Technical report OSU-CISRC-1/14-TR02

Compiler-Assisted Detection of Transient Memory Errors

Sanket Tavarageri
Department of Computer Science and Engineering
The Ohio State University
tavarageri.1@osu.edu

Sriram Krishnamoorthy
High Performance Computing Group
Pacific Northwest National Laboratory
sriram@pnnl.gov

P Sadayappan
Department of Computer Science and Engineering
The Ohio State University
sadayappan.1@osu.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 requires 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 so as to detect memory errors. Unlike prior approaches that employ checksums on computational and architectural state, 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.

1. Introduction

Trends in technology scaling have increased the likelihood of transient faults in various hardware components due to particle strikes [3, 5, 28, 31] and environmental factors [22, 41]. 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. Such transient faults in the memory subsystem result in bit flips that are potentially undetectable and lead to silent data corruption.

Hardware approaches to 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 every 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 pre-allocation of of hardware resources, incurring dynamic power and latency costs to support the worst case fault scenario to be tolerated. Commodity computing platforms with less-protected memory subsystems might nevertheless 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. For example, L1 caches in BlueGene/L [19] and GPU memory on pre-Fermi Nvidia GPUs [23] and pre-Tahiti AMD GPUs with parities. Multi-bit errors can often evade hardware detection mechanisms and 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 such errors in the memory sub-system. 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 the 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; and
- 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 are typically checked on every access. Algorithm-based fault tolerance approaches employ checksums for matrices which are maintained as part of the algorithm execution.

All these schemes attempt to associate checksums with data elements. This requires additional operations 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.

In this paper, we present 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 used 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 architectural characteristics of the memory subsystem. However, use of such a checksum scheme in practice requires several challenges to be overcome. Scaling the definition checksums with the use counts for each variable 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 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 as compared to memory resources. We therefore also consider the possibility of hardware support in the processor to assist in the checksum computation.

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.; (b) errors in address generation that result in incorrect data location being operated upon. This implies that an error caused by a fault can be transient or persistent. 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 the processor's subsystems other than memory are resilient to faults. This includes registers, ALUs, 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. We therefore focus on the detecting errors in a variable between the time it is written and later read.

```
temp = 10 + 20;
sum1 = temp + 30;
sum2 = temp + 40;
```

a) Original code

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

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

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

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

b) Error detection checksum (EDC) augmented code

Figure 1: Illustration of insertion of error detection codes

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

In this section, we describe the mathematical notation used and provide background information on polyhedral dependences.

3.1 Notation

We use the same notation as used by Verdoolaege [37] 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 the statements can be represented as sets. For example, the iteration space of statement S1 in the code shown in Fig. 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 set - I^{S_2}: $I^{S_2} = \{S2[j,i] : (0 \leq j \leq n-1) \wedge (j+1 \leq i \leq n-1)\}$.

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 the loops are specified as relations from iteration points to array indexes. E.g., the write reference - A[j][j] of statement 1 (S1) of the code shown in Fig. 2 is characterized as: $r^{S_1}_{write} = \{S1[j] \mapsto A[j,j]\}$.

Data dependences appearing in the program are also expressed as relations between statements. For example, the flow (Read After Write - RAW) dependence between write reference A[j][j] of statement 1 (S1) and read reference A[j][j] of statement 2 (S2) of the example code is represented by the following relation:
$d_{flow} = \{S1[j] \mapsto S2[j,i] : (0 \leq j \leq n-1) \wedge (j+1 \leq i \leq n-1)\}$

```
for (j = 0; j <= n-1; j++) {
   S1: A[j][j] = sqrt (A[j][j]);
   for (i = j+1; i <= n-1; i++) {
      S2: A[i][j] = A[i][j]/A[j][j];
   }
}
```

Figure 2: An example affine code snippet

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. E.g., let us say, the source iteration of interest is: $source\_iteration = \{S1[10]\}$, i.e., the dynamic instance of S1 when the value of loop iterator j is 10. Its target iterations due to the above flow dependence - d_{flow} - can be found as follows:
$d_{flow}(source\_iteration) = \{S2[10,i] : 11 \leq i \leq n-1\}$.

Schedules The order of execution of statements in a given program is encoded using 2d+1 schedules [10], 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.

The schedules for statements S1 and S2 for the working example are shown below.

$$\begin{aligned} \{ \quad & S1[j] \mapsto & [0,j,0,0,0]; \\ & S2[j,i] \mapsto & [0,j,1,i,0]\} \end{aligned}$$

3.2 Polyhedral Dependences

In the polyhedral model [9] for affine computations, dependence analysis [8] can precisely compute flow (Read After Write - RAW) and output (Write After Write - WAW) 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 [18]) for the code in Fig. 2 generates flow and output dependences. The flow dependence is shown below (same as the one presented earlier).

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

It characterizes the flow dependence that exists between the write reference – A[j][j] of S1 and the read reference – A[j][j] of S2: the value written by S1 at a particular iteration of *j* loop is used by S2 in the same iteration of *j* loop, and in all iterations of *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 from a memory cell 'X', then, the corresponding source iteration is the *last-writer* to the memory cell 'X' (and, not any of the prior writers).

3.3 Compiler Algorithm for Affine References

In this section, we develop compiler algorithms to compute use counts for affine references. Affine references are those that 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], over 99% of loops in 7 out of the 12 programs of the SpecFP2000 and Perfect Club benchmarks are affine loops.

3.3.1 Computing Contributions to def Checksum

Algorithm 1 describes how we determine the number of uses of a definition – *use_count*. For each statement that produces a value (the definition statement), we find the number of its consumers.

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

The iterators of the surrounding loops of the statement under consideration are parameterized (line 3) - I_i^{param} - so that we are now referring to a single parameterized iteration of the statement. Then, in the flow dependences which map source iterations to their target iterations, I_i^{param} is substituted as the source iteration and its target iterations are found (line 4). The cardinality of the target iteration set provides the *use_count* of the statement (line 5). The algorithm returns a collection of such statement and *use_count* pairs.

Example: The *use_count* for statement $S1$ of the example code shown in Fig. 2 is computed as follows.

$I_1^{param} = \{S1[j] : j = jp\}$ where jp is a parameter
$\mathcal{T}argets_i^{param} = \mathcal{D}_{flow}(I_i^{param}) = \{S2[jp,i] : (0 \leq jp \leq n-1)$
$\wedge (jp+1 \leq i \leq n-1)\}$
$|\mathcal{T}argets_i^{param}| = \{n-1-jp : (0 \leq jp \leq n-2)\}$

where $\mathcal{D}_{flow}$ is the flow dependence discussed in §3.2 for the example code. The value written by write reference - A[j][j] of S1 is used by read reference - A[j][j] of S2 in n-1-j following iterations (as the lower-bound of loop 'i' is j+1, and the upper-bound is n-1, which correspond to n-1-j iterations). Thus, the *use_count* of S1 is n-1-j as determined by the algorithm. Further, it can be noticed that the *use_count* output above applies to iterations of 'j' loop up to n-2 and the last iteration - when 'j' is n-1, is excluded. This is because, the last iteration has no target iterations of S2 : when j=n-1, the lower-bound of 'i' loop becomes n which is greater than its upper-bound - n-1, and hence, instance of S2 is not executed at j=n-1.

```
for (j = 0; j <= n-1; j++) {
  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; i <= n-1; i++) {
      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 3: Example affine code snippet with checksum augmentation

The checksum-inserted version of the example code is shown in Fig. 3. Given that the use counts are known at the point of each definition, the scaling factors can be incorporated at the definition point. This enables the use a single scalar each for the definition and use checksums. The *add_to_chksm* macro definition takes three parameters: first, the checksum to add to (def/use checksum), second, the value to add, and third, the number of times the value is to be added.

3.3.2 Optimization: Index Set Splitting

In the checksum-computation inserted code shown in Fig. 3, following statement S1, the error detection code that adds the value written by statement S1 – A[j][j] to the *def_checksum* has a branching structure: only for iterations of 'j' loop up to n-2, A[j][j] is added to *def_checksum* n-1-j times.

```
for (j = 0; j <= n-2; j++) { /*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; i <= n-1; i++) {
        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 4: Example affine code with index-set splitting optimization

To minimize performance penalty due to such control overheads, iteration space of 'j' loop may be split so that in each of the split iteration spaces, A[j][j] has the same *use_count* and thus, the need to evaluate 'if' conditionals in each iteration of 'j' loop is avoided. The loop-partitioned code thus formed is shown in Fig. 4. The last iteration of 'j' loop – when j = n-1 is peeled from the rest of the iterations. We also note that in the peeled iteration, S2 does not appear because at j=n-1, the lower-bound of 'i' loop - j+1 becomes n and is greater than its upper-bound which is n-1, and hence no instance of S2 gets executed at j=n-1.

Algorithm 2 Split Iteration Spaces

Input: Iteration Spaces : $I^{in}_{S_j}$, Index Sets : δ_{S_i}, Schedules : θ
Output: Loop Nest : $\mathcal{L}$

1: **for all** Iteration Spaces : $I^{in}_{S_j}$ **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: $I^{out}_{S_j} \leftarrow$ Split indexes of $I^{in}_{S_j}$ s.t ($\mathcal{R}ange$ of $Iterators_{common} \in I^{in}_{S_j}$) $\subseteq$ ($\mathcal{R}ange$ of $Iterators_{common} \in \delta_{S_i}$)
5: **end if**
6: **end for**
7: **end for**
8: $I^{out}_{S_j} \leftarrow I^{out}_{S_j} \cup (I^{in}_{S_j} \setminus I^{out}_{S_j})$
9: $\mathcal{L} \leftarrow$ Generate Code to traverse $I^{out}_{S_j}$ with schedule θ
10: **return** $\mathcal{L}$

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

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). E.g., for the example code, any split of loop 'j' affects iteration spaces of both S1, and S2. However, if only 'i' loop is to be broken up, then its loop splitting does not affect iteration space of S1 as 'i' is not a surrounding loop for S1. Then, for the common iterators found, iteration space is split so that 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 resultant smaller iteration spaces constitute $I^{out}_{S_j}$.

If any iteration points are not yet a part of $I^{out}_{S_j}$, then they are added to $I^{out}_{S_j}$ so that all the iteration points contained in $I^{in}_{S_j}$ are included in $I^{out}_{S_j}$ as well (line 8). Finally, the loop-nest code is generated by traversing through $I^{out}_{S_j}$ according to the schedule θ (line 9). The output of the algorithm is the index-split loop-nest.

4. Inspectors for Dynamic Start-time Use Count Determination

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

4.1 Basic Approach

```
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) EDC added code

Figure 5: Code snippet illustrating the general EDC scheme

Consider the code shown in Figure 5 a), that we use to illustrate the challenges and solution approach for irregular codes. The variable *temp* is defined, and 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 used at all. We proceed as follows. At the def-site, *temp* is added to *def_checksum*. At the use-site, *temp* is added to *use_checksum*, and a counter (*temp_use_count*) is incremented to keep track of the total number of times the variable gets used. Finally to match the checksums, value of *temp* is added to

def_checksum, (*temp_use_count* − 1) number of times (because it is added once already to the *def_checksum* at the def-site). If there were no memory errors, then the two checksums match. Note that subtracting from the use checksum might subtract out the erroneous values and leave them being undetected.

Doing only the above, it turns out, is not sufficient to catch all memory errors: we describe the issue with a concrete example. Let both the conditionals in the code shown be evaluated to true, and thus there be a total of two uses of *temp*. Let us further assume the first read was correct. At this point, $def\_checksum = temp$, and $use\_checksum = temp$. 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 $temp + temp'$. And, at epilogue, $temp'$ gets added to *def_checksum* (as *temp_use_count* would be 2), which makes $def\_checksum = temp + temp'$, 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 might get added to both checksums and thus 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 (and, not at each use-site). The error detection code generated using the scheme described is shown in Figure 5 b). We note that the problem explained above now gets fixed: $e\_def\_checksum = temp$, but $e\_use\_checksum = temp'$, and the memory error gets exposed.

The reason the modified scheme with additional checksums works is, at the end, when the value of the variable is added to *def_checksum* to match the number of its uses, *the auxiliary checksum confirms that the value that gets added to def_checksum is what was defined, and not a potentially corrupted value*.

4.2 Optimizations for Iterative Codes

Figure 6 shows an example of a code requiring dynamic support: accesses to array p_new at S1 are data-dependent. Further, number of iterations of the loop is dynamically determined. We describe how we generate EDC for the example focusing on two arrays – p_new, and cols – followed by description of a general scheme.

```
while(converged == false) {
  for (j_1=0;j_1<n;j_1++) {
    S1: temp1 += p_new[cols[j_1]];
  }
  for(j_2=0;j_2<n;j_2++) {
    S2: temp2 += p_new[j_2];
  }
  for(j_3=0;j_3<n;j_3++) {
    S3: p_new[j_3] = temp3;
  }
 /*convergence check not shown*/
}
```

Figure 6: Example code requiring dynamic use count determination

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

- The reads to p_new at 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.
- The elements of p_new read at S2 are amenable to compile-time analysis as the array index expressions and loop bounds of enclosed loop indexed by j_2 are affine.
- The new definition of array p_new at S3 is used a *fixed number of times* (even though not known at compile-time) before being overwritten in the next iteration of the while loop, if the algorithm has not converged.

We use an *inspector* to examine the cells of array p_new that will be read because of data-dependent accesses at S1. Since 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 previous sections, 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 apriori. 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 *e_def_checksum* (auxiliary checksum used for correctness) once, and at all read locations, it 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, and once to *e_use_checksum* to balance contributions to the def, and use checksums. This approach protects all uses of the defined value.

```
// Inspector
for (j_1=0;j_1<n;j_1++) {
  count_p_new[cols[j_1]]++;
}
iter = 0;
while(converged == false) {
  iter++;
  for (j_1=0;j_1<n;j_1++) {
    add_to_chksm(use_cs,cols[j_1],1);
    add_to_chksm(use_cs,
                 p_new[cols[j_1]],1);
    S1: temp1 += p_new[cols[j_1]];
  }
  for(j_2=0;j_2<n;j_2++) {
    add_to_chksm(use_cs, p_new[j_2], 1);
    S2: temp2 += p_new[j_2];
  }
  for(j_3=0;j_3<n;j_3++) {
    S3: p_new[j_3] = temp3;
    add_to_chksm(def_cs,p_new[j_3],
                 count_p_new[j_3]+1);
  }
  /*convergence check not shown*/
}

// Epilogue
for (i=0; i<n; i++) {
  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 7: Example code with inspector to determine use counts and generalized EDC scheme

Figure 7 shows the EDC-inserted version of the example in Figure 6. The inspector code counts the number of accesses to data-dependent references of array p_new. The loop is instrumented with *def_checksum*, and *use_checksum* computation operations,

and 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 as well to account for the fact that the new definition of p_new in the last iteration of the while loop goes unused.

5. Overall Compiler Algorithm for Transient Error Detection

The approach is based on using one global checksum to track all definitions (assignments) of values to data elements and another global checksum to track all uses of the elements; a mismatch between the two checksums indicates the occurrence of one or more data corruption errors.

Algorithm 3 Insert Error Detection Codes

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 Error Detection Codes (EDCs). At each use-site, the read values are added to *use_checksum*; and use-counts are incremented if the reads are in irregular accesses. At a def-site, if information on the number of the uses can be pre-determined (Section 3 describes a static analysis to compute this information for affine array references), then 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, a different approach is used to manage the checksums: the newly defined value is added once to $def\_checksum$, and once to an auxiliary checksum $e\_def\_checksum$, to be adjusted in the epilogue code. A use-count is also maintained and initialized to zero. But, 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 is also added to update $e\_use\_checksum$ (Section 4 develops optimizations for such memory references.) At the end of the program, the verifier code that compares def, and *use* checksums is introduced to detect any memory errors that might have occurred to any variables during execution of the program.

Checksum Function 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/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 XOR. However, addition is known to be superior to XOR in terms of fault coverage [21].

An inherent property of checksums is that they can trigger false negatives: errors canceling each other out and resulting in a correct checksum value. The add operator with a finite bit-width also potentially suffers from wrap-around cancellation errors. However, in Section 6, we show that probability of such false negatives is extremely low and errors are detected with high confidence. Further, in Section 6.1.2, we show that by using multiple checksums, false negatives can be virtually eliminated.

Theorem 5.1. *The checksum computation codes inserted by Algorithm 3 correctly detect all memory errors with a 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 – checksums match if there are no memory errors.

We first consider the case where there is a single variable in the program and then extend the analysis to show that the scheme works for multiple variables.

Single variable scenario: Let the value assigned to the variable at def-site be v and it be used n times. Now we can have two cases: 1) number of uses of the variable – *use_count* 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* is known (to be n) at compile-time, value v is multiplied by n and is added to $def\_checksum$. Therefore, $def\_checksum = n \times v$. We analyze what happens to checksums for different values of n and in the presence of errors.

i) $n = 0$: $def\_checksum = 0$ and $use\_checksum = 0$. Since number of uses is zero, data errors do not happen.

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

iii) $n > 1$: Let the values read during 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 then each v_i will be equal to v and the two checksums match.

If any of the reads is wrong, then with a high probability $def\_checksum \neq use\_checksum$ and the error is detected. A false negative can happen when $n \times v = v_1 + v_2 + \cdots + v_n$ even though some v_i's are different from v.

Table 1: Percentage of Undetected Errors with ADD checksum

#bit-flips	N	One Checksum			Two Checksums		
		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%

Case 2 — unknown use_count: At the def-site, value v is used to initialize def_ checksums: $def\_checksum = v$ and $e\_def\_checksum = v$.

i) $n = 0$: Since there are no uses, $use\_checksum = 0$ and $use\_count = 0$. At the epilogue code or just before the variable is overwritten, value v is added to $def\_checksum$ $use\_count - 1$ times which is -1 times. Therefore, $def\_checksum = v - v = 0$. Also, 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.

ii) $n = 1$: When there is only one use, $def\_checksum$ is not adjusted since $use\_count - 1 = 0$. Memory errors are detected with certainty when $n = 1$.

iii) $n > 1$: Let the values read during n reads be: $v_1, v_2, \ldots, v_n$. Therefore, $use\_checksum = v_1 + v_2 + \cdots + v_n$. At the time of adjusting of $def\_checksum$, 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, then 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 if $v \neq v_{n+1}$.

Use of *e_checksums* prevents an important class of errors going undetected: If after 1^{st} use, v changes to v' and the error persists, then $def\_checksum = use\_checksum = v + (n-1) \times v'$. But, because auxiliary checksums are being used, they correctly flag the error: $e\_def\_checksum = v$ and $e\_use\_checksum = v'$ and they do not match.

Multiple variables: Because the checksum operator is commutative and associative, an analysis similar to the one above establishes 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

6.1 Fault Coverage

A characteristic of the use of checksums for error detection is that errors in multiple values contributing to a checksum could cancel one another and thus, produce a seemingly correct checksum even though there were bit-flips in the data. In the following set of experiments, we quantify the percentage of errors that escape detection.

One-bit errors are always caught – only multi-bit errors can potentially go undetected. Therefore, we inject multi-bit errors into an array of 64-bit integers, and the percentage of cases in which checksums are successful in catching the faults is monitored. Over 100,000 trials, the following steps are repeated: The data are initialized, and initial 64-bit checksum is computed. Then, either two, three, four, five, or six bits over all bits of the array are uniformly randomly selected and values of those bits are flipped. A 64-bit checksum is again computed and is compared with the initial checksum. If there is a mismatch in the checksum values, then the injected error has been correctly caught, otherwise error has escaped detection. The percentage 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 obtained were very similar to what we report here for 100,000 trials.)

6.1.1 Effectiveness of Checksums

Table 1 shows the percentage of cases in which a multi-bit error (2, 3, 4, 5, 6 bit-flips) is not caught for different array sizes – $10^2, 10^4, 10^6$ under the column header "One Checksum". Experiments are carried out over three kinds of data: all bits of the array elements are 0's; all bits are 1's; bits are randomly initialized.

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

Among different types of data values, percentage of undetected errors is highest for random values. When bits of all array elements are initialized to either 0's, or 1's, if bits at the same bit-position in two array elements flip, the carry bit is also changed, and the error will be caught unless the bit in the next bit-position is also flipped. In contrast, random data are likely to have an equal number of 0's and 1's, and when a 0 becomes 1, and 1 becomes 0 at the same bit-position in two values, it will not change the carry bit, and thus the error can go undetected with a higher probability.

Table 2: Benchmarks

Benchmark	Description	Problem Size
ADI	Alternating Direction Implicit solver	TSteps = 500, N = 3,000
CG	Conjugate Gradient	TSteps = 1500, NZ = 513,072
cholesky	Cholesky decomposition	N = 3,000
dsyrk	Symmetric rank-k update	N = 3,000
jacobi1d	1-D Jacobi stencil computation	TSteps = 100,000, N = 400,000
LU	LU decomposition	N = 3,000
moldyn	Molecular dynamics	TSteps = 100,000, N = 400,000
seidel	2-D Seidel stencil	TSteps = 500, N = 3,000
strsm	Triangular matrix equations solver	N = 3,000
trisolv	Triangular system of linear equations solver	N = 3,000

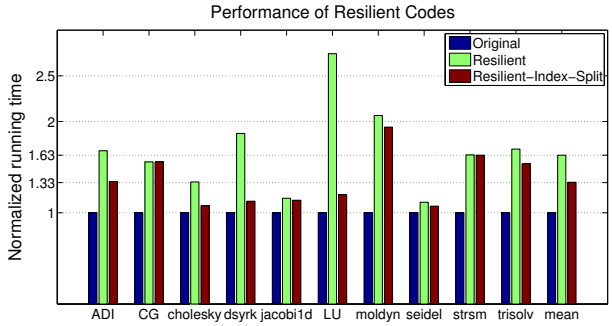

Figure 8: Performance of resilient codes with one checksum – software-only solution. y-axis employs a linear scale with selective tick marks to highlight the costs.

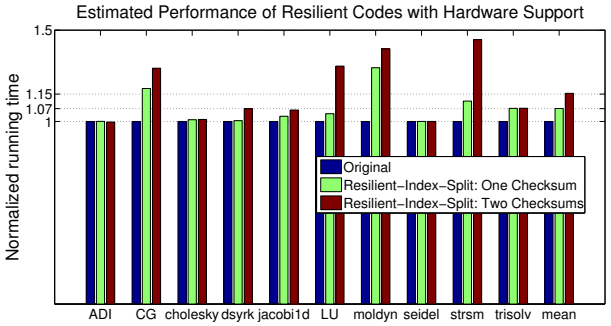

Figure 9: Estimated performance of resilient codes with a special function unit to compute checksums. y-axis employs a linear scale with selective tick marks to highlight the costs.

Further, we observe that the performance of checksums is largely independent of the number of values that go into forming the checksum. In over 99% of cases, errors are successfully detected.

6.1.2 Use of Multiple Checksums

The odds of error detection can be further improved and protection against data corruption fortified with the use of multiple checksums: the first checksum is a direct sum of array values as before, but other checksums are obtained by adding permutations of array values.

We evaluate the fault coverage when using two checksums: 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, error has been detected. While forming the second checksum, each array element is left-rotated by an amount determined by 5 bits of the address of that array element, viz., 4^{th} to 8^{th} least significant bits. (The address of a 64-bit integer, which occupies 8 bytes, will likely to be a multiple of 8. Hence, the first three least significant bits – 1^{st} to 3^{rd} are likely to be always zeros). Thus, each value that goes into computation of 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 array values are rotated by different amounts, which prevents the lined-up erroneous bits in one checksum from lining up to cancel one another out after rotation in the other checksum also. The percentages of undetected errors using the two checksum scheme are shown in Table 1 under the column header "Two Checksums". Only a very small percentage of 2-bit errors are undetected, and all 3-, 4-, 5-, 6- bit errors are detected.

6.2 Performance Overheads

Benchmarks: Table 2 lists the benchmark programs and problem sizes used to measure performance overheads of resilient codes. All array references in ADI, cholesky, dsyrk, jacobi-1d, LU, seidel, strmm, trisolv are statically analyzable, whereas a subset of array references in CG, and moldyn are irregular and for those references, dynamic analysis techniques are applied.

Machine Configuration, and Compiler: Experiments are performed on Intel Xeon E5630 processors, running at 2.53GHz with 32KB L1 cache. The programs are compiled using Intel icc 13.1.3 compiler, with `-O3` optimization flag. Each executable is run five times, and the average running time of those five runs is used in reporting the results.

6.2.1 One Checksum, Software-Only Solution

Figure 8 shows performance of resilient programs that have checksum computation codes embedded in them relative to performance of original non-resilient versions. The geometric mean of performance overheads of resilient codes across all benchmarks is 63.0%. When index-splitting optimization presented in §3.3.2 is applied, performance overheads are reduced to 33.4% on average (geometric mean).

The index-splitting transformation partitions a loop so that statements in a loop have the same *use_count*, and hence, removes any conditional statements introduced in the loop because the *use_count* of a definition changes based on values of a loop iterator. Index-splitting thus reduces control overheads and improves performance. Further, index-splitting enables vectorization

if the compiler was not able to vectorize a loop because conditional statements were present in the loop. E.g., original LU code is vectorized – its running time is 11.1 seconds, however its resilient version is not vectorized, and its running time is 30.3 seconds. When index-splitting transformation is applied to the resilient code, the icc compiler successfully vectorizes it, and the resulting running time is 13.2 seconds.

The highest overhead is observed for 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 keep track of the number of uses of array elements: at a use site, the use-count for the array-cell is incremented, and at a write site, the old value is used to adjust checksums, and new value is added to def-checksum.

The CG benchmark is an iterative method that includes sparse matrix-vector multiplications. Even though 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 for CG. The overheads are therefore smaller for CG compared to moldyn, which also has irregular accesses.

6.2.2 One Checksum, Two Checksums with Hardware Support

Checksum computation when multiple checksums are employed, is a costly operation without assistance from hardware: at a definition site, the newly produced value is multiplied by its *use_count*, and the resulting value is added to the first checksum – it involves a multiplication and an addition operation. Constructing the second checksum requires extraction of certain bits from the address of a variable, permutation of the value and then finally, a multiplication by the *use_count*, and addition to the checksum value.

Therefore, we propose addition of a special function unit (SFU) to hardware to compute checksums. We evaluate effect of addition of checksum SFU on performance by substituting checksum operations with some other instructions that model how checksum instructions proceed in the instruction pipeline except at the Execute stage, viz., Fetch, Decode, and Write-back stages.

We assume that the number of cycles the checksum SFU takes to add a value to a checksum is no greater than the number of cycles to perform an addition operation, and hence, estimate performance of resilient codes with hardware support by substituting a checksum computation instruction with an instruction that adds a constant value to the checksum. Thus, resilient codes that use one checksum will have in lieu of checksum computations, instructions that increment checksums; resilient codes that employ two checksums will have twice as many such checksum-incrementing instructions. With this set-up, we compile the programs with icc compiler, and execute them on Intel Xeon processors.

Figure 9 plots the estimated performance overheads of resilient codes on systems that have the proposed checksum SFUs. The geometric mean of performance overheads across benchmarks of resilient codes that use one checksum is 7.1%, and that use two checksums is 15.4%. Performance of resilient versions of those benchmarks whose references can be analyzed completely statically is very close to performance of non-resilient original codes. Addition of the checksum SFU takes away most of the overheads of resilient codes. The benchmarks that have irregular accesses, namely CG and moldyn also benefit from hardware support; the cost of inspectors inserted in them for dynamic analysis however does not change and therefore, CG and moldyn have slightly higher overheads compared to other benchmarks.

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 [24] and hardware duplication [1] provide the most general coverage, but 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 micro-architecture support. This includes approaches for simultaneous multithreading [13] and chip multiprocessors [33] that employ redundantly executing threads whose results are periodically compared. Process-level redundancy [36] 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 [26] or control flow [25], while duplicating all application state. SWIFT [32] checks the computation and control state without duplicating application state and assumes that memory is made fault-tolerant to other means such as ECC. The approach presented in this paper can complement SWIFT by decoupling correctness checks for the memory subsystem from that 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 [15] as the outcome of underlying hardware faults. These solutions can be software-only [12, 38] or combine hardware support [29] 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. [16] observe 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. [34] 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 [20, 27]. Yoon et al. [39, 40] 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. [11] observe that multi-bit error detection and correction in L1 caches is more expensive in terms of performance, power, and area even for reasonable sizes.

Shirvani et al. [35] designed approaches to provide checksum protection by periodically scrubbing memory, rather than check every read and write operation, which lowers fault coverage as 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 [17]. Chen at al. [7] optimize the management of checksums for such 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.

Maxino [21] evaluates error detection effectiveness of different checksum algorithms, namely, XOR, two's complement addition, one's complement addition, Fletcher checksum, Adler checksum, and Cyclic Redundancy Codes.

Compile-time Analysis and Transformation Griebl et al. [14] use the index-set splitting approach to form regular dependence structures so that effective loop transformations can be applied. As there can be a large number of ways of splitting loops, they address the problem of how to efficiently find index sets that do yield good loop transformations. The index-set splitting procedure developed in this work, on the other hand, addresses the problem of systematically achieving separation of iteration spaces according to a given criterion (the criterion for us is achieving the same *use-counts* for writes in a split index set). Inspector-executor strategies have been used to perform start-time optimizations for that can exploit data structure and dependence properties not known at compile-time [30].

8. Conclusion

The 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 as hardware does not typically have embedded multi-bit error detection and correction mechanisms. Towards that end, we developed novel compiler techniques to instrument application programs with error detection codes that at runtime protect every memory reference. The experimental evaluation demonstrates that the proposed solutions have low overheads and are practical.

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*, 1971.

[2] C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam. Putting polyhedral loop transformations to work. In *LCPC*, 2003.

[3] R. Baumann. Soft errors in advanced computer systems. *Design & Test of Computers, IEEE*, 22(3):258–266, 2005.

[4] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. *Algorithmica*, 12(2-3):225–244, 1994.

[5] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *Micro, IEEE*, 25(6):10–16, 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 *DSN*, 2005.

[8] P. Feautrier. Dataflow analysis of array and scalar references. *IJPP*, 20(1), 1991.

[9] P. Feautrier. Some efficient solutions to the affine scheduling problem: I. one-dimensional time. *IJPP*, 21(5):313–348, 1992.

[10] 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. *Intl. J. of Parallel Programming*, 34(3), 2006.

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

[12] 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.

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

[14] M. Griebl, P. Feautrier, and C. Lengauer. Index set splitting. *International Journal of Parallel Programming*, 28(6):607–631, 2000.

[15] S. K. S. Hari, S. V. Adve, and H. Naeimi. Low-cost program-level detectors for reducing silent data corruptions. In *DSN*, 2012.

[16] 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 *ACM SIGARCH Computer Architecture News*, 2012.

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

[18] ISL: Integer Set Library, 2013. `http://garage.kotnet.org/~skimo/isl/`.

[19] Y. Liang, Y. Zhang, M. Jette, A. Sivasubramaniam, and R. Sahoo. BlueGene/L failure analysis and prediction models. In *DSN*, 2006.

[20] J. Maiz, S. Hareland, K. Zhang, and P. Armstrong. Characterization of multi-bit soft error events in advanced SRAMs. In *IEDM*, 2003.

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

[22] 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. *Device and Materials Reliability*, 2005.

[23] J. Nickolls and W. J. Dally. The GPU computing era. *Micro*, 2010.

[24] M. Nicolaidis. Efficient implementations of self-checking adders and ALUs. In *FTCS*, 1993.

[25] N. Oh, P. P. Shirvani, and E. J. McCluskey. Control-flow checking by software signatures. *Reliability*, 2002.

[26] N. Oh, P. P. Shirvani, and E. J. McCluskey. Error detection by duplicated instructions in super-scalar processors. *Reliability*, 2002.

[27] 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. *Solid-State Circuits*, 2004.

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

[29] 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 *EDCC*, 2006.

[30] R. Ponnusamy, J. Saltz, and A. Choudhary. Runtime compilation techniques for data partitioning and communication schedule reuse. In *Supercomputing'93. Proceedings*, pages 361–370. IEEE, 1993.

[31] H. Quinn and P. Graham. Terrestrial-based radiation upsets: A cautionary tale. In *FCCM*, 2005.

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

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

[34] 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.

[35] P. P. Shirvani, N. R. Saxena, and E. J. McCluskey. Software-implemented EDAC protection against SEUs. *Reliability*, 2000.

[36] 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.

[37] S. Verdoolaege. isl: An integer set library for the polyhedral model. *Mathematical Software–ICMS 2010*, pages 299–302, 2010.

[38] N. J. Wang and S. J. Patel. ReStore: Symptom-based soft error detection in microprocessors. *Dependable and Secure Computing*, 2006.

[39] D. H. Yoon and M. Erez. Flexible cache error protection using an ECC FIFO. In *SC*, 2009.

[40] D. H. Yoon and M. Erez. Memory mapped ecc: low-cost error protection for last level caches. In *ISCA*, 2009.

[41] J. Ziegler and et al. IBM experiments in soft fails in computer electronics (1978–1994). *IBM Journal of Research and Development*, 1996.