Compiler-Assisted Detection of Transient Memory Errors

Sanket Tavarageri
Dept. of Computer Science and Engg.
The Ohio State University
tavarage@cse.ohio-state.edu

Sriram Krishnamoorthy
Comp. Sci. & Math. Division
Pacific Northwest National Lab
sriram@pnnl.gov

P. Sadayappan
Dept. of Computer Science and Engg.
The Ohio State University
saday@cse.ohio-state.edu

Abstract

The probability of bit flips in hardware memory systems is projected to increase significantly as memory systems continue to scale in size and complexity. Effective hardware-based error detection and correction require that the complete data path, involving all parts of the memory system, be protected with sufficient redundancy. First, this may be costly to employ on commodity computing platforms, and second, even on high-end systems, protection against multi-bit errors may be lacking. Therefore, augmenting hardware error detection schemes with software techniques is of considerable interest.

In this paper, we consider software-level mechanisms to comprehensively detect transient memory faults. We develop novel compile-time algorithms to instrument application programs with checksum computation codes to detect memory errors. Unlike prior approaches that employ checksums on computational and architectural states, our scheme verifies every data access and works by tracking variables as they are produced and consumed. Experimental evaluation demonstrates that the proposed comprehensive error detection solution is viable as a completely software-only scheme. We also demonstrate that with limited hardware support, overheads of error detection can be further reduced.

Categories and Subject Descriptors D.3.4 [*Programming languages*]: Processors — Compilers; B.3.4 [*Memory structures*]: Reliability, Testing, and Fault-Tolerance — Error-checking

General Terms Performance, Reliability

Keywords Transient memory errors; def-use tracking; checksums

1. Introduction

Trends in technology scaling have increased the likelihood of transient faults in various hardware components due to particle strikes [3, 5, 30, 34] and environmental factors [24, 44]. The simultaneous drive to reduce power consumption has led to the use of lower voltage levels and smaller noise margins, which further increase a system's susceptibility to transient faults [8]. Such transient faults in the memory subsystem can result in undetected bit flips and potentially lead to silent data corruption.

PLDI'14, June 9–11, 2014, Edinburgh, United Kingdom.

http://dx.doi.org/10.1145/2594291.2594298

Hardware approaches for detecting and correcting bit flips in the memory system employ error correcting codes on various components of the data path. These codes are checked on every data access and updated on each modification. Comprehensive error detection requires every component of the data path—reorder buffers, caches, memory lines, buses, etc.—to support sufficient redundancy or parity to detect the errors anticipated during execution. Such a design requires preallocation of hardware resources, incurring dynamic power and latency costs to support the worst-case fault scenario to be tolerated. Nevertheless, commodity computing platforms with less-protected memory subsystems may have to contend with faults for certain critical computation phases. Even custom computing platforms more cognizant of soft errors might not provide the same level of protection in all components of the memory subsystem. Examples include L1 caches in BlueGene/L [20], device memory on pre-Fermi Nvidia GPUs [25], and pre-Tahiti AMD GPUs with parities. Multi-bit errors often can evade hardware detection mechanisms and potentially lead to silent data corruption. In addition to multiple bit flips in stored data, an error in the addressing logic in the memory subsystem, including in address generation, might result in an incorrect address and be perceived as a multi-bit error. More importantly, the fault scenarios encountered in practice might not always match the fault models and projections assumed in the design phase.

In this paper, we consider software-level approaches to comprehensive detection of multi-bit errors in the memory subsystem. Such approaches can complement hardware schemes to further improve system resilience. Redundant execution of memory operations, which duplicates all variables of interest and operations on them, can be used to detect these errors in the memory subsystem. However, this basic approach significantly increases memory space and bandwidth requirements. We study the feasibility of detecting errors by employing error detection codes for the definition and every use of variables. This approach has the potential to comprehensively detect errors due to faults in any architectural state in the memory subsystem.

We present a compiler-assisted approach to augmenting the definitions and uses in a given program with checksums to detect memory errors. We present optimizations that minimize the overhead for common classes of computations: affine loops and iterative computations that could be irregular. Experimental evaluation demonstrates that the performance costs are low enough for the approach to be practical, and that we achieve excellent fault coverage for the checksum operator considered. We also demonstrate that overheads can be reduced further with hardware support in the processor (without affecting or altering the memory subsystem) to compute the checksums.

The primary contributions of this paper are:

- An algorithm to detect memory errors by augmenting definitions and uses of values with checksum computation operations

- Compile-time optimizations to make this approach more efficient in two common classes of computations
- A discussion and evaluation of the checksum operator in terms of performance and fault coverage
- Novel proposals to increase fault coverage via use of multiple checksums
- Experimental demonstration of the low overheads and feasibility of the approach.

2. Error Detection using Checksums

Typical approaches to protecting data elements in hardware and software group data elements and employ a checksum for each group. These checksums typically are checked on every access. Algorithm-based fault tolerance approaches employ checksums for matrices that are maintained as part of the algorithm execution. All of these schemes attempt to associate checksums with data elements. This requires additional steps to maintain the checksum for every operation on the data element. In architectural checksumming instantiations, this requires every storage element to be sufficiently protected by checksums.

2.1 Software Approach

In this paper, we present a software approach to checksums on the definition and use of variables. Consider the code listing in Figure 1(a). The first statement adds two constants, 10 and 20, and stores the result in variable temp. The second and third statements use the value stored in temp to perform their computations. Between the definition of temp and its subsequent uses, temp is susceptible to memory errors. We seek to verify that the value stored in temp at the time of its definition is, indeed, what is employed in all subsequent uses. The checksum approach is illustrated in Figure 1(b). The definition of temp contributes to a *definition checksum*. Each use of temp contributes to a *use checksum*. In addition, the number of uses of the variable temp is tracked. At program termination, or at any post-dominator of all definitions and uses tracked, we verify that the definition checksum scaled by the tracked number of uses equals the use checksum.

Such a def-use-based checksum provides comprehensive coverage and can detect faults irrespective of the memory subsystem's architectural characteristics. However, using such a checksum scheme in practice requires overcoming several challenges. Scaling the definition checksums with the use counts for each value potentially requires checksums for each value to be individually stored. This dramatically increases memory space and bandwidth overheads. Maintaining the use count information itself can introduce significant overheads. In addition, the checksum computation introduces an arithmetic operation for every definition and use, i.e., for every load and store instruction.

In the rest of the paper, we address the following challenges:

- How can the number of uses for a given definition be efficiently determined?
- How can the checksums be encoded to minimize the memory costs?
- How can the checksum operators be chosen to minimize the computation costs and maximize fault coverage?
- What is the overhead associated with these schemes?

As system architectures evolve, greater amounts of compute resources are available compared to memory resources. Therefore, we also consider the possibility of hardware support in the processor to assist in the checksum computation.

```
temp = 10 + 20                  temp = 10 + 20
                                contrib_def_chksum(temp)

                                contrib_use_chksum(temp)
                                inc_use_count(&temp)
sum1 = temp + 30                sum1 = temp + 30

                                contrib_use_chksum(temp)
                                inc_use_count(&temp)
sum2 = temp + 40                sum2 = temp + 40

                                /*verify def_checksum scaled by
                                      the use counts matches the
                                      use_checksum*/
```

(a) Original code (b) Checksum augmented code

Figure 1: Illustration of insertion of error detection codes

```
    for j = 0 to n−1
S1: A[j][j]   = sqrt(A[j][j])
      for i = j+1 to n−1
S2:     A[i][j]   = A[i][j]   / A[j][j]
```

Figure 2: Example affine code snippet

2.2 Fault Model

We consider undetected and uncorrected errors in the memory subsystem. This includes (a) undetected multi-bit errors in main memory, caches, write queues, etc., and (b) errors in address generation that result in incorrect data location being operated upon. We focus on incorrect memory operations that go undetected and could potentially lead to silent data corruption. Note that reading data from incorrect locations can cause several bits to differ from the expected value.

The control flow variables, such as loop indices, are assumed to be protected through other means (duplication, invariant assertions, special hardware, etc.). We assume that all processor's subsystems other than memory are resilient to faults. This includes registers, functional units, pipeline latches, and other logic. A consumed value is assumed resilient once it enters the processor, and, conversely, a produced value is assumed correct until it is written out by a store instruction. Thus, we focus on detecting errors in a variable between the time it is written and later read.

3. Compile-time Determination of Use Counts for Affine References

In this section, we describe the algorithm for determining the use counts at compile-time for affine references, beginning with the mathematical notation used. Figure 4 illustrates our approach when the use counts are known at compile-time for the example shown in Figure 1(a). Variable temp defined at the first statement has two uses. Therefore, at the site of its definition, the defined value is multiplied by two and added to def-checksum. At each use-site, the used value of temp is added to the use-checksum. If temp suffers a bit flip in one of its uses, the def- and use-checksums do not match and the error is detected. In this section, we focus on the compile-time determination of use counts for affine references. A detailed discussion of the general checksum scheme and the choice of checksum functions are discussed in Section 5.

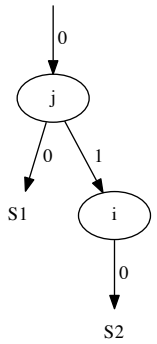

Figure 3: The example code's abstract syntax tree

```
temp = 10 + 20          temp = 10 + 20
                        def_checksum += 2*temp

                        use_checksum += temp
sum1 = temp + 30        sum1 = temp + 30

                        use_checksum += temp
sum2 = temp + 40        sum2 = temp + 40

                        assert(def_checksum
                               ==use_checksum)
```

(a) Original code (b) Checksum-augmented code

Figure 4: Code snippet illustrating the checksum-augmentation scheme when use counts are known at compile-time

3.1 Notation

We employ the same notation as used by Verdoolaege [40] for the definition of sets, relations, apply, and inverse operations.

Sets. A set s is defined as:

$$s = \{[x_1, \ldots, x_m] : c_1 \wedge \cdots \wedge c_n\}$$

where each x_i is a tuple variable and each c_j is a constraint.

The *iteration spaces* of statements (the set of the loop's iteration points) can be represented as sets. For example, the iteration space of statement S1 in the code shown in Figure 2 can be specified as the set I^{S_1}:

$$I^{S_1} = \{S1[j] : (0 \leq j \leq n-1)\}$$

And, the iteration space of statement S2 is represented as the set:

$$I^{S_2} = \{S2[j,i] : (0 \leq j \leq n-1) \wedge (j+1 \leq i \leq n-1)\}.$$

Relations. A relation r is defined as:

$$r = \{[x_1, \ldots, x_m] \mapsto [y_1, \ldots, y_n] : c_1 \wedge \cdots \wedge c_p\}$$

where each x_i is an input tuple variable, each y_j is an output tuple variable, and each c_k is a constraint.

The read and write references of loops are specified as relations from iteration points to array indexes. For example, the write reference A[j][j] in statement S1 of the code shown in Figure 2 is characterized as:

$$r^{S_1}_{write} = \{S1[j] \mapsto A[j,j]\}.$$

Data dependences appearing in the program also are expressed as relations between statements. For example, the flow (Read After Write, or RAW) dependence between write reference A[j][j] of statement S1 and read reference A[j][j] of statement S2 in the sample code is represented by the following relation:

$$d_{flow} = \{S1[j] \mapsto S2[j,i] : (0 \leq j \leq n-1) \wedge (j+1 \leq i \leq n-1)\}$$

The Apply Operation. The apply operation on a relation r and a set s produces a set s' denoted by $s' = r(s)$ and is mathematically defined as:

$$(\vec{x} \in s') \Longleftrightarrow (\exists \vec{y} \text{ s.t. } \vec{y} \in s \wedge (\vec{y} \mapsto \vec{x}) \in r).$$

For a given source iteration of a dependence, its target iterations can be found by *applying* the dependence relation on the given source iteration. For example, consider the source iteration $si = \{S1[10]\}$, i.e., the dynamic instance of S1 when the value of loop iterator j is 10. Its target iterations due to the preceding flow dependence—d_{flow}—can be determined as follows:

$$d_{flow}(si) = \{S2[10,i] : 11 \leq i \leq n-1\}.$$

Schedules. The order of execution of statements in a given program is encoded using 2d+1 schedules [11], where "d" is the maximum number of loops surrounding any statement. A schedule maps iterators of a statement to a combination of iterators and scalar values that specify a global ordering of statements within the program. The abstract syntax tree (AST) of the given program is used to deduce schedules (as follows). Each edge of the AST is assigned a number. Edges originating from a node are numbered from left to right, starting from 0). The schedule of a statement is the vector of numbers and iterators from the AST root to the statement node. If the dimensionality of a schedule is less than 2d+1, it is extended to dimensionality 2d+1 by filling the trailing components of the schedule vector with zeroes.

Figure 3 shows the AST for the example code, and edges of the AST are numbered according to the rule described. The schedules for statements S1 and S2 in the Figure 2 example follow.

$$\left\{ \begin{array}{lcl} S1[j] & \mapsto & [0,j,0,0,0] \\ S2[j,i] & \mapsto & [0,j,1,i,0] \end{array} \right\}$$

Polyhedral Dependences. In the polyhedral model [10] for affine computations, dependence analysis [9] can precisely compute flow (RAW) dependences between dynamic instances of statements. The dependences are expressed as relations from a source iteration to its target iterations involved in the dependence.

Polyhedral dependence analysis (for example, using ISL [19]) generates the following flow dependence for the code in Figure 2:

$$\mathcal{D}_{flow} = \{S1[j] \mapsto S2[j,i] : (0 \leq j \leq n-1) \wedge (j+1 \leq i \leq n-1)\}$$

It characterizes the flow dependence that exists between the write reference A[j][j] of statement S1 and the read reference A[j][j] of statement S2. The value written by statement S1 at a particular iteration of the j loop is used by statement S2 in the same iteration of the j loop and in all iterations of the i loop.

For our analysis, we consider exact dependences and exclude transitive dependences. That is, iterations involved in a dependence are such that if the target iteration of a flow dependence reads a memory cell "X," the corresponding source iteration is the last writer to the memory cell "X" that precedes this reader.

3.2 Compile-time Use-count Determination

Affine loops are those where loop bounds and array references are affine functions of loop iterators and program parameters. Affine computations form an important class of programs: the compute-intensive loops in many scientific codes (computational fluid dynamics, adaptive mesh refinement, numerical analysis, etc.) are affine. According to a study [2], more than 99% of loops in 7 out of

```
    for j = 0 to n−1
      add_to_chksm(use_cs,A[j][j],1)
S1:   A[j][j]    = sqrt(A[j][j])
      if j <= n−2
        add_to_chksm(def_cs,A[j][j],n−1−j)
      for i = j+1 to n−1
        add_to_chksm(use_cs,A[i][j],1)
        add_to_chksm(use_cs,A[j][j],1)
S2:     A[i][j]    = A[i][j]/A[j][j]
```

Figure 5: Example affine code snippet with checksum augmentation

the 12 programs of the SpecFP2000 and Perfect Club benchmarks are affine loops.

Algorithm 1 Compute flow dependence *def_checksum* contribution

Input: Flow Dependences : $\mathcal{D}_{flow}$
Output: Statement and use count pairs: ρ
1: **for all** Statements - S_i **do**
2: $I_i \leftarrow$ Iteration Space ofS_i
3: $I_i^{param} \leftarrow$ Parameterize all iterators of I_i
4: $\mathcal{T}argets_i^{param} \leftarrow \mathcal{D}_{flow}(I_i^{param})$
5: $use\_count^{param} \leftarrow |\mathcal{T}argets_i^{param}|$
6: Add $\{S_i, use\_count^{param}\}$ to ρ
7: **end for**
8: **return** ρ

Algorithm 1 describes how we determine the number of uses, referred to as the use count, for each definition. For each statement that produces a value, the definition statement, we determine the number of its consumers. The iterators of the surrounding loops of a statement under consideration are equated to some parameter values (line 3), I_i^{param}, allowing us to refer to a single parameterized iteration of that statement. In the flow dependences that map source iterations to their target iterations, I_i^{param} is substituted as the source iteration, and its target iterations are computed (line 4). The cardinality of the target iteration set provides the definition statement's use count (line 5). The algorithm returns a collection of these statement and use count pairs.

Example. The use count for statement S1 in the example code shown in Figure 2 is computed as follows:

$I_1^{param} = \{S1[j] : j = jp\}$ where jp is a parameter

$\mathcal{T}argets_1^{param} = \mathcal{D}_{flow}(I_1^{param}) = \{S2[jp,i] : (0 \le jp \le n-1)$

$\wedge (jp+1 \le i \le n-1)\}$

$|\mathcal{T}argets_1^{param}| = \{n-1-jp : (0 \le jp \le n-2)\}$

The $\mathcal{D}_{flow}$ is the flow dependence shown in Section 3.1 for the example code. The value written by the write reference A[j][j] of statement S1 is used by the read reference A[j][j] of statement S2 in n-1-j following iterations (the lower bound of iloop is j+1, and the upper bound is n-1, resulting in n-1-j iterations). Thus, the use count of S1 is n-1-j as determined by the algorithm. Furthermore, it can be noted that the preceding use count output applies to iterations of the j loop up to n-2, and the last iteration, when j=n-1, is excluded. This is because the last iteration has no target iterations in statement S2. When j=n-1, the lower bound of the i loop (n) becomes greater than its upper bound (n-1), resulting in no instance of statement S2 being executed when j=n-1.

Figure 5 shows the checksum-inserted version of the example code. The add_to_chksm macro definition takes three parameters:

```
    for j = 0 to n−2 /*index split j*/
      add_to_chksm(use_cs,A[j][j],1)
S1:   A[j][j]    = sqrt(A[j][j]);
      add_to_chksm(def_cs,A[j][j],n−1−j)
      for i = j+1 to n−1
        add_to_chksm(use_cs,A[i][j],1)
        add_to_chksm(use_cs,A[j][j],1)
S2:     A[i][j]    = A[i][j]/A[j][j]
    j = n−1 /*index split j*/
    add_to_chksm (use_cs,A[j][j],1)
    A[j][j]    = sqrt(A[j][j])
```

Figure 6: Checksum-augmented example affine code with index-set splitting optimization

the checksum to add to (def/use checksum), the value to add, and the number of times the value is to be added. The produced value is multiplied by the use count and added to the def-checksum. Consumed values are added to the use-checksum.

3.3 Optimization: Index-Set Splitting

In the checksum-augmented code shown in Figure 5, following statement S1, the error detection code that adds the value A[j][j] written by statement S1 to the def-checksum has a branching structure: A[j][j] is added to def-checksum n-1-j times only for iterations of j loop up to n-2.

To minimize performance penalty due to such control overheads, iteration space of the j loop may be split so that in each of the split iteration spaces, A[j][j] has the same use count. Thus, the need to evaluate if conditionals in each iteration of the j loop is obviated. The loop-partitioned code thus formed is shown in Figure 6. The last iteration of the j loop, when j = n-1, is peeled from the rest of the iterations. As already explained, statement S2 does not appear in the peeled iteration at j=n-1 due to the loop bounds.

Algorithm 2 Split iteration spaces

Input: Iteration Spaces : $I_{S_j}^{in}$, Index Sets : δ_{S_i}, Schedules : θ
Output: Loop Nest : $\mathcal{L}$
1: **for all** Iteration Spaces : $I_{S_j}^{in}$ **do**
2: **for all** Index Sets - δ_{S_i} **do**
3: **if** Under θ, ($Iterators_{common}$ = Iterators_of $\delta_{S_i} \cap$ Iterators_of I_{S_j}) $\neq \phi$ **then**
4: $I_{S_j}^{out} \leftarrow$ Split indexes of $I_{S_j}^{in}$ s.t ($\mathcal{R}ange$ of $Iterators_{common} \in I_{S_j}^{in}$) $\subseteq$ ($\mathcal{R}ange$ of $Iterators_{common} \in \delta_{S_i}$)
5: **end if**
6: **end for**
7: **end for**
8: $I_{S_j}^{out} \leftarrow I_{S_j}^{out} \cup (I_{S_j}^{in} \setminus I_{S_j}^{out})$
9: $\mathcal{L} \leftarrow$ Generate Code to traverse $I_{S_j}^{out}$ with schedule θ
10: **return** $\mathcal{L}$

Algorithm 2 describes the general procedure for splitting iteration spaces so that the use count of a statement in a split iteration space remains the same for all iterations in that space. Inputs to the algorithm are the iteration spaces of all statements $I_{S_j}^{in}$, the index sets δ_{S_i} (such as $0 \le j \le n-2$, $j = n-1$ in the preceding example), and the schedule that defines the order of execution of the iteration spaces. The index sets δ_{S_i} act as the criteria, defining which iteration spaces are to be split.

```
write temp

if(x[10]) {

  read temp
}

if(z[5]) {

  read temp
}
```

(a) Original code

```
write temp
def_checksum += temp
e_def_checksum += temp

if(x[10]) {
  temp_use_count++
  use_checksum += temp
  read temp;
}

if(z[5]) {
  temp_use_count++
  use_checksum += temp
  read temp
}

/*Epilogue*/
def_checksum +=
temp*(temp_use_count-1);
e_use_checksum += temp;

assert(def_checksum
       ==use_checksum)
assert(e_def_checksum
       ==e_use_checksum)
```

(b) Checksum-augmented code

Figure 7: Code snippet illustrating the general checksum-augmentation scheme

For each iteration space, it is checked if a particular index set can cause the iteration space to be severed. If there are any common iterators between the index set and the iteration space, then the index set potentially splits the iteration space (line 3). In the example code, any split of the j loop affects iteration spaces of both statements S1 and S2. However, if only the i loop is broken up, it does not affect the iteration space of statement S1 as loop i is not a surrounding loop for statement S1. For the common iterators identified, the iteration space is split so that the range of values an iterator assumes is a sub-range of values the same iterator assumes in the index set (line 4). This results in partitioning of the iteration space of that statement and ensures that no partition is a strict superset of the index set. The resulting smaller iteration spaces constitute $I_{S_j}^{out}$.

Any iteration points not yet a part of $I_{S_j}^{out}$ are subsequently added to $I_{S_j}^{out}$ so that all of the iteration points contained in $I_{S_j}^{in}$ are included in $I_{S_j}^{out}$ (line 8). Finally, the code for the loop nest is generated by traversing $I_{S_j}^{out}$ according to the schedule θ (line 9). The output of the algorithm is the index-set split loop nest.

4. Inspectors for Dynamic Start-time Use Count Determination

Hitherto, regular loops whose iteration spaces and array accesses can be characterized and properties about them discerned at compile-time were examined. In contrast, irregular codes require a combination of static and dynamic approaches to establish properties about them.

4.1 Basic Approach

Consider the code in Figure 7(a) that is shown to illustrate the challenges and the solution approach for irregular codes. The variable temp is defined, then its uses are subject to x[10] and z[5] being non-zero. Therefore, depending on their values, temp may be used once, twice, or not at all. We proceed as follows: at the def-site, temp is added to the def-checksum. At the use-site, temp is added to use-checksum, and a counter (temp_use_count) is incremented to track the total number of times the variable is used. Finally to match the checksums, the value of temp is added to def-checksum (temp_use_count − 1) times (it was added once to the def-checksum at the def-site). If there were no memory errors, then the two checksums match.

```
   while not converged
     for j1 = 0 to n−1
S1:    temp1 += p_new[cols[j1]]
     for j2 = 0 to n−1
S2:    temp2 += p_new[j2]
     for j3 = 0 to n−1
S3:    p_new[j3] = temp3
     /*convergence check not shown*/
```

Figure 8: Example code requiring dynamic use count determination

It turns out, using this approach is insufficient to catch all memory errors. Consider the scenario where both the conditionals in the code shown evaluate to true, resulting in two uses of temp. Let us further assume that the first read was correct. At this point, both def-checksum and use-checksum equal temp. Before the second read, let us suppose that a memory error occurred, and temp got changed to temp'. At second read, it gets added to use-checksum, and use-checksum is now equal to temp+temp'. At epilogue, temp' gets added to def-checksum (as temp_use_count would be 2), which makes def-checksum equal temp + temp', the same as the present value of use-checksum, and the memory corruption goes undetected.

The problem with merely adding the current value of a variable to the def-checksum in the epilogue is that the corrupted value may be added to both the def- and use-checksums and escape detection. We fix the problem by defining auxiliary checksums: e_def_checksum and e_use_checksum. e_def_checksum is computed at the def-site as before, but e_use_checksum is computed only after the last use of the variable rather than at each use-site. Figure 7(b) illustrates the error detection code generated using the scheme described. We note that the problem explained now gets resolved: e_def_checksum = temp, but e_use_checksum = temp', and the memory error is detected. The modified scheme with the auxiliary checksums ensures that the value of the variable finally added to the def-checksum to match the number of its uses is not potentially corrupted.

4.2 Optimizations for Iterative Codes

Figure 8 shows an example of a code requiring dynamic support. Accesses to array p_new at statement S1 are data-dependent. Moreover, the number of iterations of the loop is dynamically determined. We describe how to generate error-detection checksums for the example, focusing on two arrays p_new and cols followed by a description of the general scheme.

The following set of observations is made about the reads (uses) and writes (definitions) to array p_new:

- The reads to p_new at statement S1 are indexed by cols array entries. However, no element of array cols is written to in the while loop. Therefore, the same set of array elements of p_new is read in every iteration of the while loop.

```
   /*Inspector*/
   for j1=0 to n−1
     count_p_new[cols[j1]]++
   iter = 0
   while not converged
     iter++
     for j1=0 to n−1
       add_to_chksm(use_cs,cols[j1],1)
       add_to_chksm(use_cs,p_new[cols[j1]],1)
S1:    temp1 += p_new[cols[j1]]
     for j2=0 to n−1
       add_to_chksm(use_cs,p_new[j2],1)
S2:    temp2 += p_new[j2]
     for j3=0 to n−1
S3:    p_new[j3] = temp3
       add_to_chksm(def_cs,p_new[j3],count_p_new[j3]+1)
     /*convergence check not shown*/

   /*Epilogue*/
   for i=0 to n−1
   add_to_chksm(def_cs,cols[i],iter−1)
   add_to_chksm(e_use_cs,cols[i],1)
   add_to_chksm(use_cs, p_new[i],count_p_new[i]+1)
```

Figure 9: Example code with inspector to determine use counts and the generalized checksum-augmentation scheme

- The elements of p_new read at statement S2 are amenable to compile-time analysis as the array index expressions and loop bounds of enclosed loop indexed by j2 are affine.
- The new definition of array p_new at statement S3 is used a *fixed number of times* (although not known at compile-time) before being overwritten in the next iteration of the while loop if the algorithm has not converged.

An *inspector* is used to examine the cells of array p_new that will be read because of data-dependent accesses at statement S1 . Because these reads are loop invariant, the inspector is hoisted above the while loop to reduce the overhead of running the inspector. The parts of the code that are affine are subjected to static analysis techniques developed in the previous section, and the information from the inspector and static analysis are combined to generate the checksum calculation code.

The reads to array cols are affine. However, the number of iterations of the while loop is not known *a priori*. The number of accesses to a cell of array cols will be the number of accesses to it in an iteration of the loop, which can be determined by compile-time techniques, multiplied by the number of iterations. To determine the number of iterations of the loop, we introduce a new variable to count the number of dynamic executions of the loop. For such read references, the number of accesses is a function of the dynamic loop count. We observe that this count is not known at the time of the value's definition. In such cases, we adopt the following method. At the definition site, the defined value is added to def-checksum and the auxiliary def-checksum e_def_checksum once. At each use site, the used value is added to use-checksum once. Post loop execution, in the epilogue code, the value is added to def-checksum one less than loop-count times. It also is added to the auxiliary use-checksum e_use_checksum to balance contributions to the def- and use-checksums.

Figure 9 shows the checksum-augmented version of the example in Figure 8. The inspector code counts the number of accesses to data-dependent references to array p_new. The loop is instrumented with def- and use-checksum computation operations. Variable iter keeps count of the number of iterations of the while loop. In the epilogue code, the values of array cols are added to def-checksum iter−1 times. The values of p_new are added to use-checksum in the epilogue and account for the fact that the new definition of p_new in the last iteration of the while loop goes unused.

5. Overall Approach To Transient Error Detection

We employ one global checksum to track all definitions of values and another global checksum to track all uses. A mismatch between the two checksums indicates the occurrence of one or more data corruption errors. In general, array accesses are classified into two categories: affine and non-affine. Affine references are analyzed using compile-time techniques developed in Section 3. For non-affine accesses, inspector code is generated to examine the number of uses of any array cell. The inspector is hoisted out of the looping structure when legally feasible—when the indexing data structures are not modified in the loop. The number of uses of a definition from affine and non-affine uses are combined to arrive at the use count for a definition.

Algorithm 3 Insert error detection checksums

Input: The abstract syntax tree of a program: *AST*
Output: The AST of equivalent resilient program: AST'

1: $Live-in \leftarrow$ Gather live-in values and associated *use_count*s in the *AST*
2: $Prologue \leftarrow$ Generate operations adding $Live-in$ to $def\_checksum$ and to auxiliary $e\_def\_checksum$
3: **for all** $Use\_Site \in AST$ **do**
4: Insert code to add read operands to *use_checksum*
5: **if** the read is in an irregular access **then**
6: Insert code to increment the value of *use_count*
7: **end if**
8: **end for**
9: **for all** $Def\_Site \in AST$ **do**
10: **if** *use_count* is known **then**
11: Insert code to add the defined value to $def\_checksum$, *use_count* times
12: **else**
13: Insert code to add the previous value to $def\_checksum$, $use\_count-1$ times
14: Insert code to add the previous value to $e\_use\_checksum$ once
15: Insert code to add the new value to $def\_checksum$ and $e\_def\_checksum$
16: Insert code to set *use_count* to 0
17: **end if**
18: **end for**
19: $Adjustment\_Pending\_Defs \leftarrow$ Gather variables in the *AST* whose *use_count* was not known
20: **for all** $Var \in Adjustment\_Pending\_Defs$ **do**
21: Insert code in $Epilogue$ to add Var to $def\_checksum$, $use\_count-1$ times
22: Insert code in $Epilogue$ to add Var to $e\_use\_checksum$ once
23: **end for**
24: $Verifier \leftarrow$ Add code to assert equality of def and use checksums
25: $AST' \leftarrow$ Append $\{Prologue, AST, Epilogue, Verifier\}$

Algorithm 3 presents the general approach to generating the error detection checksums. At each use-site, the read values are added to the use-checksum, and the use counts are incremented if the reads are in irregular accesses. At a def-site, if the number

of the uses can be predetermined (such as described in Section 3 for affine array references), the produced value is added to def-checksum as many times as it will be used. If the number of uses for a definition of a value cannot be determined *a priori*, the newly defined value is added once to def-checksum and once to an auxiliary def-checksum, to be adjusted in the epilogue code. A use count also is maintained and initialized to zero. However, before the new value is written, checksum adjustments are made for the previous value of that variable about to be overwritten. The old value is added to def-checksum, use_count−1 times, and also is added to update the auxiliary use-checksum. At the end of the program, the verifier code that compares the def- and use-checksums is introduced to detect any memory errors that might have occurred to any variables during execution of the program.

Any commutative and associative checksum operator can be chosen for error detection. We use *integer modulo addition* as the checksum function in this work. If an operand is a floating-point number or a character, its bit representation is viewed as an integer of the appropriate size before invoking the checksum operation. Another candidate checksum operator that has associative and commutative properties is exclusive or (XOR). We use integer arithmetic because it is supported as efficiently in hardware as XOR while exhibiting superior fault coverage [23].

The fault model considered in this work (Section 2.2) assumes that the ALU and registers are resilient to errors, and the memory subsystem that includes caches, memory, and write queues is vulnerable to errors. Therefore, the checksum approach requires that the produced values be resident in registers until they are added to def-checksum and conversely, the consumed values that are added to use-checksum reside in registers until their actual use. Load and store operations due to register spills require additional checksum contributions to ensure their correctness. The checksums must be held in registers so they are not subject to memory errors. If checksums are affected by bit-flips, it would cause false-positives (i.e., an error may be flagged even when the data are not corrupted). Overflows in checksum values do not affect the presented scheme as integer modulo addition is associative and commutative. An inherent property of checksums is they can trigger false negatives: errors canceling each other out and resulting in a correct checksum value. We empirically evaluate the probabilities of fault negatives of the checksum scheme in Section 6.1.

We now discuss the correctness of the checksum scheme.

Theorem 5.1. *The checksum computation codes inserted by Algorithm 3 correctly detect all memory errors with high probability provided checksums are register-resident.*

Proof. We want to prove that checksums flag an error—def- and use-checksums do not match—if for any variable, the value assigned to the variable at its definition-site is different from the values carried by the variable at any of its subsequent use-sites. And, we want to show that various checksum adjustments are correct and do not trigger false positives,i.e., checksums match if there are no memory errors.

We first consider the case where there is a single variable in the program then extend the analysis to show that the scheme works for multiple variables. Let the value assigned to the variable at def-site be v. Let it be used n times. Now, we can have two cases: 1) use count, the number of uses of the variable, is known at compile-time or 2) uses occur in data-dependent paths of the AST and use count is not known at compile-time.

Case 1 (known use count). When use count, say n, is known at compile-time, value v is multiplied by n and is added to def-checksum. Therefore, def-checksum is $n \times v$. We analyze what happens to checksums for different values of n and in the presence of errors.

(a) $n = 0$: Both def-checksum and use-checksum are 0. Because the number of uses is zero, data errors do not happen.

(b) $n = 1$: At the use-site, if the value read is v, the two checksums match. Otherwise the error is caught.

(c) $n > 1$: Let the values read during the n reads be $v_1, v_2, \ldots, v_n$. The checksum values will be $def\_checksum = n \times v$ and $use\_checksum = v_1 + v_2 + \cdots + v_n$. If there were no memory errors, each v_i will equal v, and the two checksums will match. If any of the reads are wrong, then with a high probability def-checksum and use-checksum will not match and the error will be detected. A false negative can happen when $n \times v = v_1 + v_2 + \cdots + v_n$, although some v_is are different from v.

Case 2 (unknown use count). At the def-site, value v is used to initialize def-checksums: $def\_checksum = v$ and $e\_def\_checksum = v$.

(a) $n = 0$: Because there are no uses, use-checksum and use count are 0. In the epilogue code or just before the variable is overwritten, value v is added to the def-checksum $use\_count - 1$ $(= -1)$ times. Therefore, def-checksum becomes 0. v is added to auxiliary checksum: $e\_use\_checksum = v$. Thus, $def\_checksum = use\_checksum = 0$, and $e\_def\_checksum = e\_use\_checksum = v$, unless there are memory errors.

(b) $n = 1$: When there is only one use, def-checksum is not adjusted because $use\_count - 1 = 0$ and memory errors are detected with certainty.

(c) $n > 1$: Let the values read during the n reads be: $v_1, v_2, \ldots, v_n$. Therefore, $use\_checksum = v_1 + v_2 + \cdots + v_n$. When adjusting def-checksum, let the value of the variable be v_{n+1}. Hence, finally $def\_checksum = v + (n-1) \times v_{n+1}$, and $e\_use\_checksum = v_{n+1}$. If there are no errors, the checksums match: $def\_checksum = use\_checksum = n \times v$ and $e\_def\_checksum = e\_use\_checksum = v$. When there is an error, it is caught if either $v + (n-1) \times v_{n+1} \neq v_1 + v_2 + \cdots + v_n$ or $v \neq v_{n+1}$.

Using *e_checksums* prevents an important class of errors from going undetected. If after first use, v changes to v' and the error persists, then $def\_checksum = use\_checksum = v + (n-1) \times v'$. However, auxiliary checksums correctly flag the error: $e\_def\_checksum = v$ and $e\_use\_checksum = v'$, and they do not match.

Given that the checksum operator is commutative and associative, an analysis similar to the one described can be employed for multiple variables to establish that the checksum values match if there are no errors and they do not match with a high probability in the presence of memory errors.

□

6. Experimental Evaluation

We first evaluate the fault coverage achieved by the checksum scheme. Then, performance overheads of checksum computations over a number of benchmarks are characterized.

6.1 Effectiveness of Checksums

A characteristic of using checksums for error detection is that errors in multiple values contributing to a checksum could cancel out one another, producing a seemingly correct checksum even though there were bit flips in the data. We therefore empirically assess the percentage of errors that can escape detection. One-bit errors are always caught. Only multi-bit errors potentially can go undetected. Therefore, we inject multi-bit errors into an array of 64-bit integers,

Table 1: Percentage of undetected errors with integer modulo addition checksums

		One checksum			Two checksums		
#bit-flips	N	All 0 bits	All 1 bits	Random bits	All 0 bits	All 1 bits	Random bits
2	10^2	0.025%	0.025%	0.790%	0.011%	0.011%	0.024%
	10^4	0.014%	0.014%	0.755%	0%	0%	0.017%
	10^6	0.014%	0.014%	0.763%	0%	0%	0.022%
3	10^2	0.002%	0.002%	0.020%	0%	0%	0%
	10^4	0.002%	0.002%	0.030%	0%	0%	0%
	10^6	0.002%	0.002%	0.020%	0%	0%	0%
4	10^2	0%	0%	0.015%	0%	0%	0%
	10^4	0%	0%	0.020%	0%	0%	0%
	10^6	0%	0%	0.014%	0%	0%	0%
5	10^2	0%	0%	0.001%	0%	0%	0%
	10^4	0%	0%	0.002%	0%	0%	0%
	10^6	0%	0%	0.003%	0%	0%	0%
6	10^2	0%	0%	0%	0%	0%	0%
	10^4	0%	0%	0%	0%	0%	0%
	10^6	0%	0%	0%	0%	0%	0%

We therefore inject multi-bit errors into an array of 64-bit integers and monitor the percentage of cases where checksums are successful in detecting them. Over $100,000$ trials, the following steps are repeated: the data are initialized and the initial 64-bit checksum is computed. Then, either two, three, four, five, or six bits over all bits of the array are uniformly and randomly selected, and values of those flipped bits. Again, a 64-bit checksum is computed and compared with the initial checksum. If there is a mismatch in the checksum values, the injected error has been correctly caught. Otherwise, error has escaped detection. The percentage of instances when errors are not caught over $100,000$ trials is reported. For small array sizes, we gathered data for even higher number of trials, up to 1 million trials. The percentage of undetected errors we observed was similar to what we report here for $100,000$ trials.

Table 1 shows the percentage of cases in which a multi-bit error (2, 3, 4, 5, 6 bit-flips) is not caught for arrays with $10^2, 10^4$, and 10^6 64-bit integers under the column header "One checksum." Experiments are carried out over three kinds of data. All bits of the array elements are 0s, all bits are 1s, and the bits are randomly initialized.

Two-bit errors experience the greatest percentage of undetected errors. The number of undetected errors approaches zero as the number of flipped bits increases. The reason for this is that when multiple bits (greater than 2) are flipped, other bit-flips may not exactly line up, even though a pair of errors in different values line up to cancel each other out. Thus, the error still is caught with a high probability.

Among different types of data values, the percentage of undetected errors is highest for random values. When bits of all array elements are initialized to either 0s or 1s and if bits at the same bit-position in two array elements flip, the carry bit also is changed, and the error will be caught unless the bit in the next significant bit-position also is flipped. In contrast, random data are likely to have an equal number of 0s and 1s. When a 0 becomes 1 and a 1 becomes 0 at the same bit-position in two values, it will not change the carry bit. Thus, the error can go undetected with a higher probability. Furthermore, we observe that checksum performance is largely independent of the number of values that go into forming the checksum. In more than 99% of cases, errors are successfully detected.

The odds of error detection can be further improved and protection against data corruption fortified by using multiple checksums. As before, the first checksum is a direct sum of the values. The second checksum is obtained by adding permutations of the values encountered.

We evaluate the fault coverage when two checksums are used. Two 64-bit checksums are computed from initial data, errors are injected, and checksums are recomputed. If there is a mismatch between either pair of checksum values, the error has been detected. While forming the second checksum, each array element is left-rotated by an amount determined by five bits of the array element's address, i.e., the fourth to eighth least significant bits. The address of a 64-bit integer, which occupies 8 bytes, likely will be a multiple of 8. Hence, the first three least-significant bits—the first to third—are likely to be zero. Thus, each value that goes into computing the second checksum is left-rotated by an amount between 0 and 31 ($2^5 - 1$) bits, depending on its address.

Therefore, with a high probability, the corrupted values are rotated by different amounts, which prevents the lined-up erroneous bits in one checksum from lining up to cancel out one another after rotation in the other checksum. The percentages of undetected errors using the two-checksum scheme are shown in Table 1 under the column header "Two checksums." Only a small percentage of two-bit errors are undetected, and all three-, four-, five-, six-bit errors are detected.

6.2 Performance Overheads

Table 2 lists the benchmark programs and problem sizes used to measure performance overheads of the checksum-augmented codes. All array references in ADI, cholesky, dsyrk, jacobi1d, LU, seidel, strmm, and trisolv are statically analyzable, whereas a subset of array references in CG and moldyn are irregular. For those references, dynamic analysis techniques are applied. The affine benchmark codes are taken from the PLUTO benchmark suite [32].

The performance overheads are evaluated on an Intel Xeon E5630 processor running at 2.53 GHz with 32 KB L1 cache. The

Table 2: Benchmarks

Benchmark	Description	Problem size
ADI	Alternating direction implicit solver	TSteps = 500, N = 3000
CG	Conjugate gradient	TSteps = 1500, NZ = 513072
cholesky	Cholesky decomposition	N = 3000
dsyrk	Symmetric rank-k update	N = 3000
jacobi1d	1-D Jacobi stencil computation	TSteps = 100000, N = 400000
LU	LU decomposition	N = 3000
moldyn	Molecular dynamics	TSteps = 100000, N = 400000
seidel	2-D seidel stencil	TSteps = 500, N = 3000
strsm	Triangular matrix equations solver	N = 3000
trisolv	Triangular system of linear equations solver	N = 3000

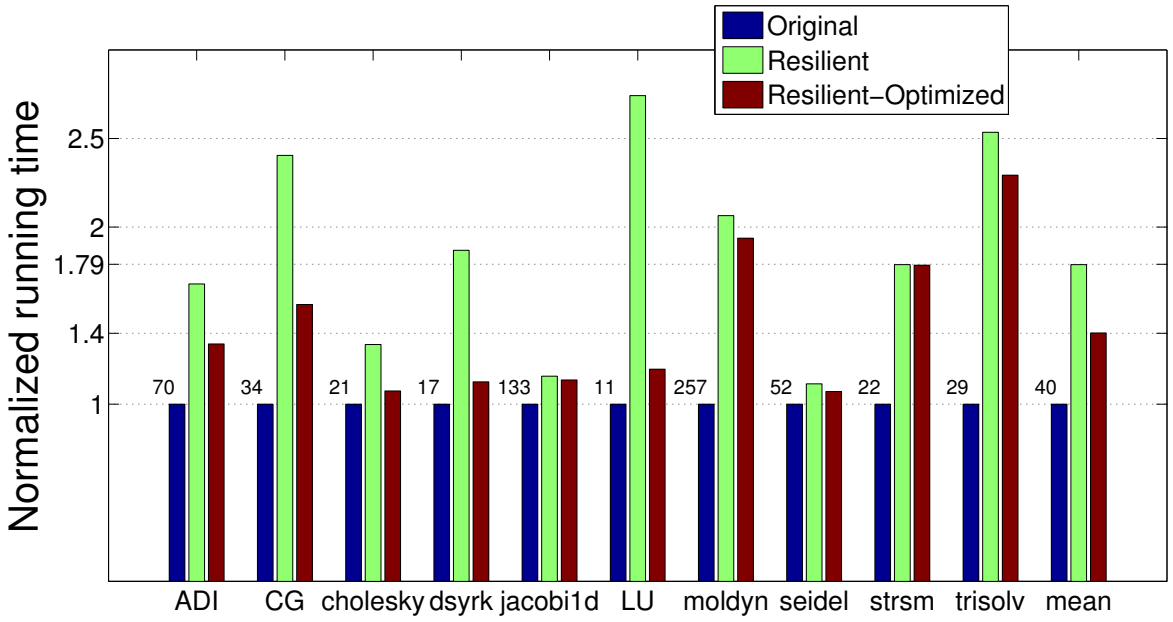

Figure 10: Normalized running time of the resilient codes with one checksum software-only solution. Absolute running times, in seconds, of the Original unmodified codes are also shown

programs are compiled using Intel icc 13.1.3 compiler with -O3 optimization flag. Each benchmark is run five times and the average execution time is reported.

6.2.1 One Checksum, Software-only Solution

Figure 10 shows the performance of the checksum-augmented programs, referred to as resilient versions, as compared to the performance of the original non-resilient versions. Across all benchmarks, the geometric mean of performance overheads of resilient codes is 78.8% (the Resilient bars in the figure). When the index-set splitting optimization presented in Section 3.3 and inspector-hoisting optimization described in Section 4.2 are applied, performance overheads are reduced to 40.2% on average (geometric mean) (the Resilient-Optimized bars).

The index-set splitting transformation partitions a loop. Hence, statements in a partitioned loop have the same use count, removing any conditional statements introduced in the loop due to use count of a definition changing based on the loop iterator values. Thus, index-set splitting reduces control overheads and improves performance. Further, it enables vectorization if the compiler was unable to vectorize a loop because of conditional statements in the loop. For example, the original LU code is vectorized, with a running time of 11.1 seconds, while its resilient version is not vectorized, incurring a running time is 30.3 seconds. When index-set splitting transformation is applied to the resilient code, the compiler successfully vectorizes the resulting loops, and the resulting running time is 13.2 seconds.

The highest overhead is observed for the moldyn benchmark. While moldyn is an iterative code, the inspector for one of the arrays used cannot be hoisted out as loop-invariant properties are not preserved. Therefore, counters are used to track the number of uses of array elements. The CG benchmark is an iterative method that includes sparse matrix-vector multiplications. Although it has irregular accesses, each iteration of the computation has the same data access pattern. Hence, the inspector code for running through irregular accesses is hoisted out of the iterative loop. The optimized

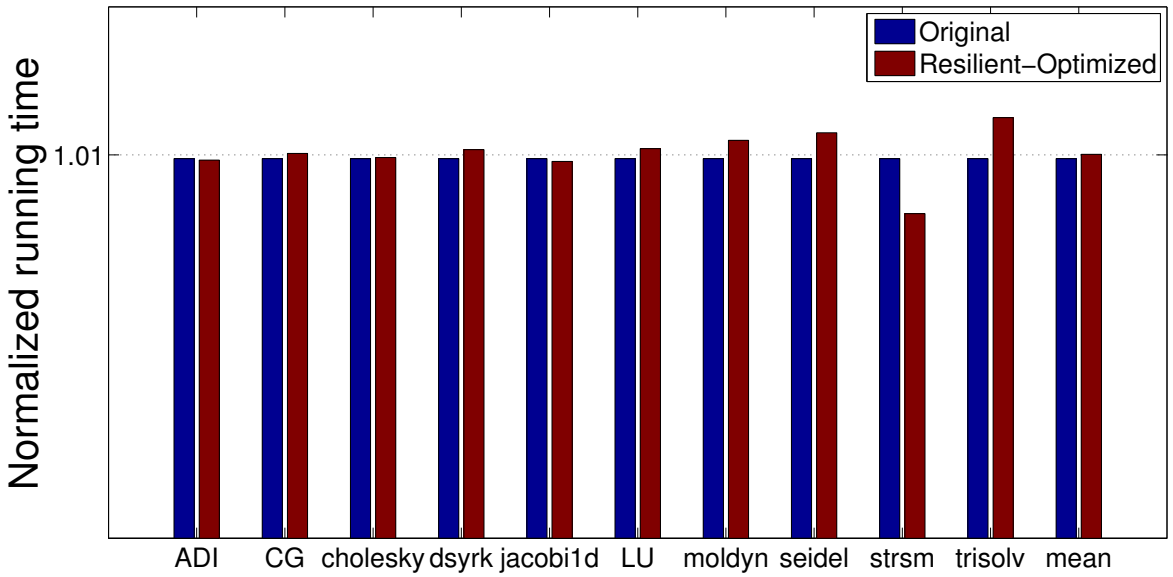

Figure 11: Estimated performance of resilient codes with a special function unit to compute checksums. Y-axis employs a linear scale with selective tick marks to highlight the costs.

running time shown in Figure 10 for CG includes the effect of both index-set splitting and inspector-hoisting optimizations. However, all performance benefits for the CG benchmark accrue from the inspector-hoisting optimization. The running time increases from 33.7s for the original version to 81.1s for the resilient version, but reduces to 52.7s for the resilient version with the inspector hoisted out of the iterative loop. Index-set splitting does not affect this performance.

6.2.2 Evaluating Overheads under Hardware Support

Software-based checksum schemes computing one checksum incur a non-trivial overhead. Tracking multiple checksums in software would be too expensive to be used in practice. Here, we evaluate the potential improvements in the costs when hardware assistance is available. Rather than realize the optimal hardware design to support def-use checksums, we present a candidate design and estimate the likely performance overheads.

We consider the presence of special checksum functional units to compute the checksums. We assume sufficient parallelism in these units so that they never lead to stalls in the execution pipeline. For example, one checksum unit could be associated with every functional unit to compute the checksums on the inputs and outputs of that functional unit. Given that a checksum unit only performs integer modulo arithmetic, the hardware space requirements would be similar to that of an integer ALU, and much lesser than that of a floating point unit. Note that a more detailed analysis of the execution pipeline might achieve the same effect with less resources dedicated to checksum computation.

The intermediate values of the checksums are stored in internal registers and are reduced to a scalar when requested. We assume the presence of instructions to set and query the checksums. In addition, we assume instructions to specify the checksum contribution behavior of another instruction. For example, a checksum instruction could specify which of the operands of a following floating-point instruction contribute to the def- and use-checksums. In addition, def-checksums can be associated with a scale factor based on the use count. Use counts that are small constants are encoded directly in the checksum instruction. For large or non-constant use counts, we assume the presence of a use-count register that is set by a separate instruction. A checksum instruction can scale the value defined by a subsequent arithmetic or memory instruction by a small constant or by the value in the use-count register.

We estimate the cost of supporting checksums by transforming the index-split resilient version of each benchmark to exploit such a feature set. Because the checksums can now be computed in hardware, we remove the operations that compute them in software. In place of the actual checksum calculation operations, we insert instructions corresponding to the checksum instructions. For all benchmarks other than CG and moldyn, one checksum instruction is inserted to precede every floating point arithmetic operation. For CG and moldyn, which also checksum the integer indirection arrays, we insert a checksum instruction for every memory operation. Note that these are more instructions than required. However they provide adequate coverage for all checksum points of interest.

We considered candidate instructions to estimate the overhead and report results for a nop instruction (no operation—the instruction is fetched and decoded but does not use any functional unit resources) is used to represent each checksum instruction. The nop instruction is of sufficient width to pack the actions specified by the checksum instruction. Because insertion of assembly instructions might interfere with compiler optimizations, the nop instructions are inserted into optimized assembly generated by the compiler (`icc -O3` in this case). The operations to compute the use counts and all operations in the prologue and epilogue portions of the resilient code are still retained.

Note that this design can support multiple checksums without introducing any additional software overhead. We evaluate the overhead of this design by benchmarking the resulting nop-inserted code. The resulting overheads are shown in Figure 11. We observe that the overheads are significantly reduced, with the largest overheads (4%–10%) incurred by moldyn, seidel, and trisolv. We observed that the input original version of trisolv was performing

worse than the index-split version and made the index-split version, without any checksums, the baseline for comparison in both Figure 10 and Figure 11. We observe a speedup in strsm due to the compiler vectorization having different effects on the two versions of the code. Excluding strsm, we observe an average overhead of 3% (geometric mean), demonstrating that hardware support could help significantly reduce the overheads in computing the checksums while also supporting multiple checksums.

7. Related Work

Fault Tolerance. Approaches to tackling soft errors have been considered at various levels of the hardware-software environment and typically involve redundancy coupled with periodic validation. At the lowest level, hardware checkers, such as self-checking logic [26] and hardware duplication [1], provide the most general coverage, but they can be expensive in terms of chip area, performance, and power and not widely available on all systems of interest.

More general schemes on commodity hardware resort to different forms of execution redundancy and periodic validation, possibly aided with microarchitecture support. This includes approaches for simultaneous multithreading [14] and chip multiprocessors [36] that employ redundantly executing threads whose results are periodically compared. Process-level redundancy [39] involves duplicating the inputs and entire execution on distinct processes whose outputs are then compared. These schemes incur significant overheads or require specialized hardware to frequently validate the ongoing computation.

Software-level redundancy techniques that are agnostic of application structure duplicate instructions within a single thread of execution and introduce additional checking instructions for validation. Such an approach could check the computation [28] or control flow [27] while duplicating all of the application state. SWIFT [35] checks the computation and control state without duplicating application state and assumes that memory is made fault tolerant through other means, such as ECC. The approach presented in this paper can complement SWIFT by decoupling correctness checks for the memory subsystem from those for control and computation.

An alternative to redundancy-based techniques, symptom-based detection techniques employ low-cost detectors that observe violation of application visible properties, such as loop trip counts and invariants [16], as the outcome of underlying hardware faults. These solutions can be software-only [13, 41] or combine hardware support [31] in the design of low-cost symptom-based detectors. These incur lower overheads as compared to redundancy-based techniques while potentially trading off fault coverage. Hari et al. [17] observed the trade-off between fault detection latency—the delay in detecting a fault—and the detection overhead in symptom-based detectors. These techniques focus on detection of errors in computation or control flow instructions. Our work focuses on the design of symptom-based detectors for multi-bit memory errors.

Schroeder et al. [37] observed that the likelihood of repeated failures in DRAM increases after a first failure has occurred. Multi-bit memory errors have been observed in SRAM soft error evaluation experiments [22, 29]. Yoon et al. [42, 43] designed approaches to virtualize ECC for main memory so as to increase its flexibility and offload expensive ECC error correction for last-level caches to DRAM. Gold et al. [12] observed that multi-bit error detection and correction in L1 caches are more expensive in terms of performance, power, and area even for reasonable sizes.

Shirvani et al. [38] designed approaches to provide checksum protection by periodically scrubbing memory, rather than check every read and write operation, which lowers fault coverage compared to our approach. Checksum approaches for various data structures, such as trees, have been considered by exploiting structure-specific properties [6]. Algorithm-based fault tolerance for linear algebra relies on distributivity of floating point multiplication over addition to make specific array operations resilient [18]. Chen at al. [7] optimized the management of checksums for these algorithm-based fault tolerance schemes. Blum et al. [4] presented checkers that can validate operations on data structures stored in unreliable memory using the minimal amount of reliable memory required. Our approach does not rely on any assumptions about floating point arithmetic and is not restricted to specific algorithms or data structures.

The Flikker system [21] distinguishes critical and non-critical data in programs and stores non-critical data in DRAM memories that are refreshed at lower rates, saving energy but introducing data corruption. The compiler-assisted approach in this paper can complement Flikker and similar systems in detecting errors in data stored in unreliable memory, further enabling the use of such energy-saving strategies.

Maxino [23] evaluated error detection effectiveness of different checksum algorithms, namely, exclusive or, two's complement addition, one's complement addition, Fletcher checksum, Adler checksum, and cyclic redundancy codes.

Compile-time Analysis and Transformation. Griebl et al. [15] used the index-set splitting approach to form regular dependence structures so effective loop transformations can be applied. As there can be a large number of ways to split loops, they addressed the problem of efficiently finding index sets that yield good loop transformations. In contrast, the index-set splitting procedure developed in this work addresses the problem of systematically achieving separation of iteration spaces according to a given criterion. In this work, the criterion is the same use count applies to writes in each split index set. Inspector-executor strategies have been used in prior work to perform start-time optimizations and exploit data structure and dependence properties not known at compile-time [33].

8. Conclusions

Decreasing transistor sizes, use of lower voltage levels, and smaller noise margins have increased the probability of multi-bit errors in the memory subsystem. Therefore, it is of increased interest to design efficient solutions to address this problem in software, especially as hardware does not typically have embedded multi-bit error detection and correction mechanisms. To this end, we have developed novel compiler techniques to instrument application programs with error detection codes that protect every memory reference at runtime. The experimental evaluation demonstrates that the proposed solutions have low overheads and are practical.

Acknowledgments

We would like to thank the reviewers for their detailed and insightful comments which helped us improve the presentation of the work. This work was supported, in part, by the U.S. Department of Energy's (DOE) Office of Science, Office of Advanced Scientific Computing Research, under award number 63823, and by the U.S. Department of Defense under award number 63810.

References

[1] A. Avizienis, G. C. Gilley, F. P. Mathur, D. A. Rennels, J. A. Rohr, and D. K. Rubin. The STAR (self-testing and repairing) computer: An investigation of the theory and practice of fault-tolerant computer design. *IEEE Transactions on Computers*, C-20(11), Nov 1971.

[2] C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam. Putting polyhedral loop transformations to work. In *Languages and Compilers for Parallel Computing*, 2004.

[3] R. Baumann. Soft errors in advanced computer systems. *Design & Test of Computers, IEEE*, 22(3), 2005.

[4] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. *Algorithmica*, 12(2-3), 1994.

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

[6] J. D. Bright, G. F. Sullivan, and G. M. Masson. Checking the integrity of trees. In *Fault-Tolerant Computing*, 1995.

[7] G. Chen, M. Kandemir, and M. Karakoy. A data-centric approach to checksum reuse for array-intensive applications. In *International Conference on Dependable Systems and Networks*, 2005.

[8] R. G. Dreslinski, M. Wieckowski, D. Blaauw, D. Sylvester, and T. Mudge. Near-threshold computing: Reclaiming moore's law through energy efficient integrated circuits. *Proceedings of the IEEE*, 98(2), 2010.

[9] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1), 1991.

[10] P. Feautrier. Some efficient solutions to the affine scheduling problem: I. one-dimensional time. *International journal of parallel programming*, 21(5), 1992.

[11] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming*, 34(3), 2006.

[12] B. T. Gold, M. Ferdman, B. Falsafi, and K. Mai. Mitigating multi-bit soft errors in L1 caches using last-store prediction. In *Workshop on Architectural Support for Gigascale Integration*, 2007.

[13] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, and M. Violante. Soft-error detection using control flow assertions. In *Defect and Fault Tolerance in VLSI Systems*, 2003.

[14] M. Gomaa, C. Scarbrough, T. Vijaykumar, and I. Pomeranz. Transient-fault recovery for chip multiprocessors. In *Computer Architecture*, 2003.

[15] M. Griebl, P. Feautrier, and C. Lengauer. Index set splitting. *International Journal of Parallel Programming*, 28(6), 2000.

[16] S. K. S. Hari, S. V. Adve, and H. Naeimi. Low-cost program-level detectors for reducing silent data corruptions. In *International Conference on Dependable Systems and Networks*, 2012.

[17] S. K. S. Hari, S. V. Adve, H. Naeimi, and P. Ramachandran. Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults. In *Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2012.

[18] K.-H. Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers*, 100(6), 1984.

[19] ISL: Integer Set Library. `http://garage.kotnet.org/~skimo/isl/`.

[20] Y. Liang, Y. Zhang, M. Jette, A. Sivasubramaniam, and R. Sahoo. BlueGene/L failure analysis and prediction models. In *International Conference on Dependable Systems and Networks*, 2006.

[21] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn. Flikker: Saving dram refresh-power through critical data partitioning. In *Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2011.

[22] J. Maiz, S. Hareland, K. Zhang, and P. Armstrong. Characterization of multi-bit soft error events in advanced SRAMs. In *IEEE International Electron Devices Meeting*, 2003.

[23] T. C. Maxino. The effectiveness of checksums for embedded networks. Master's thesis, Carnegie Mellon University, 2006.

[24] S. E. Michalak, K. W. Harris, N. W. Hengartner, B. E. Takala, and S. A. Wender. Predicting the number of fatal soft errors in los alamos national laboratory's ASC Q supercomputer. *IEEE Transactions on Device and Materials Reliability*, 5(3), 2005.

[25] J. Nickolls and W. J. Dally. The GPU computing era. *IEEE micro*, 30(2), 2010.

[26] M. Nicolaidis. Efficient implementations of self-checking adders and ALUs. In *Fault-Tolerant Computing*, 1993.

[27] N. Oh, P. P. Shirvani, and E. J. McCluskey. Control-flow checking by software signatures. *IEEE Transactions on Reliability*, 51(1), 2002.

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

[29] K. Osada, K. Yamaguchi, Y. Saitoh, and T. Kawahara. SRAM immunity to cosmic-ray-induced multierrors based on analysis of an induced parasitic bipolar effect. *IEEE Journal of Solid-State Circuits*, 39(5), 2004.

[30] T. Osada and M. Godwin. International technology roadmap for semiconductors. 1999.

[31] K. Pattabiraman, G. P. Saggese, D. Chen, Z. Kalbarczyk, and R. K. Iyer. Dynamic derivation of application-specific error detectors and their implementation in hardware. In *European Dependable Computing Conference*, 2006.

[32] PLUTO: A polyhedral automatic parallelizer and locality optimizer for multicores. `http://pluto-compiler.sourceforge.net`.

[33] R. Ponnusamy, J. Saltz, and A. Choudhary. Runtime compilation techniques for data partitioning and communication schedule reuse. In *Supercomputing*, 1993.

[34] H. Quinn and P. Graham. Terrestrial-based radiation upsets: A cautionary tale. In *Field-Programmable Custom Computing Machines*, 2005.

[35] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software implemented fault tolerance. In *Code generation and optimization*, 2005.

[36] E. Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *Fault-Tolerant Computing*, 1999.

[37] B. Schroeder, E. Pinheiro, and W.-D. Weber. DRAM errors in the wild: a large-scale field study. In *Measurement and modeling of computer systems*, 2009.

[38] P. P. Shirvani, N. R. Saxena, and E. J. McCluskey. Software-implemented EDAC protection against SEUs. *IEEE Transactions on Reliability*, 49(3), 2000.

[39] A. Shye, T. Moseley, V. J. Reddi, J. Blomstedt, and D. A. Connors. Using process-level redundancy to exploit multiple cores for transient fault tolerance. In *Dependable Systems and Networks*, 2007.

[40] S. Verdoolaege. isl: An integer set library for the polyhedral model. *Mathematical Software–ICMS 2010*, 2010.

[41] N. J. Wang and S. J. Patel. ReStore: Symptom-based soft error detection in microprocessors. *IEEE Transactions on Dependable and Secure Computing*, 3(3), 2006.

[42] D. H. Yoon and M. Erez. Flexible cache error protection using an ECC FIFO. In *High Performance Computing Networking, Storage and Analysis*, SC, 2009.

[43] D. H. Yoon and M. Erez. Memory mapped ECC: low-cost error protection for last level caches. In *International Symposium on Computer Architecture*, ISCA, 2009.

[44] J. F. Ziegler, H. W. Curtis, H. P. Muhlfeld, C. J. Montrose, and B. Chin. IBM experiments in soft fails in computer electronics (1978–1994). *IBM journal of research and development*, 40(1), 1996.