Using Loop Invariants to Fight Soft Errors in Data Caches

Sri Hari Krishna N. , Seung Woo Son, Mahmut Kandemir, Feihui Li
Department of Computer Science and Engineering
The Pennsylvania State University
{snarayan,sson,kandemir,feli}@cse.psu.edu

Abstract

Ever scaling process technology makes embedded systems more vulnerable to soft errors than in the past. One of the generic methods used to fight soft errors is based on duplicating instructions either in the spatial or temporal domain and then comparing the results to see whether they are different. This full duplication based scheme, though effective, is very expensive in terms of performance, power, and memory space. In this paper, we propose an alternate scheme based on loop invariants and present experimental results which show that our approach catches 62% of the errors caught by full duplication, when averaged over all benchmarks tested. In addition, it reduces the execution cycles and memory demand of the full duplication strategy by 80% and 4%, respectively.

This work was supported in part by NSF CAREER Award 0093082.

Using Loop Invariants to Fight Soft Errors in Data Caches

Sri Hari Krishna N. , Seung Woo Son, Mahmut Kandemir, Feihui Li
Department of Computer Science and Engineering
The Pennsylvania State University
{snarayan,sson,kandemir,feli}@cse.psu.edu

Abstract – Ever scaling process technology makes embedded systems more vulnerable to soft errors than in the past. One of the generic methods used to fight soft errors is based on duplicating instructions either in the spatial or temporal domain and then comparing the results to see whether they are different. This full duplication based scheme, though effective, is very expensive in terms of performance, power, and memory space. In this paper, we propose an alternate scheme based on loop invariants and present experimental results which show that our approach catches 62% of the errors caught by full duplication, when averaged over all benchmarks tested. In addition, it reduces the execution cycles and memory demand of the full duplication strategy by 80% and 4%, respectively.

I. INTRODUCTION

As process technology continues to scale, sizes of semiconductor devices are decreasing and data being stored in memory is becoming increasingly vulnerable to soft errors. Soft errors are a form of transient errors where radiation changes the logic value of a node by charging or discharging it. The eventual outcome of a soft error is a bit flip in memory or logic components of the system. There are at least two reasons that make embedded systems more vulnerable to soft errors as compared to their high-end general-purpose counterparts. The first reason is that many battery-operated embedded systems employ power-reducing techniques that increase chances for a particle to create a bit flip. Secondly, embedded systems are generally employed in much harsher environments than the general-purpose systems, exposing them to more radiation.

There exist both hardware and software solutions developed over the years to fight soft errors and other types of transient errors [1,2,3]. One of the generic methods is based on duplicating instructions either in the spatial or temporal domain. It involves executing two copies of each instruction and comparing the results, to see whether they are different. If so, an error is assumed and a corrective action may be taken. This full duplication based scheme, though effective, is very expensive in terms of performance, power, and memory space. From the performance angle, the extra instructions put additional pressure on limited system resources, and if their executions cannot be overlapped with those of the primary instructions, this can lead to an increase in execution time. Even if the time taken by the duplicate instructions is hidden entirely, these instructions still contribute to the power consumption, which cannot be hidden. In addition, memory demand is increased, which is also undesirable for memory-constrained embedded devices.

This paper explores a technique that allows us to detect a large portion of the soft errors that are detected by full duplication, in a less expensive fashion. Our focus is on loop-intensive embedded applications where a large fraction of the execution time is spent within loops. The proposed idea is based on exploiting *loop invariants,* which are expressions that hold true in all iterations of a given loop [4]. They are typically used to check whether the loop has been formed correctly, i.e., as a measure against software errors. In contrast, in this work, we employ loop invariants to detect soft errors. Specifically, at each iteration or at every k^{th} iteration of a loop nest, we check whether the loop invariant holds. While if the loop invariant fails, this can be an indication for an error anywhere in the hardware, our focus in this study is on data cache errors; i.e., the soft errors that occur in data caches.

Note that using loop invariants instead of duplicating the entire loop body brings at least two benefits. Firstly, computing loop invariants is typically less expensive than re-computing the entire loop body from both power and performance perspectives. Secondly, loop invariants normally occupy less memory space than loop bodies, which in turn reduces the memory space demand. However, the downside is that the loop invariants may not be able to catch all the soft errors caught by full duplication. This is because an invariant usually touches only a subset of the total data manipulated by the loop. In other words, just the fact that a loop invariant holds does *not* mean that no data manipulated by the loop has any soft error. This is the tradeoff studied in this work: *How many of the soft errors that could be caught by full duplication are caught by loop invariants, and what are the associated costs?*

Our experimental analysis shows that our loop invariant based approach catches 62% of the soft errors caught by full duplication, when averaged over all benchmarks tested, and that it reduces the execution cycles and memory demand of the full duplication strategy by 80% and 4%, respectively.

The remainder of the paper is organized as follows. Section II discusses related work. Section III presents how invariants are detected. Section IV gives an example and presents a classification of invariants. Section V presents experimental evaluation, and we conclude in Section VI.

II. RELATED WORK

Traditional approaches to protect against soft errors have been to use radiation-hardened technologies or to include an error detection and correction mechanism [1,2,3] using spatial redundancy, or repeating the execution on the same processor, i.e., temporal redundancy.

Pure software-based techniques to overcome transient errors are based on the replication of code either manually or automatically by the compiler; or on the insertion of control code into the source code. One of code duplication methods is error detection by duplicated instructions (EDDI) [3]. Similar to the hardware, these methods are also costly, i.e., they typically double the static code size and execution time [5]. Examples of the control code based methods include assertions [6], algorithm based fault tolerance [7], and code flow checking [8].

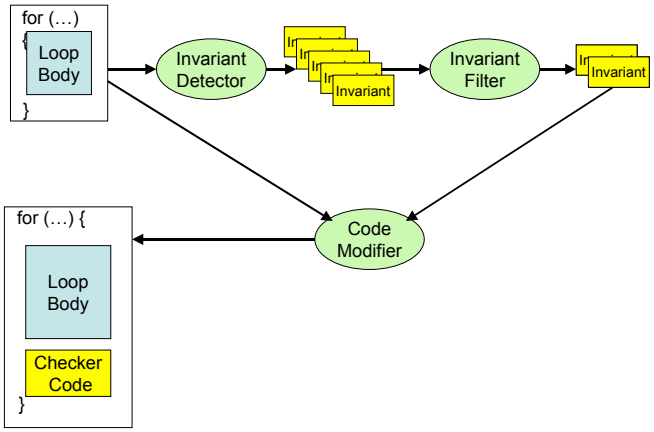

Fig. 1. Overview of our approach.

III. DETECTING LOOP INVARIANTS

The loop invariant is a value or property that does not change during the execution of different iterations of the same loop. For example, a property of the 'for loop' is that the value of the loop index is bounded by the lower and upper loop bound of the loop, assuming that the loop code does not itself explicitly change this value. There exist several publicly-available tools that can extract loop invariants for a large class of programs. Daikon is a dynamic invariant detector that detects likely invariants over the program's data structures [9]. In this work, we used Daikon to detect the invariants present in the loops.

Algorithm bubble-sort(*sequence*):
Input: sequence of integers stored in *sequence*
Post-condition: *sequence* is sorted and contains the same integers as the original sequence
length = length of *sequence*
for *i* = 0 **to** *length* - 1 **do**
  for *j* = 0 **to** *length* - *i* -2 **do**
    if *j*th element of *sequence* >(*j*+1)th element of *sequence*
    then
      swap *j*th and (*j*+1)th element of *sequence*

Fig. 2. Bubble-sort algorithm.

IV. APPROACH DETAILS AND EXAMPLE

Fig. 1 depicts an overview of our approach. We process the loop nests of the application being optimized one by one. This figure shows the scenario for one loop nest. We use Daikon to detect the invariants and then filter them to select the ones to be embedded in the code. Then we generate "checker code" using these selected invariants and embed them into the original loop code.

To explain our method in detail, we first consider a simple bubble-sort program. Fig. 2 shows the bubble-sort algorithm. The following loop invariants must be satisfied during the execution of each of the loop iterations.

- Outer Loop: Last *i* elements of the sequence are sorted and are all greater than or equal to the other elements of the sequence.
- Inner Loop: Same as outer loop and the jth element of the sequence is greater than or equal to the first *j-1* elements of the sequence.

A sample of the invariants detected by Daikon is listed in Fig. 3. It should be observed that not all the invariants detected by Daikon are useful for our purposes. For example, if an invariant does not touch much data, it is unlikely that it will catch any soft errors in the data cache. On the other hand, the invariant to be used should not be very complex either since such an invariant will not help us significantly reduce the power, performance, and memory space overheads of the full duplication scheme. Therefore, a balance must be struck in selecting the invariants to employ in checking the errors.

Based on this discussion, we classify the invariants into three groups: "easy", "medium", and "difficult". The easy invariants are the ones easy to compute but since they do not touch too much data, we do not use them in our checker code. The difficult invariants would be useful to employ for checking for soft errors; however, they are expensive to compute. Therefore, as far as the overheads are concerned, they are not any better than full duplication. Hence, we do not use them either within the checker code. The only type of invariants we employ then are the medium invariants, and these are the ones that help us study the tradeoffs between error coverage and associated overheads. However, even in this category, we do not use all the invariants detected; instead, we employ only a subset of them. To sum up, in our invariant filtering step, we select a subset of the medium invariants to be used in building the checker code.

As an example, consider some of the invariants (Fig. 3) extracted from the bubble-sort program in detail. In this example, we have four "easy" invariants and two "medium" invariants. An easy invariant is the one that would require checking the value of a scalar, or a single array element. A medium one would be one that involves a simple check of many of the array values. A more complicated one would involve a complex check on the array values or on the address locations, etc. As we can see, in this particular example, the invariants that fall into easy category are just loop bound conditions or simple comparisons between array elements and loop index. These kinds of invariants are easy to check at runtime as checking needs only one or two *if* statements; however as mentioned earlier, they are not very useful for our purposes. Next, we have two medium invariants. These invariants exactly describe the condition at the ith loop iteration. At first, these conditions look very complex to embed in a checker code. However, a closer look reveals that we can easily check these

conditions using an additional loop iteration.

In entering the bubble-sort routine:

sequence[] != null ⇒ easy

length of sequence[] =10 ⇒ easy

At the beginning of the kth iteration of the outer loop:

$\forall i : 0 \le i \le k$ *sequence[] sorted by >* ⇒ medium

$k \ge 1$ ⇒ easy

$k \le$ *(length of sequence[] –1)* ⇒ easy

At the end of the kth iteration of the outer loop:

$\forall i$: *length-k+1* $\le i \le$ *length sequence[] sorted by <* ⇒ medium

Fig. 3. Invariants detected by Daikon for bubble-sort.

Once the invariants to be used in the checker code are determined, the next step is to build the checker code. Basically, the selected invariants are embedded into the loop body and their value checked at each (or every k^{th}) loop iteration. When an error is detected, we jump to a routine called *error()*, where we may take a corrective action. If no correction is implemented, the *error()* routine can contain a stop statement that terminates the application by printing an error message. The augmented code thus obtained for the bubble-sort program is shown in Fig. 4. Due to space constraints, difficult invariants are not shown.

V. EXPERIMENTAL RESULTS

In this section, we present an experimental evaluation of our approach, and compare it to a full duplication based scheme, where the entire loop body is duplicated. To implement our fault injection mechanism, we modified SimpleScalar v3.0d [10] to inject faults and obtain statistics on the faults injected. Due to space limitations the details are not discussed here.

We tested our loop invariant based approach and full duplication using a set of five benchmark programs.: *iter-merge* (iterative merge program), *adi* (alternate direction integration application), *heap-sort* , *bubble-sort* , and *mxm*.

We focus on three metrics. The first one is the percentage error detection rate, i.e., the percentage of soft errors detected by any scheme. Clearly, one can expect close to 100% detection rates from the full duplication based scheme (missing perhaps only the errors that could occur in unduplicated parts of the code such as loop headers etc). We want to check how close our approach comes to the full duplication scheme. The second metric we are interested in is the percentage increase in code size (executable size) as a result of the extra instructions added for error protection purposes. We want this percentage to be as small as possible since most embedded systems are memory constrained. The last metric we measure is the percentage increase in execution cycles due to enhanced reliability.

```
Algorithm new-bubble-sort(sequence):
Post-condition: sequence is sorted and contains the
                same integers as the original sequence
length = length of sequence
for i = 0 to length - 1 do
  for j = 0 to length - i -2 do
    if jth element of sequence >(j+1)th element of sequence
    then
      swap jth and (j+1)th element of sequence
  for k = length - i to length do
    if jth element of sequence >(j+1)th element of sequence
    then
      error()
```

Fig. 4. The bubble-sort code with the "checker-code".

The graph in Fig. 5 gives the percentage error detection rates for our scheme and full duplication (i.e., what percentage of the injected soft errors are detected?). We see that the average error detection rates are 40% and 75% for our approach and the full duplication based scheme, respectively. We see that while our approach performs very well in some benchmarks (e.g., *bubble-sort* and *adi*), it is not as successful in benchmarks such as *mxm*. This can be explained as follows. Our approach depends heavily on the availability of invariants to detect the occurrence of errors in the results generated by the application. Codes such as *mxm* do not contain very useful invariants. Therefore, the ability to check for errors is constrained, and consequently the detection rate is not as high as, say, that in bubble sort.

Having presented our error detection rates, we now turn our attention to the percentage increases in code sizes, which are illustrated in Fig. 6. We see that the average percentage increase due to our method and full duplication is 3% and 8%, respectively, showing that our approach incurs less overhead. The differences in code size overheads incurred by the different benchmarks are due to the nature of the invariants used to detect the errors. For example, *adi* has more error checking code embedded in it than bubble-sort; thus, the overhead in *adi* is much higher than that of *bubble-sort*.

The percentage increases in the original execution cycles are given in Fig. 7. As in the case of code sizes, we see that our approach is much better than the full duplication based scheme as far as this metric is concerned. It is to be noted that, the largest increases occur with *mxm* and *iter-merge*. This is because the relative size of the checker code added to the original application code is larger in the case of mxm and *iter-merge* when compared to, say, *bubble-sort*.

In our final set of experiments, we measure the sensitivity of our approach to the error injection rate. The default injection rate used thus far in our experiments was 1e-08. Fig. 8 gives the percentage error detection rates for our approach with different error injection rates. We see that the error detection rate remains generally stable with changes in the injection rates. This is mainly because the invariant method used is independent of the error injection rates.

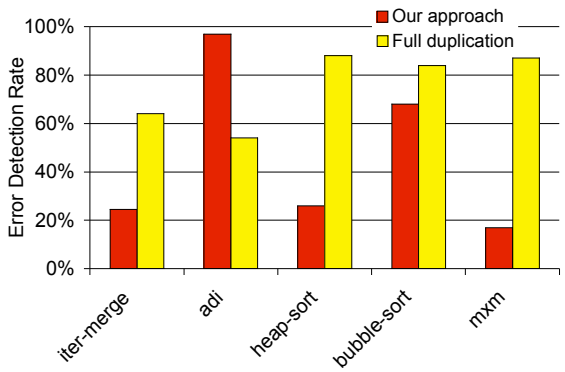

Figure 5. Error detection rates.

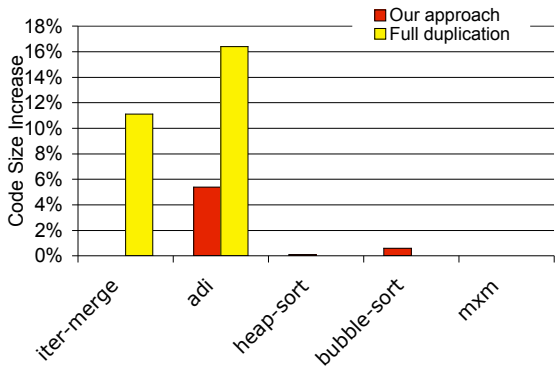

Fig. 6. Percentage code size increases.

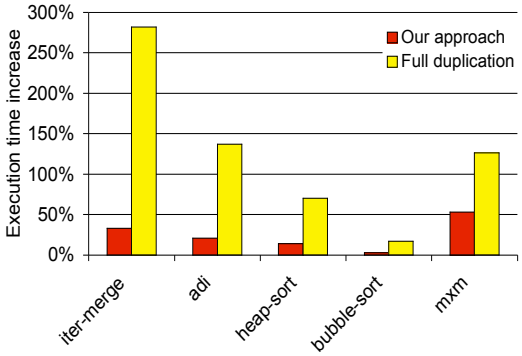

Fig. 7. Percentage increases in execution times.

VI. CONCLUSION

Soft errors are becoming increasingly important as process technology continues to scale down. Current methods to detect soft errors are based on either spatially redundant or temporally redundant computation, and are generally expensive from power, performance, and memory space perspectives. In this paper, we propose to use loop invariants to detect soft errors in data caches. We show, through an experimental analysis, that loop invariants can catch a large fraction of the errors that would be caught by an alternate scheme that adopts duplicate execution of instructions, and achieve this in a less expensive way as compared to the latter.

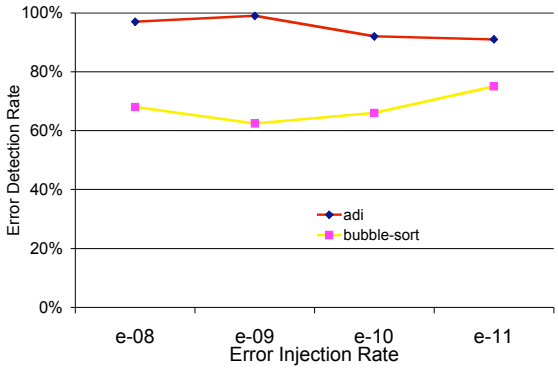

Fig. 8. Error detection rates with varying injection rates.

ACKNOWLEDGEMENTS

This work was supported in part by NSF CAREER Award #0093082.

REFERENCES

[1] M. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz, Transient-fault recovery for chip multiprocessors. In *Proceedings of the International Symposium on Computer Architecture,* June 2003.

[2] N. Oh, S. Mitra, and E. J. McCluskey, Error detection by duplicated instructions in Super-scalar processors. *IEEE Transactions on Reliability,* Vol. 51, Issue 1, Mar. 2002.

[3] B. Nicolescu, R. Velazco, Detecting soft errors by a purely software approach: method, tools and experimental results. In *Proceedings of DATE,* 2003.

[4] A.V.Aho, J.D. Ullman, Principles of Compiler Design, *Addison-Wesley Publishing Company Inc*, USA

[5] M. Rebaudengo, et al .Soft-error detection through software fault-tolerance techniques. In *Proceedings of the International Symposium on Defect and Fault Tolerance in VLSI Systems,* Nov. 1999, pp. 210-218.

[6] M. Zenha Rela, H. Madeira, J. G. Silva, Experimental Evaluation of the Fail-Silent Behavior in Programs with Consistency Checks, *Proc. FTCS-26,* 1996, pp. 394-403.

[7] K. H. Huang, J. A. Abraham, Algorithm-Based Fault Tolerance for Matrix Operations.In *IEEE Trans. Computers, vol. 33,* Dec 1984, pp. 518-528.

[8] S. Yau, F. Chen, An Approach to Concurrent Control Flow Checking, *IEEE Transactions on Software Engineering, Vol. SE-6, No. 2,* March 1980, pp. 126-137.

[9] MIT, The Daikon Invariant Detector User Manual, Apr. 2004.

[10] D. Burger, and T. M. Austin, The SimpleScalar Tool Set, Version 2.0. In *University of Wisconsin-Madison CS Department Technical Report #1342,* Jun. 1997.