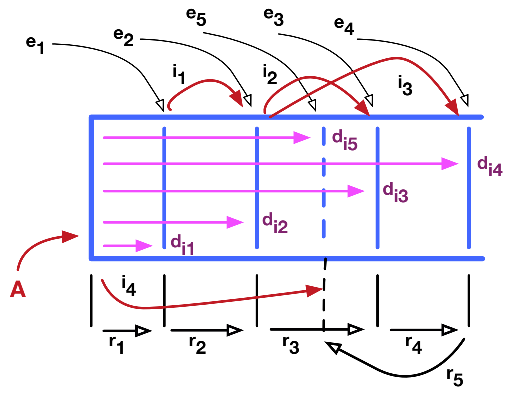


Fig. 2. FailAmp Illustration

Section 5.2). We have not yet implemented this protection for the add instruction; even if we were to, the overhead would be quite negligible.

On the other hand, if the subtraction is corrupted then the gep will also be corrupted, but the relative index, $\%d'$, will not be. This will be caught by our system.

Shortcut (R1): An example of this shortcut can be seen when handling the instruction $e_3 = g(e_2.Here, i_2)$ builds off the just calculated e_2. This allows us to directly determine that $\Delta_3 = i_2$, thus eliminating a runtime subtraction. Thus, under this shortcut, we are inserting just an addition. Corrupting this addition will only change the relative index while the relative pointer will be uncorrupted, hence the corruption will be caught.

Shortcut (R2): This shortcut can be exploited when handling the instruction $e_5 = g(A, i_4)$. In this example, A's effective address directly forms the first argument of the GEP. This allows us to determine that $d_{i5} = i_4$, given that $pm[A] = 0$, thus helping avoid a runtime addition. Thus, under this shortcut, only a runtime subtraction is involved. This subtraction will flow to the gep and yield a corrupt relative pointer but will not corrupt the relative index. The inconsistent state left by this will be caught by our system.

Shortcut (R3): If the subtraction is a constant value for all paths to the basic block, then we can also avoid a runtime subtraction. Thus, under this shortcut, there is no added computation whatsoever. We refer to this case as "stride" in Section 5.2. Section 5 provides a detailed evaluation of these cases.

3.3 Formal Translation Algorithm

We will now express the formal LLVM transformation as a process of code generation, focusing on an intra basic-block perspective. We insert assert statements into the translated output, serving as the error detectors. In the actual FailAmp implementation, we push these asserts outside of the basic blocks as far as possible. The basic idea is to push the asserts to the point just before the relativized pointers are lost. More specifically:

- For pointers that are live throughout a function, namely argument pointers, the assert can be pushed to the exit block. When a dependent pointer is transformed, these asserts are pushed along to two points:
  – just before the pointer is about to be overwritten, and

$$\begin{array}{l}
\langle pm, \quad \%e' = g(\%e, \%i); P, \quad \%r, \quad \%d, \quad R\rangle \\
\longrightarrow \\
\langle pm+ = (\%r', \%d'), \quad P, \quad \%r', \quad \%d', \quad R; NewCode\rangle
\end{array}$$

where,

$$\begin{array}{l}
NewCode = \\
\quad \%d' = add(pm[\%e], \%i); \\
\quad \%\Delta = sub(\%d', \%d); \\
\quad \%r' = g(\%r, \%\Delta) \\
\quad assert(\%A + \%d' == \%r')
\end{array}$$

Fig. 3. Formal Transformation in FailAmp

– when the pointer goes dead.

The first case occurs when the block that created the dependent pointer is revisited. In our implementation, an assert is placed before this information is lost (since the relativized pointer will otherwise be overwritten).

- For the second case, a simple liveness check is used, and all edges leading to blocks where the pointer becomes dead will carry an assert.

We also sketch how we stitch together the basic blocks, flowing the relative pointer around, introducing Phi nodes as needed—again, techniques considered standard. We do however check through extensive testing plus formal analysis described in Section 4 that these steps (beyond a single basic block) are (best-effort) correct. The correctness of the transformation at the basic block level has also been covered by our testing and formal analysis.

We express the translation through one state transition (structural operational semantics-style) rule in Figure 3. We use variable names of the form "$\%v$" mimicking LLVM SSA variable names. Unique internal names (corresponding to $\%r'$, $\%d'$, etc.) are automatically generated during the LLVM pass. The state contains five components:

- The pointer map pm, which is updated via $pm+ = (a, b)$, adding the key-value pair (a, b).
- An input program with one instruction highlighted, namely $\%e' = g(\%e, \%i)$ and the remainder of the program P coming after the semi-colon.
- A relative pointer $\%r$ coming into the basic block or already available in the basic block.
- A displacement $\%d$ coming into the basic block or already available in the basic block.
- The generated relativized program so far, namely R.

We generate a new five-tuple of next state:

- The pointer map pm is updated with $(\%r', \%d')$.
- The input program shrinks down to P (the rest of the program).
- The name of the new relative pointer $\%r'$ being passed along is generated.
- The name of the new displacement variable $\%d'$ being passed along is generated. Notice that $\%d'$ is being assigned in the $NewCode$ generated, namely as:
  $\%d' = add(pm[\%e], \%i);$
- The generated relativized code is R followed by $NewCode$ which consists of the code to compute Δ, the code to compute $\%r'$ using a new GEP that works off the relativized address $\%r$ coming in.

- The generated code also carries an *assert* that checks for bit-flips (as said earlier, these asserts can be delayed till the "exit points" as illustrated in Section 1, Figure 1.

It is assumed that all other instruction types are ignored by the rule in Figure 3. Also, notice how *pm* is updated during code generation. The three shortcuts mentioned in Section 3.2 can be incorporated into the above generalized transition rule.

4 SEMI-FORMAL CHECKING OF FAILAMP

We check the correctness of the relativized pointers generated by FailAmp using the SMACK verifier [24]. SMACK works by translating LLVM code into an intermediate verification language called Boogie, which is then used for assertion checking using an SMT solver. SMACK is a *bounded* verifier, meaning it verifies programs with loops by unrolling them a statically bounded number of times. To apply SMACK on our benchmarks, we instrumented them as follows:

- We make all inputs to the kernel *nondeterministic* (i.e., unconstrained), meaning they can take any value. This differentiates verification from testing, where concrete values have to be provided.
- After every address computation from a relativized pointer, we insert an assertion that the computed value is the same as the non-relativized pointer.

This instrumentation allows us to verify the correctness of a transformed kernels pointer relativization.

4.1 Finding a Correctness Bug in FailAmp

When we first ran SMACK on the benchmark suite, we discovered an error in the way FailAmp handled GEP chains. This issue was encountered in vectorized-loop entry code, which was never executed in concrete test runs since the input arrays were aligned by malloc. Hence, testing failed to catch this error, and it was only revealed during our verification step. After correcting it, we uncovered an issue with SMACK's translation of variables modified in loops generated at higher optimization levels. Specifically, SMACK translated loop blocks which branch directly to a loop header and to another loop block to Boogie with updates happening before the branch. This caused SMACK to erroneously report errors related to address computation since indexing increments were performed before address calculation. The SMACK developers addressed this issue, permitting a more complete verification with SMACK.

We have verified all FailAmp transformed kernels with SMACK with a loop bound between 7 and 14, and for various optimization and vectorization levels. Some benchmarks take up to a day for SMACK to verify, but most complete in under 10 minutes.

Assertion Coverage: We also checked whether SMACK is able to reach each assertion generated by FailAmp. We accomplished this by negating a single assertion in the transformed kernel and running SMACK to find an assertion violation, thereby showing that SMACK is able to reach the assertion. We found that 90% of all assertions generated by FailAmp are reachable, and hence also verified, by SMACK. Our inspection of the unreached assertions shows they occur in blocks that may in fact not be reachable. Specifically, we found that such blocks are only reachable after an unsigned 64 bit indexing variable has overflowed. This insight proved valuable to us during the sanity-checking of the FailAmp algorithm.

In summary, our SMACK experiments have greatly enhanced our confidence in our FailAmp transformations. To gain additional confidence in FailAmp, we exhaustively checked within a small scope that all the injected faults are detected. This is discussed in Section 5.4.

Single Data Layout

Shortcut	2mm	3mm	adi	atax	bicg	cholesky	correlation	covariance	deriche	doitgen	durbin	fdtd_2d	floyd_warshall	gemm	gemver	gesummv	gramschmidt	heat_3d	jacobi_1d	jacobi_2d	lu	ludcmp	mvt	nussinov	seidel_2d	symm	syr2k	syrk	trisolv	trmm
None	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R1	9	9	5	6	0	0	31	30	26	5	3	18	1	5	12	0	27	2	0	4	7	11	5	6	0	5	8	9	0	9
R2	22	27	33	20	12	16	50	45	84	13	12	62	8	19	36	13	27	46	14	19	18	52	23	13	4	19	17	18	9	12
R3	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	2	0	3	0	0	0	0	1	0	0	0	0	0

Multi Data Layout

Shortcut	2mm	3mm	adi	atax	bicg	cholesky	correlation	covariance	deriche	doitgen	durbin	fdtd_2d	floyd_warshall	gemm	gemver	gesummv	gramschmidt	heat_3d	jacobi_1d	jacobi_2d	lu	ludcmp	mvt	nussinov	seidel_2d	symm	syr2k	syrk	trisolv	trmm
None	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R2	31	36	38	26	12	16	81	75	110	19	15	80	9	24	48	13	54	49	14	23	25	63	28	19	4	24	25	27	9	21
R3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	3	0	0	0	0	1	0	0	0	0	0

Fig. 4. GEP Shortcut Counts

5 EVALUATION

5.1 Benchmark Suite

To assess the performance of our transformation and test its correctness, we started with the full set of benchmarks in the PolyBench benchmark suite [23]. These were modified to allow two different representations of multi-dimensional arrays, named *single-* and *multi-* data layout. We use the term *single data layout* to refer to a multi-dimensional array laid out as a contiguous chunk: a two dimensional matrix in this form is indexed as `[x * width + y]`. We use the term *multi data layout* to refer to structures similar to `argv` (represented as a pointer array of arrays). This layout is indexed as `[x][y]`.

PolyBench contains benchmarks in four main categories; data mining, linear algebra, stencil computation, and medley. The linear algebra category is further broken down into BLAS operations, solvers, and kernels. There are 30 benchmarks in PolyBench and we run each with the two data layouts which leads to 60 benchmarks in total. All were used in our experimentation.

Two machine types were used for evaluation. Most experiments were run on the x86-64 instruction set. Specifically the machines were equipped with dual Intel Xeon E5-2680 v4 CPUs and 128 GB of memory. Some benchmarks were run on ARM based machines, these were ARMv8 APM X-GENE CPUs with 64 GB of memory.

5.2 GEPs transformed

In Figure 4, we conduct a detailed study of the GEP instructions transformed. Recall that the R3 shortcut relies on being able to determine that a GEP has a stride that is statically known. In such cases, the original GEP can be replaced with a GEP that uses a constant index. For example, consider the case where we are accessing every even index position. In this case, each access is displaced by 2 items from the last accessed location.

There are a total of 2048 static instances of the GEP instruction across our benchmarks. When transforming these, the first shortcut R1 was matched 253 times, the second shortcut R2 1781 times, and R3 12 times. Two GEPs did not fit any shortcut method. These GEPs are part of a vector alignment code section which is only run when argument arrays are not aligned in the benchmark called `durbin`. This study shows that shortcuts are the norm and not the exception, and protecting the introduced `add` instructions (occurs only 12 out of the 2048 cases), justifying the claim in Section 3.2 that runtime protection of the introduced instructions can incur fairly low overheads.

Overall Success in Detecting Strided Accesses: The application of the optimization to detect strided accesses is often limited by the ability to *statically* determine whether a calculation is strided. Unfortunately, depending on how a compiler generates code, this can often be obfuscated at the LLVM level. As an example, if a loop is unrolled and the indexing variable is known to be aligned to a multiple of the unroll, LLVM will often *or* the index with 1 instead of adding 1. Such logical operations can limit our ability to statically detect strided accesses.

5.3 Error Model

All resilience schemes work under a specific error model, and help detect instances of such errors during runtime. Our error model is that any of the GEP instructions employed can return a corrupted result different from the actual effective address to be returned by the same GEP. We assume that faults that corrupt the outcomes of other LLVM-level instructions are either not present, or are handled separately. These include arithmetic instructions, including the arithmetic instructions newly inserted by FailAmp to support its own activities.

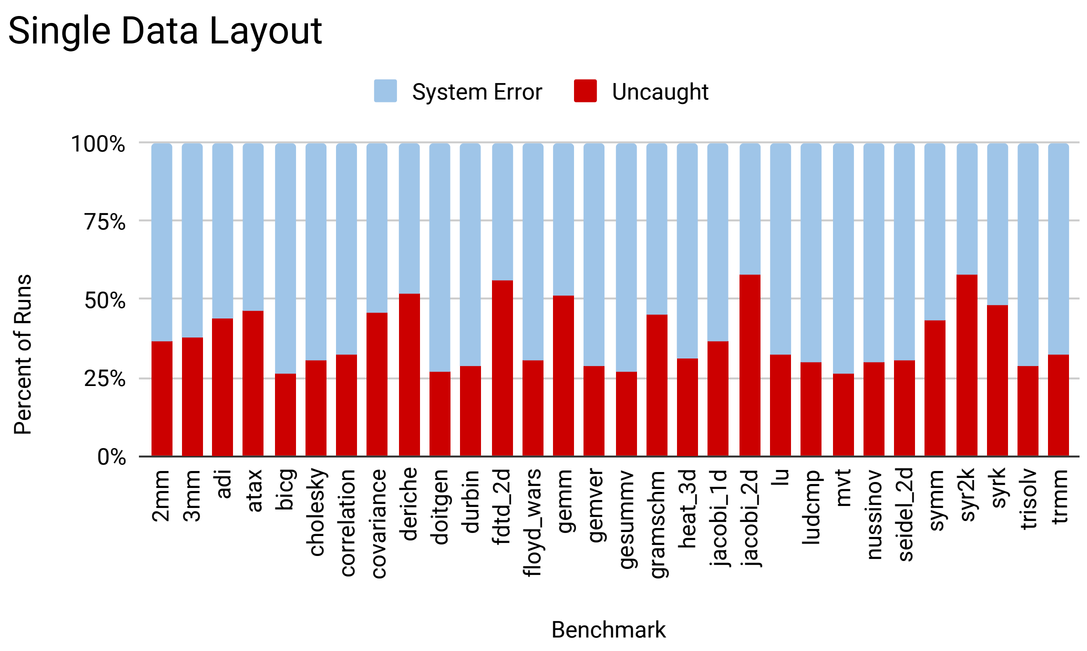
Given our focus on the protection of structured address generation, we focus on error impacting instructions that contribute to address generation. Precise analysis of the propagation of soft errors from microarchitectural state to the application can require expensive particle-beam experiments (e.g., [7, 27]) or time-consuming microarchitectural simulations. Some studies [6] point to the inadequacies in the space of fault injection at higher levels of abstraction. Later studies [4] have shown that random single-bit flips at lower levels of abstraction produced ensemble outcomes that are statistically similar to RTL-level error injection. To enable rigorous analysis of our strategy, we focus on LLVM-level fault injection studies. Given our specific target of coverage—namely the results of GEP calculations—we believe that this model is a reliable basis for use in our studies.

5.4 Fault Injection Based Validation of Relativization

To test that the detectors inserted by FailAmp were performing properly, we injected faults into running benchmarks using the LLVM based fault injector Vulfi [29]. Vulfi injects errors into computations at a controllable random rate, and it can target specific types of LLVM instructions as candidates for injection. We modified the runtime of Vulfi so that, instead of a random rate, a specific instruction can be fault injected. Faults were modelled as a single bit flip to any part of the resultant pointer of a memory address computation.

Results of a fault run fall into three categories: system error ("free" detections), FailAmp caught, and uncaught. System errors include segmentation violations and aborts caused by corruption of memory structures used by malloc. We performed three experiment types using this ability: (1) one where the benchmarks were run without FailAmp, and a random triple was selected for each run; (2) another where the benchmarks were run *with* FailAmp, and a random triple was selected for each run; (3) another run where the data sizes used were smaller, *with* FailAmp, *but every possible triple was enumerated.* This was done to gain further confidence in FailAmp.

For the random injection experiment type, the distribution of triples was chosen such that each runtime GEP invocation is equally likely. We performed 10,000 runs of each benchmark for both of the data layout types. Figure 5 is



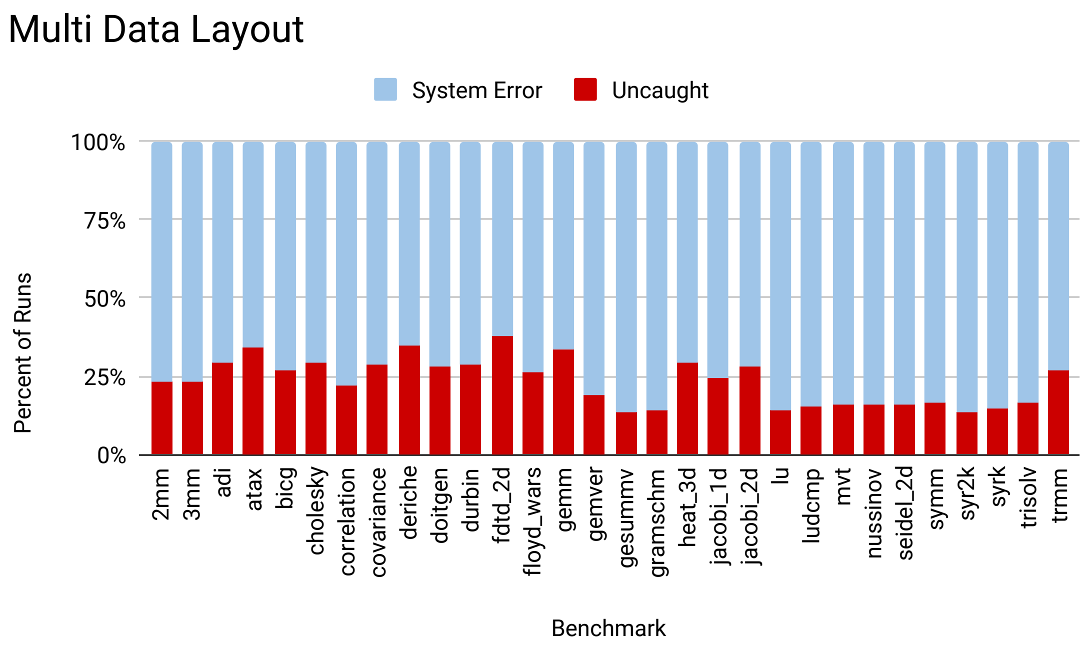
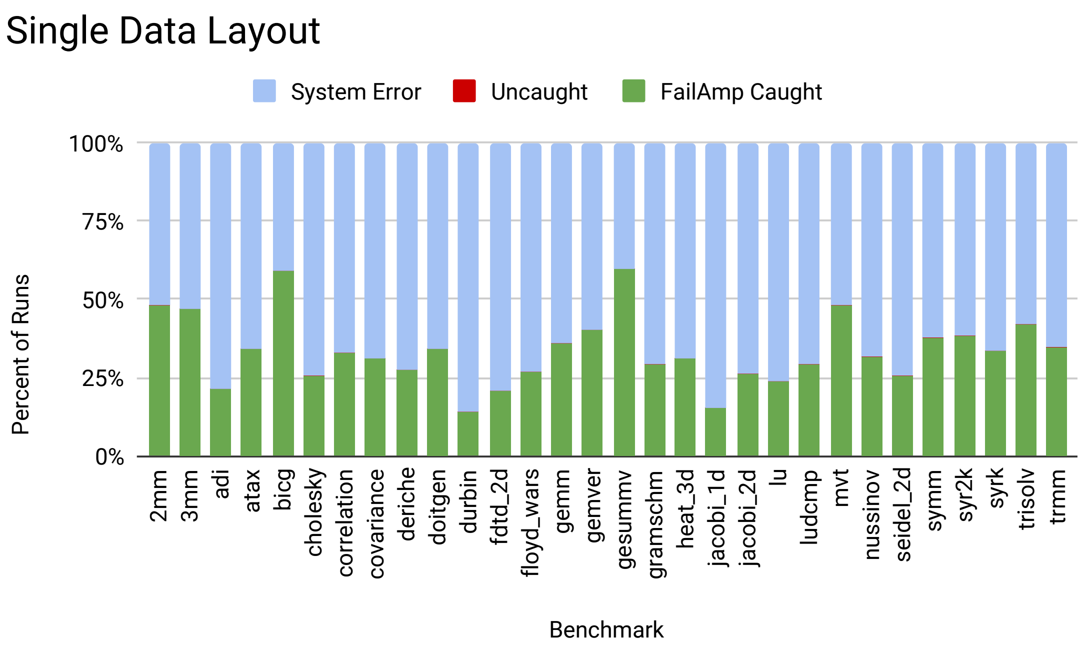


Fig. 5. Error rates of untransformed code for x86



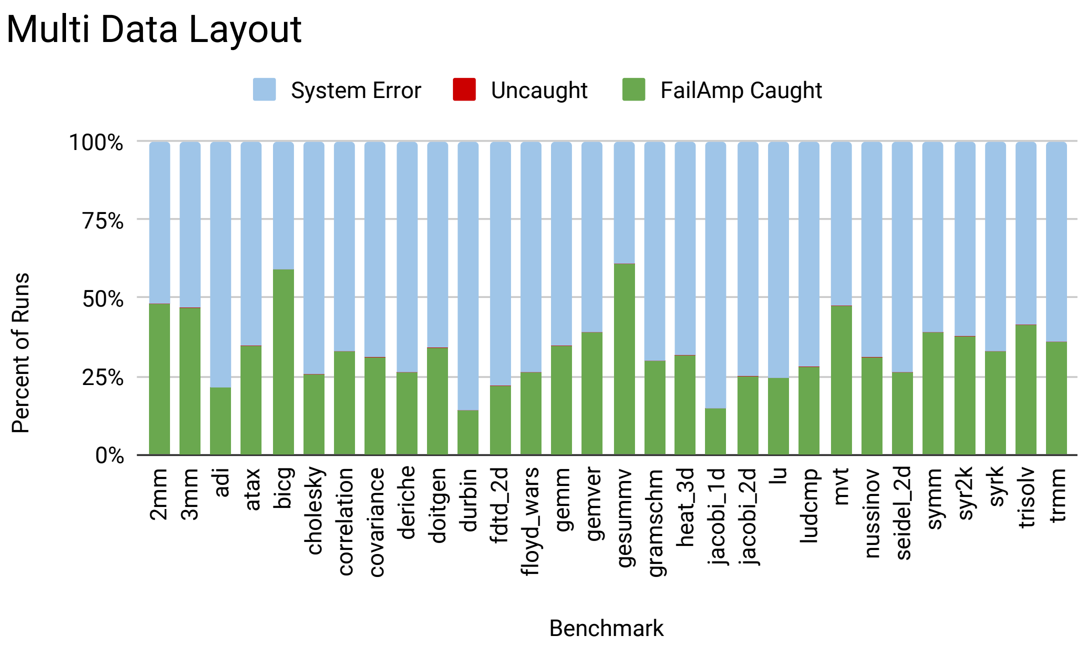


Fig. 6. Error rates of FailAmp transformed code for x86

the results of the injections on the unmodified code with both single and multi data layout. Both data layouts lead to the same distribution of result types for a given benchmark. When relativization is used, these distributions shift to the results seen in Figure 6.

Even for these sizes of input (PolyBench's medium sizes equates to a few megabytes of data) the system lets through about 30 percent of the injected errors. When detectors are used, either the fault is caught by the detector, or a system facility catches the fault before a detector is run. In all cases when FailAmp is used the fault is detected.

In our exhaustive enumeration of triples experiment, we further confirmed that all the GEPs were being protected. In all cases, either a crash occurred from a system error, or one of our detectors triggered. Notice that FailAmp might end up protecting even in those cases where a system failure might be triggered. This is important for many reasons including the impossibility of knowing whether a system failure will trigger, and supporting solutions such as microcheckpointing and compiler directed checkpointing [17]. In our studies, we found that in many cases system failures were triggered only when "free" was called—and in the interim, user data might be corrupted and rendered useless, thus preventing the possibility of any checkpoint/recovery method. Removing this lag time from fault to detection could allow application of more tightly coupled temporal and spatial checkpointing [13, 14, 36].

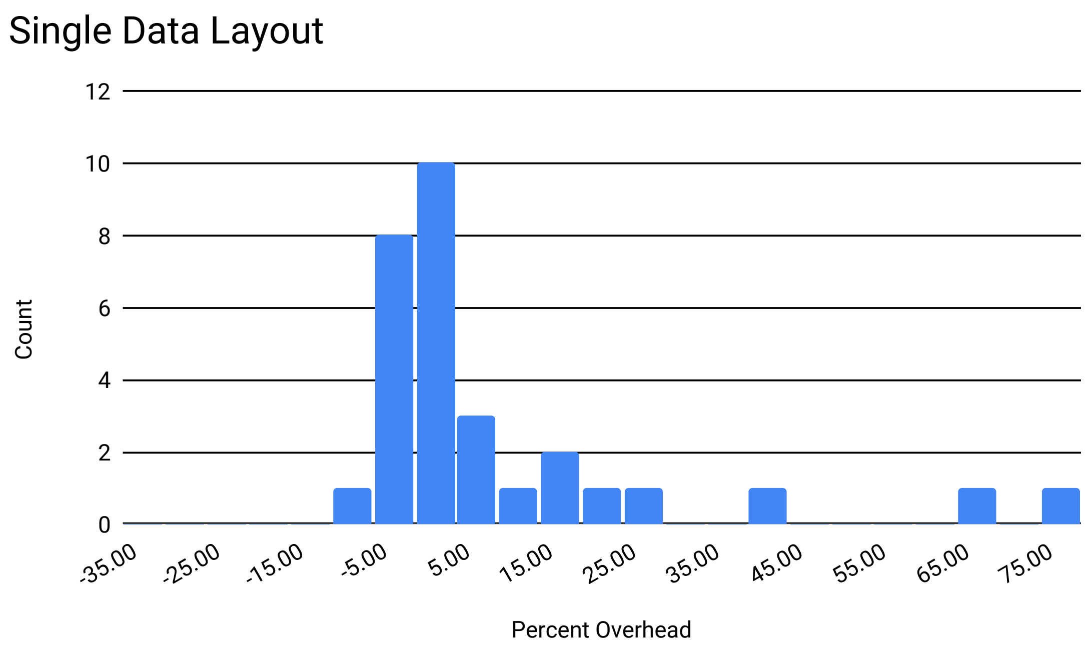
5.5 Analysis of Overheads

The overhead of FailAmp stems from two sources: (1) the extra integer operations before address calculations, and (2) the detector code. The detector overhead becomes proportionally smaller as the size of the data increases; this was also observed in our experiments. As the data size increases, more data will be processed before exiting the scope of a pointer, and so more items will be processed per detector. The extra integer operations will cause a constant overhead since they are used for every memory computation. The combined effect is that at small input sizes, there will be higher overhead, which drops asymptotically as the size increases to just the overhead of the extra integer operations.

From a machine level view, the indexing of arrays is done with three operations. First, the index is multiplied with the size of the array type, then the pointer value is added to this product, and finally, the resulting sum is used as the address for a load. Since this is such a common operation, it has been wrapped up as a single instruction in x86, namely the complex `mov` instruction. This instruction performs all three steps at once, *but does not allow access to the intermediates of the calculation.* For our relativization, we need access to the sum value used for the load, so that we can use it for our next address computation. Luckily, the first two steps have also been packaged in x86 as the Load Effective Address, `lea`, instruction. So, it seems we should be able to perform this relativization while only adding one to two integer instructions for the delta calculation to each memory operation.

A problem arises when looking at restrictions on the `lea` instruction. If an `lea` uses a pointer located in `%ebp`, `%RBP`, or `%R13` it will incur an additional three cycle latency. If an `lea` uses all three source operands it will incur the same latency. If all source operands are used it must be dispatched on port one. If a source operand is coming from an execution unit a three cycle delay will occur [15]. All these restrictions mean that we will often be meeting one or more of the criteria and have the address calculation delayed. This can be an imperceptible difference for code which is computation heavy, but is a big downside for memory intensive code.

Figure 7 shows histograms of the overhead of using the FailAmp transformation. Nine (9) benchmarks in the single layout actually speed up when using FailAmp. Overall, there is a tight grouping at the low end of the overhead plot, with a few outliers. Reassuringly, the outliers were benchmarks which had sub-second runtimes, where a 50% overhead is obtained for runs that take a second of compute time. In particular, these outliers include the kernels `jacobi_1d`,



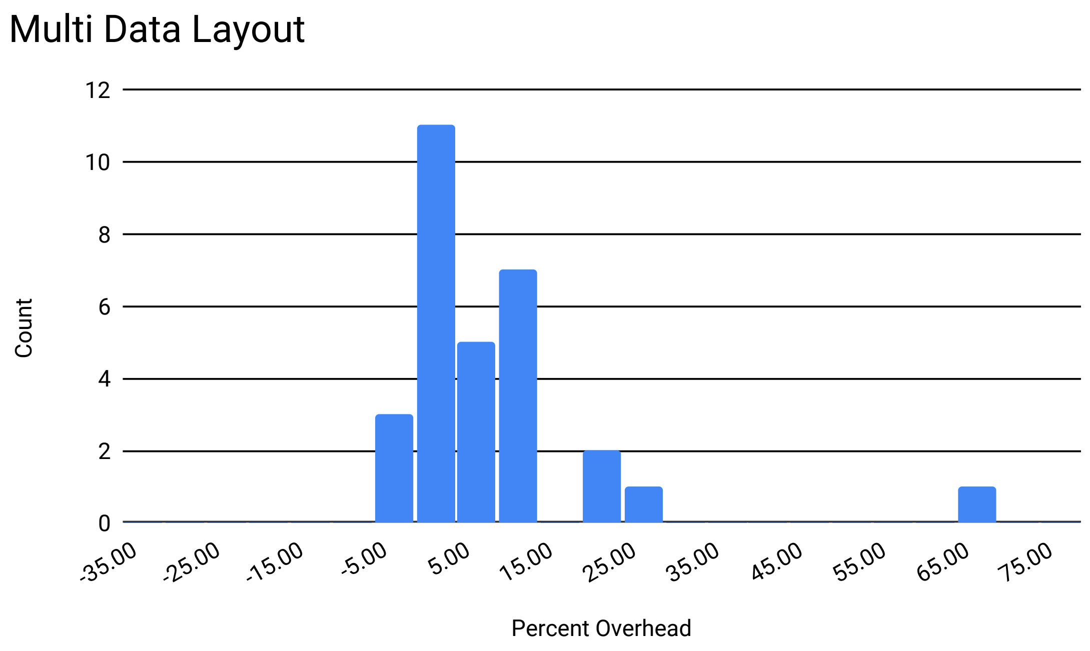


Fig. 7. Overhead histograms for x86

durbin, and mvt which took 5 milliseconds, 40 milliseconds, and 500 milliseconds to execute. This further reassures us that for realistic sizes, the overhead of FailAmp can be acceptably small.

5.6 Exploiting the ARM ISA to Reduce FailAmp Overhead

The ARM instruction set also has instructions that help combine the steps of indexing and loading. The standard model of computing the address to load and then performing the load is present in ARM, and called offset addressing. There are also two modes, called pre- and post-indexing modes, which modify the pointer upon indexing. These modes essentially update the register holding the pointer to the base plus offset value, while also performing the load. They are different only with respect to when the load occurs: pre-indexing loads the pointer then calculates the base plus offset, whereas post-indexing calculates the base plus offset prior to the load. *The post-indexed addressing mode does exactly what we want.* Relativizing accesses has been baked into the ISA, so we should see lower overheads for ARM since the only cost is the additional integer instructions.
Assessment of Overall Success of ARM-specific Optimizations: A small number of tests show a speedup when using ARM than was seen for x86, with an outlier showing a 25 to 35 percent speedup, as seen in figure 8. This speedup was achieved on the seidel_2d benchmark with both single and multi data layouts, and dropped the runtime from 26 minutes normally to 18 minutes with FailAmp. While these fast outliers are nice, the overall grouping is not quite what we predicted. A probable reason for this relates to the exact hardware being used. The AppliedMicro X-Gene processor [11] was used to run the benchmarks, and it only contains two ports for integer operations. The overhead of our additional integer instructions is being amplified when these ports are full. In an ARM CPU with a wider execution path, this problem would perhaps be alleviated.

5.7 Case study: LULESH

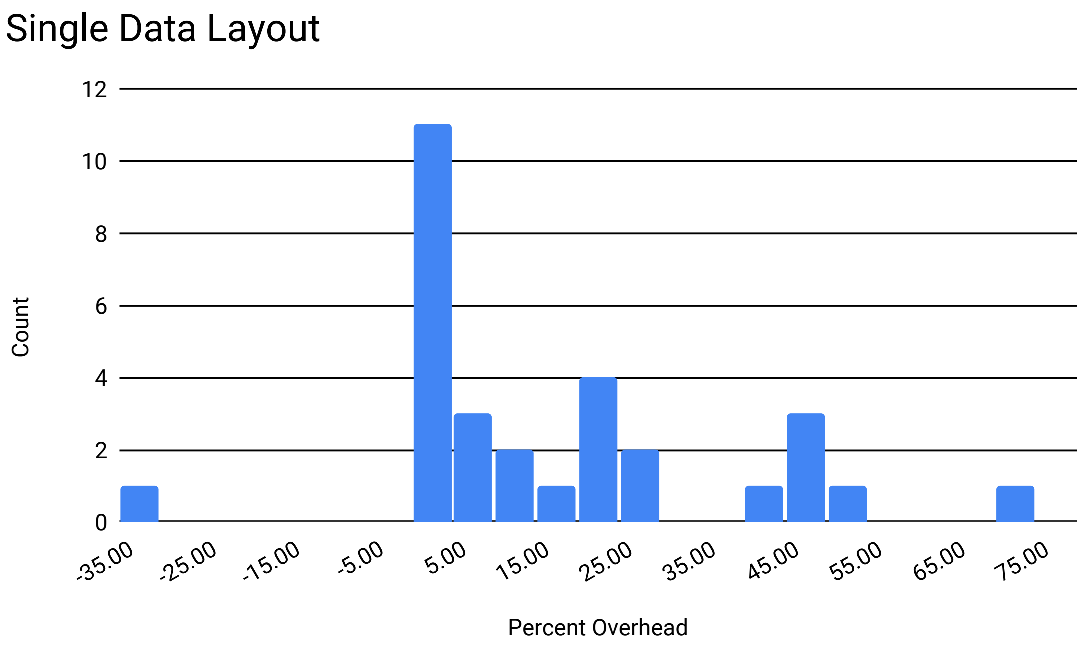
LULESH is the Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics miniapp which is representative of numerical algorithms, data motion, and programming style typical in scientific C or C++ based applications [16]. We utilized the FailAmp transformation on LULESH targeting the computation functions which it employs, and tested the resulting miniapp for both error detection and overhead.

We performed $10,000$ random error injections to both the transformed and untransformed applications. When injecting errors into the untransformed application the rate of "system detection" is similar to what we obtained with PolyBench. That is, without FailAmp, 71% of the faults will result in "system detection" but 29% will slip by uncaught. When FailAmp is used, *all errors* are detected, and we observed that the system detection rate lowers to 67%, with 33% being picked up by FailAmp.

Next, we ran both applications (with and without FailAmp) 100 times to determine FailAmp's overhead. The default size of simulation for LULESH is a cube with side length of 32. Even at this small data size, there is only a 13% overhead for FailAmp, with a 3% standard deviation. Moving to a side length of 64 yields a drop to 11% overhead. Increasing the size again to 96 brings the overhead to 6%. This study is the final acid-test that FailAmp deserves strong consideration for HPC code, given its 100% address error detection rate and around 5% overhead.

5.8 Analysis of Robustness

5.8.1 Effect of Polyhedral and Tiling Optimizations. Given that polyhedral optimizations are important code optimizations to improve locality, and are widely applied to our class of programs, our first goal is to ensure that FailAmp's



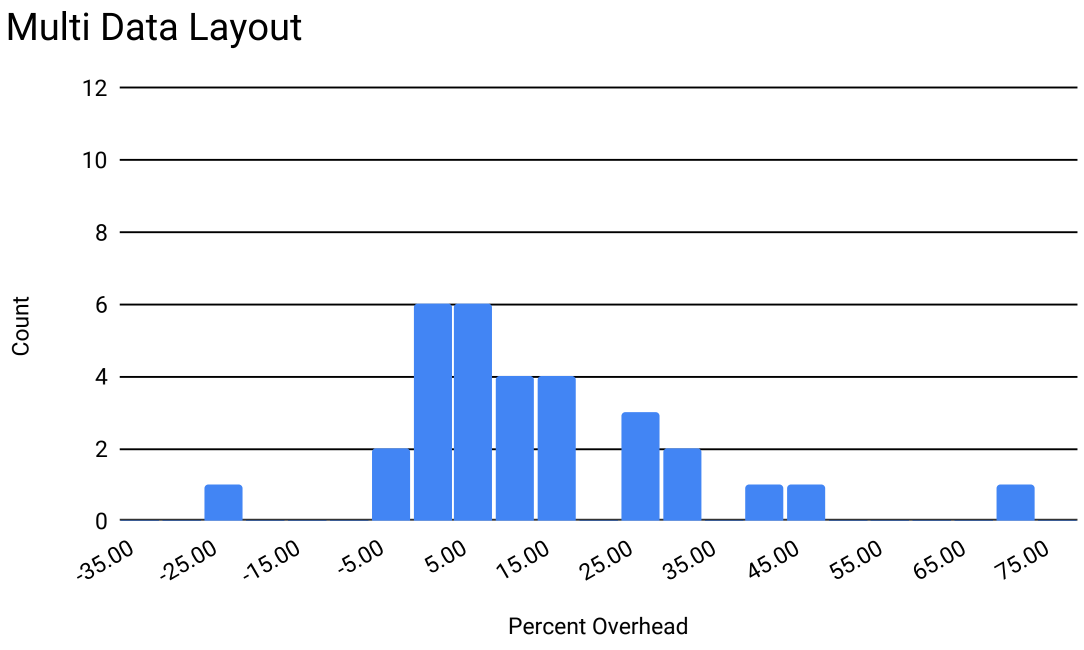


Fig. 8. Overhead histograms for ARM

advantages are retained in the presence of these optimizations. Polyhedral optimizations are algebraic transformations applicable whenever nested loops contain statically predictable control flows. Such loop nests are called *static control parts*, characterized by a collection of statements where the loop bounds and conditional expressions are affine functions of the loop iterators as well as compile-time constants. We employ Pluto [2], an automatic parallelizer and locality optimizer for affine loop nests to generate a polyhedrally optimized version of our primary benchmark suite. Pluto discovers and applies affine transformations leading to efficient tiling, thus accruing the advantages of coarse-grained parallelism and data locality.

5.8.2 Pluto Results Discussion. It is a known fact that Pluto-optimized code suffers not only code-bloat but also a significant increase in the number of basic blocks constituting the main kernel. For the "single" data layout, it requires a single detector per argument pointer to be invoked. Hence, FailAmp transformation and detector invocation do not incur significant overheads with respect to the Pluto-unoptimized version. In case of the multi dimensional layout of arrays, a detector invocation is exercised for each trip through the innermost loop (pointer goes out of scope). Correspondingly, the Pluto optimized version tends to increase the invocation frequency of the detectors for this case (due to dependent pointers). Hence, we see a general increase in the overhead cost with the multi-layout in Figure 9. Even then, polyhedral codes with large tile sizes (1024) tend to plateau out with respect to overheads.

We did verify that our guarantees of error detection hold even for FailAmp transformations applied to Pluto-optimized code. Pluto did not change FailAmp's ability to detect faults.

6 RELATED WORK

System resilience research is gaining momentum; space does not permit an exhaustive survey. In this section, we briefly discuss efforts closely related to ours, and point out key challenges we overcame to make relativization work correctly and efficiently, spanning multiple application classes.

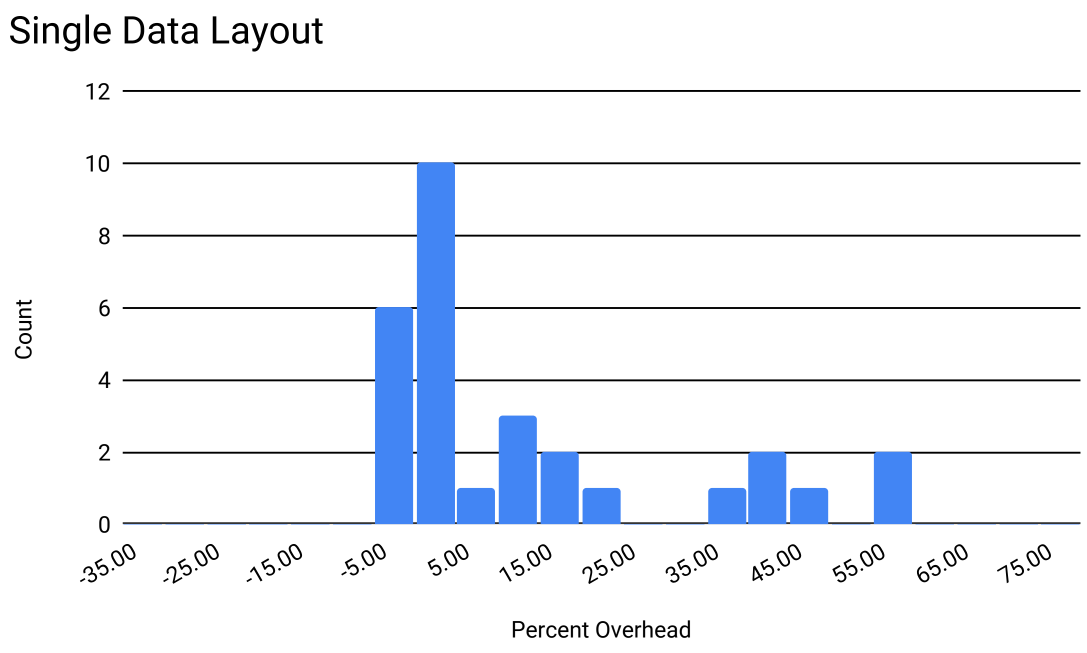
Error detection strategies through value tracking have gained some attention. Precisely detecting silent data corruptions presents a challenging problem. The work in [8] presents an impact driven model for SDC detection with an objective to reduce false alarms trading off with acceptable SDC rates. The work in [33] explores the usage of support vector machine techniques for SDC detection with low memory overhead costs for HPC applications. However, these methods target the data part of computation by through value tracking and do not target address generation.

Error detection through redundant execution has also been pursued in many works. The work in [22] attempts to minimize the number of dynamic instructions executed for protection. Instead of executing an expensive redundant computation, the re-computation output is approximated to validate the computed results. While these approaches are often quite effective, they require additional hardware or incurs higher costs. The work in [31] proposes an algorithmic approach to correct faulty application outputs through partial recomputation. The work in [34] proposes an algorithm based fault tolerance (ABFT) scheme for soft error detection and recovery methods for iterative solvers.

The work in [38] performs soft-error detection in microprocessors based on symptoms that hint at the presence of soft errors including recovering from an earlier checkpoint. Symptom based recovery might detect errors in terms of anomalies but do not guarantee detection.

The work closest to ours is that reported in [28] where the authors propose a basic relativization transformation for indexing error detection. Our work differs in the following respects:

- The work in [28] handles only single dimensional arrays. It cannot handle the full generality of types we handle.



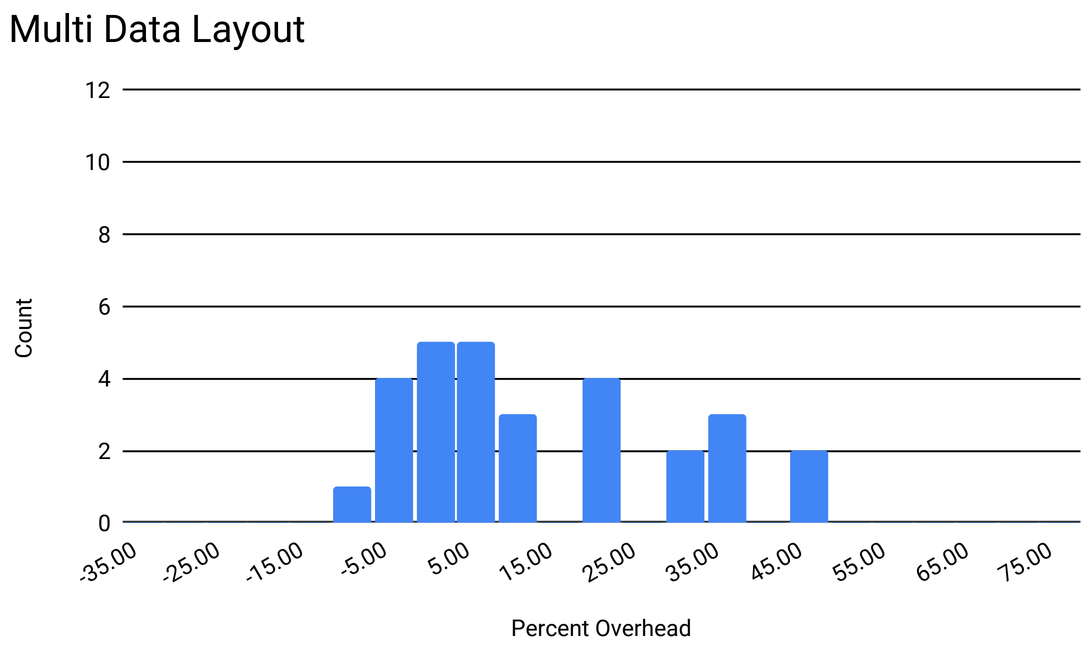


Fig. 9. Overhead histograms for x86 while usin Pluto

- The error detection approach in [28] is based on loop exit conditions obtained through Hoare axioms. Their work cannot accommodate loops with unknown iteration ranges. Our work can handle *any* LLVM-based flow-graph (even non single-entry/exit graphs), as we flow enough information for each index to check whether its relativized and absolute indices agree.

In contrast, FailAmp (1) generalizes the data types handled to arbitrary nests of arrays and records (see Figure 1's grammar), (2) handles optimizations such as vectorization that chains GEPs, and (3) exploits pointer-comparison (not Hoare axioms) for error detection. We additionally provide extensive benchmarking, robustness analysis, formal analysis, and insights on how to exploit architectural features (x86 versus ARM).[2]

Relativization transformations within a local are often done by compilers to improve code quality. However, compilers typically only relativize when there is a known stride and when there is an estimated performance improvement (thanks to the known stride). This allows compilers employing this optimization to omit runtime delta index calculations, since each delta is the same, and a compile-time constant. This relativization is not performed when the access pattern is not known at compile time.

The area of LLVM transformations is quite active, with transformations developed for one purpose often playing a useful role for another. In this vein, our work can be viewed as a *formally validated LLVM translation of interest in its own right.* We do not yet know where else such relativization that spans basic blocks and loops will find a role; but putting FailAmp out can only help add to the repertoire of LLVM transformations out there (especially because, as Section 5 shows, that after relativization the performance may even improve).

7 DISCUSSIONS, CONCLUDING REMARKS

Soft errors threaten the integrity of scientific results produced by long-running simulations—especially for new explorations where there are often very few (if any) cost-effective alternative checks (for example, the simulation of binary black holes [9]). Any method that enhances system resilience with low overheads is well suited for such application domains. Solutions tailored toward structured address generation address a crucial vulnerability window where a series of addresses computations occur in long-running loops. Our experiments demonstrate that faults in these address calculations often do not stray outside of legal address ranges—and hence, in effect, *turn into silent data corruptions* of a fairly egregious kind, where the data fetched under the corrupted address may be far removed from the data fetched under the absence of faults. In contrast, data corruptions suffered by ALU calculations—the traditional mainstay of system resilience research—often tend to resemble exaggerated rounding results. Our focus on complex address generation was directly responsible for our point solution being efficient and precise. We leave the task of protection against other error types (e.g., memory read/write errors) to other complementary research efforts.

Our approach of *amplifying* errors is unusual, in contrast with other efforts where the attempt is often to automatically *repair* errors and move on. Such repair is known to be extremely difficult to achieve. In contrast, the FailAmp approach allows computations to revert to earlier checkpoints, repair and move on. This approach actually fits well within microcheckpointing and recovery methods that are central to exascale computing [37].

In this paper, we extensively validate the algorithm implemented in FailAmp both empirically as well as formally. We also demonstrated FailAmp's acceptable overheads on a number of benchmarks, and also showed that these benefits are unaffected by polyhedral transformations.

[2]We made a concerted effort to resurrect the PRESAGE code and benchmark FailAmp Against it. This effort failed due to "bit rot" in the PRESAGE code-base.

Our work also involved interesting challenges in the architecture and operating system spaces in a compilation setting. We showed promise of exploiting special features of ISAs such as ARM to reduce the overhead of relativization. During our experiments, we had to turn off address-space randomization (ASLR) to provide repeatable measurements with respect to crashes ("system caught errors" in our studies). Thus, the pursuit of relativization offers many intellectual challenges cutting through the system stack.

There are many interesting research questions to be further pursued in this space. Questions such as the design of modern AGUs, and whether they provide value-forwarding (thus helping hide the overhead of relativization) are important to further consider. It appears that relativization does not sequentialize the AGU operation itself, as AGUs can be busy fetching across different address streams. It also appears that relativization does not limit the ability of the compute core to exploit instruction level parallelism. Memory allocation-wise, it is known that the "array of array" layout for multidimensional arrays is not as efficient as single linearly allocated multidimensional arrays. This picture does not appear to change with relativization. These questions are worth answering backed with actual measurements. The code of FailAmp is in a stable form and will be released along with the final version of this paper.

ACKNOWLEDGMENTS

This research was supported in part by NSF Awards 1817073 and 1704715, and DOE Contract DE-SC0014096. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under Award Number 66905. Pacific Northwest National Laboratory is operated by Battelle for DOE under Contract DE-AC05-76RL01830.

REFERENCES

[1] Rizwan 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 2015, Austin, TX, USA, November 15-20, 2015*, Jackie Kern and Jeffrey S. Vetter (Eds.). ACM, 72:1–72:12. https://doi.org/10.1145/2807591.2807670

[2] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. 2008. Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model. In *International Conference on Compiler Construction (ETAPS CC)*.

[3] Franck Cappello, Geist Al, William Gropp, Sanjay Kale, Bill Kramer, and Marc Snir. 2014. Toward Exascale Resilience: 2014 Update. *Supercomput. Front. Innov.: Int. J.* 1, 1 (April 2014), 5–28. https://doi.org/10.14529/jsfi140101

[4] 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 2018, Dallas, TX, USA, November 11-16, 2018*. IEEE / ACM, 45:1–45:13. http://dl.acm.org/citation.cfm?id=3291716

[5] S. Chen, L. Peng, and G. Bronevetsky. 2015. *A Framework For Evaluating Comprehensive Fault Resilience Mechanisms In Numerical Programs*. Technical Report LLNL-SR-666073.

[6] Hyungmin Cho, Shahrzad Mirkhani, Chen-Yong Cher, Jacob A. Abraham, and Subhasish Mitra. 2013. Quantitative Evaluation of Soft Error Injection Techniques for Robust System Design. In *Proceedings of the 50th Annual Design Automation Conference (DAC '13)*. ACM, New York, NY, USA, Article 101, 10 pages. https://doi.org/10.1145/2463209.2488859

[7] Daniel A. G. de Oliveira, Laércio Lima Pilla, Nathan DeBardeleben, Sean Blanchard, Heather Quinn, Israel Koren, Philippe O. A. 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 2017, Denver, CO, USA, November 12 - 17, 2017*, Bernd Mohr and Padma Raghavan (Eds.). ACM, 28:1–28:12. https://doi.org/10.1145/3126908.3126960

[8] Sheng Di and Franck Cappello. 2016. Adaptive Impact-Driven Detection of Silent Data Corruption for HPC Applications. *IEEE Trans. Parallel Distrib. Syst.* 27, 10 (2016), 2809–2823. https://doi.org/10.1109/TPDS.2016.2517639

[9] Milinda Fernando, David Neilsen, Hyun Lim, Eric Hirschmann, and Hari Sundar. 2019. Massively Parallel Simulations of Binary Black Hole Intermediate-Mass-Ratio Inspirals. *SIAM Journal on Scientific Computing 2019* 41, 2 (2019), C97–C138. Also https://arxiv.org/abs/1807.06128.

[10] Ganesh Gopalakrishnan, Paul D. Hovland, Costin Iancu, Sriram Krishnamoorthy, Ignacio Laguna, Richard A. Lethin, Koushik Sen, Stephen F. Siegel, and Armando Solar-Lezama. 2017. Report of the HPC Correctness Summit, Jan 25-26, 2017, Washington, DC. *CoRR* abs/1705.07478 (2017). arXiv:1705.07478 http://arxiv.org/abs/1705.07478

[11] P. Gopi, G. Singh, and G. Favor. 2012. X-Gene™: 64-bit ARM CPU and SoC. In *2012 IEEE Hot Chips 24 Symposium (HCS)*. 1–19. https://doi.org/10.1109/HOTCHIPS.2012.7476502

[12] Saurabh Gupta, Tirthak Patel, Christian Engelmann, and Devesh Tiwari. 2017. Failures in Large Scale Systems: Long-term Measurement, Analysis, and Implications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*. ACM, New York, NY, USA, Article 44, 12 pages. https://doi.org/10.1145/3126908.3126937

[13] Saurabh Gupta, Devesh Tiwari, Christopher Jantzi, James Rogers, and Don Maxwell. 2015. Understanding and Exploiting Spatial Properties of System Failures on Extreme-Scale HPC Systems. In *Proceedings of the 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '15)*. IEEE Computer Society, Washington, DC, USA, 37–44. https://doi.org/10.1109/DSN.2015.52

[14] Zaeem Hussain, Taieb Znati, and Rami Melhem. 2018. Partial Redundancy in HPC Systems with Non-uniform Node Reliabilities. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '18)*. IEEE Press, Piscataway, NJ, USA, Article 44, 11 pages. https://doi.org/10.1109/SC.2018.00047

[15] Intel. 2016. Intel 64 and IA-32 Architectures Optimization Reference Manual. *Order Number: 248966-033* 25 (2016).

[16] Ian Karlin, Jeff Keasler, and Rob Neely. 2013. *LULESH 2.0 Updates and Changes*. Technical Report LLNL-TR-641973. 1–9 pages. https://computation.llnl.gov/projects/co-design/lulesh.

[17] Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. 2016. Compiler-directed Lightweight Checkpointing for Fine-grained Guaranteed Soft Error Recovery. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16)*. IEEE Press, Piscataway, NJ, USA, Article 20, 12 pages. http://dl.acm.org/citation.cfm?id=3014904.3014931

[18] LLVM Language Reference Manual [n.d.]. LLVM Language Reference Manual. http://llvm.org/docs/LangRef.html#getelementptr-instruction.

[19] Shubhendu S Mukherjee, Christopher Weaver, Joel Emer, Steven K Reinhardt, and Todd Austin. 2003. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*. IEEE, 29–40.

[20] B. Nie, D. Tiwari, S. Gupta, E. Smirni, and J. H. Rogers. 2016. A large-scale study of soft-errors on GPUs in the field. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 519–530. https://doi.org/10.1109/HPCA.2016.7446091

[21] Vojin G. Oklobdzija. 2001. *The Computer Engineering Handbook: Electrical Engineering Handbook*. CRC Press, Inc., Boca Raton, FL, USA.

[22] Sunghyun Park, Shikai Li, and Scott A. Mahlke. 2018. Low Cost Transient Fault Protection Using Loop Output Prediction. In *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops, DSN Workshops 2018, Luxembourg, June 25-28, 2018*. 109–113. https://doi.org/10.1109/DSN-W.2018.00047

[23] polybench [n.d.]. PolyBench/C: The Polyhedral Benchmark suite. http://web.cs.ucla.edu/~pouchet/software/polybench/.

[24] Zvonimir Rakamarić and Michael Emmi. 2014. SMACK: Decoupling Source Language Details from Verifier Implementations. In *Proceedings of the 26th International Conference on Computer Aided Verification (CAV) (Lecture Notes in Computer Science)*, Vol. 8559. Springer, 106–113.

[25] Jude A. Rivers, Meeta S. Gupta, Jeonghee Shin, Prabhakar N. Kudva, and Pradip Bose. 2011. Error Tolerance in Server Class Processors. *in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (CADICS)* 30, 7 (2011), 945–959.

[26] Pia N. Sanda, Jeffrey W. Kellington, Prabhakar Kudva, Ronald N. Kalla, Ryan B. McBeth, Jerry Ackaret, Ryan Lockwood, John Schumann, and Christopher R. Jones. 2008. Soft-error resilience of the IBM POWER6 processor. *IBM Journal of Research and Development* 52, 3 (2008), 275–284. https://doi.org/10.1147/rd.523.0275

[27] Norbert Seifert, Vinod Ambrose, B Gill, Q Shi, R Allmon, C Recchia, S Mukherjee, N Nassif, J Krause, J Pickholtz, et al. 2010. On the radiation-induced soft error performance of hardened sequential elements in advanced bulk CMOS technologies. In *Reliability Physics Symposium (IRPS), 2010 IEEE International*. IEEE, 188–197.

[28] Vishal Chandra Sharma, Ganesh Gopalakrishnan, and Sriram Krishnamoorthy. 2016. PRESAGE: Protecting Structured Address Generation against Soft Errors. In *23rd IEEE International Conference on High Performance Computing, HiPC 2016, Hyderabad, India, December 19-22, 2016*. 252–261. https://doi.org/10.1109/HiPC.2016.037

[29] Vishal Chandra Sharma, Ganesh Gopalakrishnan, and Sriram Krishnamoorthy. 2016. Towards Resiliency Evaluation of Vector Programs. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2016, Chicago, IL, USA, May 23-27, 2016*. IEEE Computer Society, 1319–1328. https://doi.org/10.1109/IPDPSW.2016.187

[30] Vishal C. Sharma, Ganesh Gopalkrishnan, and Greg Bronevetsky. 2015. Detecting Soft Errors in Stencil based Computations. http://formalverification.cs.utah.edu/fmr/. In *Workshop on Silicon Errors in Logic – System Effects (SELSE)*. Austin, TX.

[31] Joseph Sloan, Rakesh Kumar, and Greg Bronevetsky. 2013. An algorithmic approach to error localization and partial recomputation for low-overhead fault tolerance. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Budapest, Hungary, June 24-27, 2013*. 1–12. https://doi.org/10.1109/DSN.2013.6575309

[32] Marc Snir, Robert W. Wisniewski, Jacob A. Abraham, Sarita V. Adve, Saurabh Bagchi, et al. 2014. Addressing failures in exascale computing. *IJHPCA* 28, 2 (2014), 129–173.

[33] Omer Subasi, Sheng Di, Leonardo Bautista-Gomez, Prasanna Balaprakash, Osman S. Ünsal, Jesús Labarta, Adrián Cristal, and Franck Cappello. 2016. Spatial Support Vector Regression to Detect Silent Errors in the Exascale Era. In *IEEE/ACM 16th International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2016, Cartagena, Colombia, May 16-19, 2016*. 413–424. https://doi.org/10.1109/CCGrid.2016.33

[34] Dingwen Tao, Shuaiwen Leon Song, Sriram Krishnamoorthy, Panruo Wu, Xin Liang, Eddy Z. Zhang, Darren Kerbyson, and Zizhong Chen. 2016. New-Sum: A Novel Online ABFT Scheme For General Iterative Methods. In *International Symposium on High-Performance Parallel and Distributed*

Computing (HPDC). 43–55.

[35] Sanket Tavarageri, Sriram Krishnamoorthy, and P. Sadayappan. 2014. Compiler-assisted detection of transient memory errors. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. 204–215. https://doi.org/10.1145/2594291.2594298

[36] D. Tiwari, S. Gupta, and S. S. Vazhkudai. 2014. Lazy Checkpointing: Exploiting Temporal Locality in Failures to Mitigate Checkpointing Overheads on Extreme-Scale Systems. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 25–36. https://doi.org/10.1109/DSN.2014.101

[37] Augusto Vega, Pradip Bose, and Alper Buyuktosunoglu. 2016. *Rugged Embedded Systems: Computing in Harsh Environments* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[38] Nicholas J. Wang and Sanjay J. Patel. 2006. ReStore: Symptom-Based Soft Error Detection in Microprocessors. *IEEE Trans. Dependable Sec. Comput.* 3, 3 (2006), 188–201. https://doi.org/10.1109/TDSC.2006.40