Memory Space Conscious Loop Iteration Duplication for Reliable Execution

G. Chen[1], M. Kandemir[1], and M. Karakoy[2]

[1] CSE Department, Penn State University, University Park, PA, USA
{guilchen, kandemir}@cse.psu.edu
[2] 180 Queen's Gate, Imperial College, London, SW7 2AZ, UK
mk22@doc.ic.ac.uk

Abstract. Soft errors, a form of transient errors that cause bit flips in memory and other hardware components, are a growing concern for embedded systems as technology scales down. While hardware-based approaches to detect/correct soft errors are important, software-based techniques can be much more flexible. One simple software-based strategy would be full duplication of computations and data, and comparing the results of the corresponding original and duplicate computations. However, while the performance overhead of this strategy can be hidden during execution if there are idle hardware resources, the memory demand increase due to data duplication can be dramatic, particularly for array-based applications that process large amounts of data.

Focusing on array-based embedded computing, this paper presents a memory space conscious loop iteration duplication approach that can reduce memory requirements of full duplication (of array data), without decreasing the level of reliability the latter provides. Our "in-place duplication" approach reuses the memory locations from the same array to store the duplicates of the elements of a given array. Consequently, the memory overhead brought by the duplicates can be reduced. Further, we extend this approach to incorporate "global duplication", which reuses memory locations from other arrays to store duplicates of the elements of a given array. This paper also discusses how our approach operates under a memory size constraint. The experimental results from our implementation show that the proposed approach is successful in reducing memory requirements of the full duplication scheme for twelve array-based applications.

1 Introduction

Soft errors, a certain type of transient errors, generally result from random electric discharges caused by background radiation, including alpha particles, cosmic rays, and nearby human sources [18,25]. The impact of a soft error on a computer system is a *bit flip* in memory components and computational logic. With the scaling of technology down into the deep-submicron range, digital circuits are even more susceptible to random failure than previous generations. If not addressed properly, soft errors can lead to dramatic problems in embedded applications from a variety of domains. For example, in safety-critical applications, unpredictable reliability can result in significant cost in terms of human and equipment loss. Similarly, in commercial consumer applications where high-volume, low-margin production is the norm, high levels of product failures may necessitate the costly management of warranty support or expensive field maintenance, eventually affecting brand reputation.

Recent research has focused on the soft error problem from both architecture and software perspectives. We will discuss the related efforts in Section 6. One of the techniques that have been proposed is based on executing duplicates of instructions and comparing the results of the primary copy and duplicate to check correctness. It must be observed, however, that embedded environments typically operate under multiple constraints such as power consumption, memory size, performance and mean time to failure, and maintaining a required level of reliability against soft errors should be carefully balanced with other constraints. More specifically, one needs to consider the extra memory consumption, execution cycles, and power consumption due to duplicated instructions and data. In particular, limiting extra memory space demand of an application due to enhanced reliability is extremely important in many embedded environments. In embedded environments that execute a single application, the memory demand of the application directly determines the size of the memory to be employed, which means that an increase in memory demands can increase the overall cost of the embedded system and its area (form factor). Also, in multi-programmed embedded environments, increasing memory consumption of an application can reduce the number of applications that can execute simultaneously, thereby impacting overall performance of the system. Therefore, when increasing the number of instructions and size of data for reliability reasons, one must be careful in limiting the required extra memory space.

Motivated by this observation, this paper presents a memory space conscious loop iteration duplication scheme for reliability. The idea is to execute a copy (duplicate) of an original loop iteration (along with the original) and compare their results for correctness. In storing the results of the duplicates, we try to reuse some of the memory locations that originally store the data manipulated by the program. In other words, we recycle the memory locations as much as possible to reduce the extra memory demand due to duplicate executions. This is expected to bring two benefits. First, memory space consumption is reduced, which is very important for memory-constrained systems. Second, performance can be improved due to improved data cache behavior. Targeting array-intensive embedded applications, this paper makes the following contributions:

- We present a compiler-based approach to memory conscious loop iteration duplication. Our "in-place duplication" approach reuses memory locations from the same array to store the duplicates of its elements. Specifically, it reuses the locations of dead array elements to store the duplicates of the actively-used array elements. As a result, the memory overhead brought by duplicates is reduced.
- We discuss a "global duplication" scheme, which allows us reuse memory locations from other arrays to store the duplicates of the elements of a given array.
- We present experimental evidence demonstrating the effectiveness of the proposed approaches. Both in-place duplication and global duplication are automated within an optimizing compiler. We test our approaches using twelve array-based applications and show that in-place duplication can reduce the extra memory consumption of a duplication based scheme that does not consider memory space consumption by about 33.2%, and that the global duplication scheme brings up this figure to 42.1%.

- We demonstrate how our approach can be made to work when a limited extra memory consumption is permissible. In this scenario, our approach tries to reuse as many memory locations as possible under the specified memory constraint.

It must be emphasized that array-based codes are very important for embedded systems. This is because many embedded image and video processing programs/applications are array intensive [4], and they are usually in the form of loop nests operating on arrays of signal data.

There are several reasons why our approach is better than a hardware-based scheme, e.g., a combination of redundant instruction execution and ECC memory (i.e., memory protected by error correction code). First, if some applications (or some portions of an application) require greater reliability than others, software will be able to selectively apply duplication, instead of incurring the fixed ECC overhead on all of the memory accesses. Second, if an application needs a high level of reliability on existing hardware without ECC, a software technique would be needed. Third, our scheme can use whatever memory is available to increase reliability, i.e., we are able to decrease failure rate under a given memory space constraint.

The rest of this paper is structured as follows. In Section 1, we describe the representation used for loop iterations and array data. Section 3 discusses our assumptions, and presents our approach to in-place duplication. In the same section, we discuss our approach to duplication under a memory constraint as well. Section 4 discusses extensions to our base approach when some of our assumptions are relaxed. Section 5 gives our experimental results that show memory savings when using our approaches. In Section 6, we describe related work. Finally, in Section 7, we draw conclusions.

2 Representation for Loop Iterations, Data Space, and Array Accesses

Table 1 presents the notation used in this paper. The domain of our approach is the set of sequential array-intensive embedded programs consisting of nested loops. We assume that the loop bounds and the array indices (subscript functions) are affine functions of enclosing loop indices and loop-invariant constants. We handle other constructs such as non-affine array accesses and conditional statements conservatively.

Table 1. Notations

n	Number of enclosing loops for an array reference.
w	Number of arrays in a loop nest.
$\mathcal{I}$	Iteration space.
$\boldsymbol{I}$	$= [i_1 \quad i_2 \quad \cdots \quad i_n]^T$. An iteration point.
$\boldsymbol{I}^+$	$= \boldsymbol{I} + [0 \quad 0 \cdots 0 \quad 1]^T$.
$\preceq$	$\boldsymbol{I} \preceq \boldsymbol{J}$ means $\boldsymbol{I}$ is lexically less than or equal to $\boldsymbol{J}$.
X_k	An array.
M_k	Number of read references to X_k.
D_k	Number of dimensions of X_k.
$N_{k,i}$	Size of the ith dimension of X_k.
N_k	$= [N_{k,1} \quad N_{k,2} \quad \cdots \quad N_{k,D_k}]^T$. Size of X_k.
$\boldsymbol{x}$	Index of an array element.
$\mathcal{F}_{k,l}(\boldsymbol{I})$	lth reference to X_k. $\mathcal{F}_{k,l}(\boldsymbol{I}) = F_{k,l} \cdot \boldsymbol{I} + \boldsymbol{f_{k,l}}$.
R	Set of all read references in the loop body.
$\mathcal{G}_k(R)$	Right hand side of the kth statement.
$\boldsymbol{d_{k,l}}$	Dependence distance from $X_k(\mathcal{F}_{k,0}(\boldsymbol{I}))$ to $X_k(\mathcal{F}_{k,l}(\boldsymbol{I}))$; that is, $X_k(\mathcal{F}_{k,0}(\boldsymbol{I})) = X_k(\mathcal{F}_{k,l}(\boldsymbol{I} + \boldsymbol{d_{k,l}}))$
$\boldsymbol{d_{k,l_{max}}}$	$\max_{1 \le l \le M_k} \boldsymbol{d_{k,l}}$; that is, the maximum reuse distance (in terms of lexicographical order) from $X_k(\mathcal{F}_{k,0}(\boldsymbol{I}))$ to $X_k(\mathcal{F}_{k,l}(\boldsymbol{I}))$.
$\boldsymbol{L_k}$	Iteration offset for duplicates. The duplicate of $X_k(\mathcal{F}_{k,0}(\boldsymbol{I}))$ is stored in $X_k(\mathcal{F}_{k,0}(\boldsymbol{I} + \boldsymbol{L_k}))$.
$\boldsymbol{P_k}$	$= [p_{k,1} \quad p_{k,2} \quad \cdots \quad p_{k,D_k}]^T$. Space offset for duplicates. The duplicate of $X_k(\boldsymbol{x})$ is stored in $X_k(\boldsymbol{x} + \boldsymbol{P_k})$.
$\lvert p_{k,i} \rvert$	Absolute value of $p_{k,i}$.
INC_k	Array size expansion of X_k.

```
for i = 0,N-1
  for j = 1,N-1
    A(i,j) = 2*A(i,j-1) + 1;
```

Fig. 1. An example nested loop

In a given loop nest with n loops, iterators surrounding any statement can be represented as an n-entry vector $\boldsymbol{I} = [i_1 \quad i_2 \ \cdots \ i_n]^T$. The iteration space $\mathcal{I}$ consists of all the iterations of a loop nest. We use $\boldsymbol{I}^+$ as a shorthand for $\boldsymbol{I} + [0 \quad 0 \ \cdots \ 0 \quad 1]^T$. The index domain of an m-dimensional array X_k is a rectilinear polyhedron, in which each element can be represented as an m-entry vector $\boldsymbol{x_k} = [a_1 \quad a_2 \ \cdots \ a_m]^T$. We use $\mathcal{F}_{k,l}(\boldsymbol{I})$ to represent the access function of the lth reference to array X_k. $\mathcal{F}_{k,l}(\boldsymbol{I})$ can also be defined in a matrix/vector form as: $\mathcal{F}_{k,l}(\boldsymbol{I}) = F_{k,l} \cdot \boldsymbol{I} + \boldsymbol{f_{k,l}}$, where $F_{k,l}$ is an $m \times n$ matrix and $\boldsymbol{f_{k,l}}$ is an m-entry vector. As an example, for the two references shown in Fig. 1, we have:

$$\mathcal{F}_{1,0}(\boldsymbol{I}) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} \text{ and } \mathcal{F}_{1,1}(\boldsymbol{I}) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ -1 \end{bmatrix}.$$

A *data reuse* is said to exist from an array reference $\mathcal{F}_{k,l_1}$ to an array reference $\mathcal{F}_{k,l_2}$ if: $\exists \boldsymbol{I_1} \in \mathcal{I}, \boldsymbol{I_2} \in \mathcal{I} : \boldsymbol{I_1} \preceq \boldsymbol{I_2}$ and $\mathcal{F}_{k,l_1}(\boldsymbol{I_1}) = \mathcal{F}_{k,l_2}(\boldsymbol{I_2})$. In this case, $\boldsymbol{I_2} - \boldsymbol{I_1}$ is defined as the *reuse distance* between $\mathcal{F}_{k,l_1}$ and $\mathcal{F}_{k,l_2}$. For example, in Fig. 1, data reuse exists from $\mathcal{F}_{k,0}$ to $\mathcal{F}_{k,1}$ since $\mathcal{F}_{k,0}(\boldsymbol{I}) = \mathcal{F}_{k,1}(\boldsymbol{I^+})$, and the reuse distance between them is $[0 \quad 1]^T$.

3 Array Duplication

A simple approach to enhance reliability is to create a duplicated copy for each array, duplicate the execution of each iteration, and compare the result of the primary with that of the duplicate. We refer to this approach as *full duplication* in this paper. An important problem with this approach is that it doubles the memory space consumption (as each array is duplicated). Our objective is to improve the reliability of the computation in the loop, and keep the incurred memory cost at minimum. We achieve this by not duplicating the array fully, but *reusing* some memory locations (that are used to store other elements) for duplicates.

3.1 Assumptions

Our algorithm works on a per-loop basis. We assume that all the loops are normalized, i.e., the loop index variable of each loop nest increases by 1 at each step. Loop normalization [1] is a standard code modification technique that can be used to ensure this. In this section, we consider "in-place duplication", which means reusing array elements (i.e., their memory locations) for storing the duplicates of the elements of the same array. That is, for an array element, its duplicate can be stored only within the same array. In Section 4.2 we present the algorithm that allows "global duplication", i.e., reusing array locations for storing the duplicates of the elements from other arrays. Our algorithm operates on one array at a time. For an array X_k to be considered as a candidate by our algorithm, the following assumptions must be satisfied:

- **Assumption 1:** For every pair of array references to X_k, the reuse distance between them is a constant vector. Note that if two array references do not have any data reuse between them, they are also assumed to have a constant reuse distance vector. Most existing compiler optimizations for array-based codes operate under this assumption.
- **Assumption 2:** If an array element of X_k is written in the loop nest, all the reads to this array element retrieve the value stored by some write reference in the loop nest (that is, none of the reads to this element retrieves a value stored before the loop nest).
- **Assumption 3:** There is only one write reference to X_k in the loop body.

Whether an array satisfies Assumption 1 and Assumption 2 can be checked using data reuse analysis [23] and value dependence test [11]. Checking Assumption 3 is straightforward. In Section 4, we discuss the cases where we relax these assumptions. In in-place duplication, if an array does not satisfy all of the above assumptions, we fall back to the full duplication strategy for that array. For now, let us assume that all the arrays in the loop satisfy these assumptions. Based on the assumptions above, a loop body with w arrays can be represented as:

$$\begin{aligned} X_1(\mathcal{F}_{1,0}(\boldsymbol{I})) &= \mathcal{G}_1(R); \\ X_2(\mathcal{F}_{2,0}(\boldsymbol{I})) &= \mathcal{G}_2(R); \\ &\vdots \\ X_w(\mathcal{F}_{w,0}(\boldsymbol{I})) &= \mathcal{G}_w(R). \end{aligned}$$

R is the set of all read array references in the loop body, and $\mathcal{G}_i$ $(1 \le i \le w)$ represents a function of these read references. In mathematical terms:

$$\begin{aligned} R = \{ \; & X_1(\mathcal{F}_{1,1}(\boldsymbol{I})), X_1(\mathcal{F}_{1,2}(\boldsymbol{I})), \cdots, X_1(\mathcal{F}_{1,M_1}(\boldsymbol{I})), \\ & X_2(\mathcal{F}_{2,1}(\boldsymbol{I})), X_2(\mathcal{F}_{2,2}(\boldsymbol{I})), \cdots, X_2(\mathcal{F}_{2,M_2}(\boldsymbol{I})), \\ & \vdots \\ & X_w(\mathcal{F}_{w,1}(\boldsymbol{I})), X_w(\mathcal{F}_{w,2}(\boldsymbol{I})), \cdots, X_w(\mathcal{F}_{w,M_w}(\boldsymbol{I})) \; \}. \end{aligned}$$

$\mathcal{F}_{k,0}$ is the write reference to array X_k, and M_k is the number of read references to X_k.

Based on these assumptions, we can determine that there is a reuse from $\mathcal{F}_{k,0}$ to each read reference $\mathcal{F}_{k,l}$, and the corresponding reuse distance is a constant vector, which we denote using $\boldsymbol{d_{k,l}}$. This means that the array element $X_k(\mathcal{F}_{k,0}(\boldsymbol{I}))$, which is written at iteration $\boldsymbol{I}$, will be used at iterations $\boldsymbol{I} + \boldsymbol{d_{k,1}}$, $\boldsymbol{I} + \boldsymbol{d_{k,2}}$, …, $\boldsymbol{I} + \boldsymbol{d_{k,M_k}}$. This can be also expressed as:

$$\mathcal{F}_{k,l}(\boldsymbol{I} + \boldsymbol{d_{k,l}}) = \mathcal{F}_{k,0}(\boldsymbol{I}).$$

Let us assume that $\mathcal{F}_{k,l_{max}}$ is the one with the maximum reuse distance (in terms of lexicographical order) from $\mathcal{F}_{k,0}$; that is, $\boldsymbol{d_{k,l_{max}}} = \max_{1 \le l \le M_k}(\boldsymbol{d_{k,l}})$. Therefore, $X_k(\mathcal{F}_{k,0}(\boldsymbol{I}))$ written at iteration $\boldsymbol{I}$ is *last-used* at iteration $\boldsymbol{I} + \boldsymbol{d_{k,l_{max}}}$ by array reference $\mathcal{F}_{k,l_{max}}$.

3.2 In-place Duplication

Approach and Algorithm. For the execution of a statement to be reliable, we need to duplicate its input data, duplicate its execution, and compare the results of the original and duplicated executions. For example, the reliable version of a statement "`A(i)=`" "`A(i-1)+1;`" would be:

```
A(i) = A(i-1) + 1;
A'(i) = A'(i-1) + 1;
if A(i) != A'(i)
  error();
```

We assume that, `A'` is a duplicate for array `A` in the above statement. In this section, we discuss how we reduce the memory space overhead brought by duplicates without compromising reliability.

A memory location can be in two different states: *active* or *inactive*. At a given time, a memory location is "active" if the value stored in it will be used in the future. On the other hand, a memory location is "inactive" if there is not any future read operation on it, or its value is updated before any read operation on it takes place. As an example, Fig. 2 gives the states of three variables, a, b and c, at different points of time during execution. At any given time, we need to provide a duplicate for an active array element, so that any soft error that occurs in its location can be detected by comparing this array element and its duplicate. It is to be noted that, we can modify the value in an inactive location without affecting the correctness of the program. Therefore, the inactive memory locations are good candidates for storing the duplicates of the active memory locations. For example, in Fig. 2, we can use variable c to store the duplicate of a, because variable c is inactive from t_1 to t_2 and from t_4 to t_6, during which a is active, and needs to be duplicated if it is to be protected against soft errors. In this section, we focus on "in-place duplication", which means reusing inactive array elements (i.e., their locations) for storing the duplicates of the elements of the same array.

Let us consider an array X_k. Fig. 3 illustrates a scenario for selecting the location to store the duplicate for an element updated at loop iteration $\boldsymbol{I}$. At iteration $\boldsymbol{I}$, $X_k(\mathcal{F}_{k,0}(\boldsymbol{I}))$ is updated, and the same array element is last-used at iteration $\boldsymbol{J}$. Consequently, the array element $X_k(\mathcal{F}_{k,0}(\boldsymbol{I}))$ is active between iterations $\boldsymbol{I}$ and $\boldsymbol{J}$, and we need to keep a duplicate for it during this period. To save memory space, we want to find an element in X_k, which is inactive during this period. Recall that Assumption 2 presented in Section 3.1 says that all the read references to an array element are executed after the corresponding write reference (if such a write reference exists). Therefore, if an element is written in some loop iteration, it is inactive before that iteration. Consequently, if an array element is written after iteration $\boldsymbol{J}$, the last iteration at which

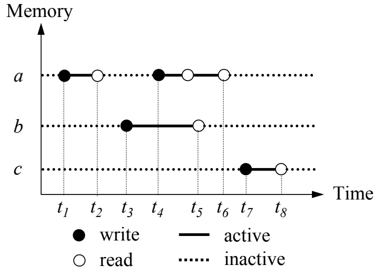

Fig. 2. States of memory locations during execution

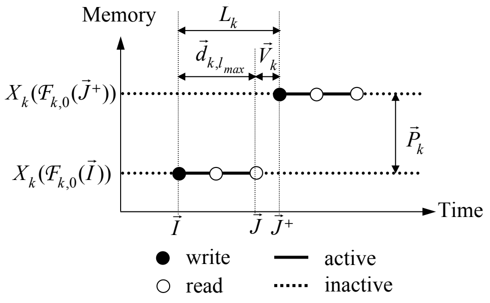

Fig. 3. Determining a memory location to store the duplicate for element $X_k(\mathcal{F}_{k,0}(\boldsymbol{I}))$

$X_k(\mathcal{F}_{k,0}(\boldsymbol{I}))$ is used, this element can be used to store the duplicate of $X_k(\mathcal{F}_{k,0}(\boldsymbol{I}))$, since it is inactive between $\boldsymbol{I}$ and $\boldsymbol{J}$. Our approach uses the array element written at iteration $\boldsymbol{J} + \boldsymbol{V_k}$, which is $\mathcal{F}_{k,0}(\boldsymbol{J} + \boldsymbol{V_k})$, to store the duplicate of the array element written at iteration $\boldsymbol{I}$. Here, $\boldsymbol{V_k}$ is a constant vector and $[0 \quad 0 \cdots 0 \quad 0]^T \prec \boldsymbol{V_k}$ so that iteration $\boldsymbol{J} + \boldsymbol{V_k}$ is executed after iteration $\boldsymbol{J}$. A possible choice for $\boldsymbol{V_k}$ will be discussed shortly.

Based on the discussion in Section 3.1, we know that $\boldsymbol{J} = \boldsymbol{I} + \boldsymbol{d_{k,l_{max}}}$. We use $\boldsymbol{L_k}$ to represent the distance between $\boldsymbol{I}$ and $\boldsymbol{J} + \boldsymbol{V_k}$. Hence, we have:

$$\boldsymbol{L_k} = \boldsymbol{J} + \boldsymbol{V_k} - \boldsymbol{I} = \boldsymbol{d_{k,l_{max}}} + \boldsymbol{V_k}.$$

Thus, the duplicate of $X_k(\mathcal{F}_{k,0}(\boldsymbol{I}))$ is stored in $X_k(\mathcal{F}_{k,0}(\boldsymbol{I} + \boldsymbol{L_k}))$. Consequently, the memory space distance between these two elements, denoted as $\boldsymbol{P_k}$, can be calculated as:

$$\begin{aligned}\boldsymbol{P_k} &= \mathcal{F}_{k,0}(\boldsymbol{I} + \boldsymbol{L_k}) - \mathcal{F}_{k,0}(\boldsymbol{I}) \\ &= (F_{k,0} \cdot (\boldsymbol{I} + \boldsymbol{L_k}) + f_{k,0}) - (F_{k,0} \cdot \boldsymbol{I} + f_{k,0}) \\ &= F_{k,0} \cdot (\boldsymbol{I} + \boldsymbol{L_k}) - F_{k,0} \cdot \boldsymbol{I} \\ &= F_{k,0} \cdot \boldsymbol{L_k}.\end{aligned}$$

Note that $\boldsymbol{P_k}$ is a constant vector since both $F_{k,0}$ and $\boldsymbol{L_k}$ are constant vectors. Consequently, for an arbitrary array element $X_k(\boldsymbol{x})$, its duplicate can reside in $X_k(\boldsymbol{x} + \boldsymbol{P_k})$, and this process can be carried out for every array used in the loop nest.

It should be observed that if $\boldsymbol{x}$ is near the array boundary, $\boldsymbol{x} + \boldsymbol{P_k}$ may exceed the original array boundary. In this case, we need to expand array X_k so that $\boldsymbol{x} + \boldsymbol{P_k}$ remains within the boundary. Assuming that $\boldsymbol{P_k} = [p_{k,1} \quad p_{k,2} \cdots p_{k,D_k}]^T$ and that the original size of the ith dimension of X_k is $N_{k,i}$, the ith dimension of X_k needs to be expanded by $|p_{k,i}|$, units (i.e., array elements) to $N_{k,i} + |p_{k,i}|$ (we use $|p_{k,i}|$ to denote the absolute value of $p_{k,i}$). Therefore, the total memory expansion for array X_k, denoted as INC_k, can be calculated as:

$$INC_k = \prod_{i=1}^{D_k}(N_{k,i} + |p_{k,i}|) - \prod_{i=1}^{D_k} N_{k,i} \tag{1}$$

Note that we expect INC_k to be much smaller than $\prod_{i=1}^{D_k} N_{k,i}$, the total size of the array. Let us now look at the problem of how to select a suitable V_k. Since our objective is to minimize the memory consumption due to duplication, we want to select a $\boldsymbol{V_k}$ so that INC_k can be minimized. Although an optimum $\boldsymbol{V_k}$ can be calculated by exhaustive enumeration or other sophisticated methods, we use a simple heuristic here that sets $\boldsymbol{V_k}$ to $[0\ 0 \cdots 0\ 1]^T$. The rationale behind this choice is that by minimizing V_k, we can minimize $\boldsymbol{P_k}$, and, thus, we can minimize INC_k. More specifically, in this case, we obtain:

$$\boldsymbol{L_k} = \boldsymbol{d_{k,l_{max}}} + [0 \quad 0 \cdots 0 \quad 1]^T = \boldsymbol{d_{k,l_{max}}}^+.$$

A potential problem is that $p_{k,i}$ could be negative for some i, which means that $x_i + p_{k,i}$ can be a negative number, where x_i is the array index of the original array reference for the ith dimension. Such a case can arise if the array is accessed from upper to lower index along the ith dimension. If this is the case, we use "$(x_i + p_{k,i} + N_{k,i}) \bmod N_{k,i}$" as the array index for this dimension. That is, we use the additional (upper) elements for placeholders of the duplicates of the lower elements.

Our algorithm for in-place duplication is given in Fig. 4. Assume that there are K arrays in the loop body, the average number of references to each array is M, the average

```
Algorithm I:

foreach array $X_k$ do
  check the following three assumptions:
    there is only one write reference to $X_k$;
    each read reference has a data reuse from the write
        reference;
    the reuse distances are constant vectors;
  if all assumptions are satisfied
    $\boldsymbol{d_{k,l_{max}}} = \max_{1 \le l \le M_k} \boldsymbol{d_{k,l}}$;
    $\boldsymbol{L_k} = \boldsymbol{d_{k,l_{max}}} + [0 \ \ 0 \cdots 0 \ \ 1]^T$
    $\boldsymbol{P_k} = F_{k,0} \cdot \boldsymbol{L_k}$;
    foreach dimension $i$ of array $X_k$ do
      $N_{k,i}$ += $|p_{k,i}|$;
    endfor
    foreach reference $X_k(\mathcal{F}_{k,l}(\boldsymbol{I}))$ do
      its duplicate is stored in
          $X_k(\mathcal{F}_{k,l}(\boldsymbol{I}) + \boldsymbol{P_k})$;
    endfor
  else use full duplication for $X_k$;
  endif
endfor
```

Fig. 4. Algorithm I: The algorithm for in-place duplication

```
Algorithm II:

foreach array $X_k$ do
  calculate $INC_k$;
endfor
sort the arrays as $X_{k_1}, X_{k_2}, \ldots, X_{k_w}$,
so that $INC_{k_1} \le INC_{k_2} \le \cdots \le INC_{k_w}$;
$h$ = 1;
$Mem$ = 0;
while $h \le w$ do
  if $(Mem + INC_{k_h}) \le U$ do
    $Mem$ += $INC_{k_h}$;
    $h$++;
  else goto LoopExit;
  endif
endwhile
LoopExit:
$h$ = $h$ - 1;
for $i$ = 1, $h$ do
  duplicate $X_{k_i}$;
endfor
```

Fig. 5. Algorithm II: The algorithm for selecting the arrays to duplicate under memory constraint (U)

```
int A(N);
for i=0,N-2
  A(i+1)=A(i)+a;
```

(a) Original program

```
int A(N),A'(N);
for i=0,N-2 {
  A(i+1)=A(i)+a;
  A'(i+1)=A'(i)+a;
  if A(i+1)!=A'(i+1)
    error();
}
```

(b) Full duplication

```
int A(N+2);
for i=0,N-2 {
  A(i+1)=A(i)+a;
  A(i+3)=A(i+2)+a;
  if A(i+1)!=A(i+3)
    error();
}
```

(c) In-place duplication

Fig. 6. Example application of in-place duplication

number of dimensions of each array is D, and the number of enclosing loops is n. Apart from checking our three assumptions, for each array X_k, the time to calculate $\boldsymbol{d_{k,l}}$ and $\boldsymbol{d_{k,l_{max}}}$ is $O(MD)$. It takes $O(nD)$ time to calculate $\boldsymbol{P_k}$. Therefore, the complexity of our algorithm, without taking into account the complexity of checking assumptions, is $O((M+n)DK)$. The time for checking our three assumptions is determined by the algorithm used for value dependence testing.

Example. We now discuss an example to illustrate our in-place duplication algorithm. Fig. 6 gives the example for our algorithm written in a pseudo-language syntax. In this figure, a and N are constants. In Fig. 6, we have:

$$F_{1,0} = [1]; \;\; \mathcal{F}_{1,0}(\boldsymbol{I}) = i+1; \;\; F_{1,1} = [1]; \;\; \mathcal{F}_{1,1}(\boldsymbol{I}) = i.$$

It is easy to determine that array A satisfies the three assumptions in Algorithm I, and we have $\boldsymbol{d_{1,1}} = [1]$. Therefore, we can obtain $\boldsymbol{d_{1,l_{max}}}$ as:

$$\boldsymbol{d_{1,l_{max}}} = \boldsymbol{d_{1,1}} = [1].$$

Based on this, we can calculate $\boldsymbol{L_1}$ and $\boldsymbol{P_1}$ as follows:

$$\boldsymbol{L_1} = \boldsymbol{d_{1,l_{max}}} + [1] = [2] \quad \text{and} \quad \boldsymbol{P_1} = F_{1,0} \cdot \boldsymbol{L_1} = [2].$$

Thus, we determine that we need to expand the original memory space allocated for array `A` by 2 elements. As a result, the duplicate of `A(i+1)` is in `A(i+1+2)`, which is `A(i+3)`, and the duplicate of `A(i)` is in `A(i+2)`. Using full duplication shown in Fig. 6(b), the total size of memory is increased by 100% over the original case with no duplication. In comparison, using our in-place duplication version in Fig. 6(c), the percentage memory increase over the original case is `2/N`, which is less than 2% when $\texttt{N} > 100$.

3.3 Duplication Under Memory Constraint

Approach and Algorithm. There exist cases where one may want to limit the memory consumption brought by duplication to a certain value. In this part, we discuss how our approach can be made to work under such a memory size constraint.

We assume that all the array elements are of equal importance (as far as improving reliability against soft errors is concerned), and our objective is to have duplicates for as many array elements as possible. Let us assume that we cannot reserve more than U units (array elements) of memory space to store duplicates. From Algorithm I and Equation (1), we can calculate the memory expansion for arrays that can make use of in-place duplication. On the other hand, for an array that needs to be fully duplicated, the incurred extra memory expansion is equal to its original size. In either case, we are able to determine INC_k for each array X_k. Next, we sort our arrays according to non-decreasing INC_k values, that is:

$$X_{k_1},\ X_{k_2},\ \ldots, X_{k_w},\ \text{ where } INC_{k_1} \leq INC_{k_2} \leq \cdots \leq INCk_w.$$

After that, we determine a maximum h such that $h \leq w$ and $\sum_{i=1}^{h} INC_{k_i} \leq U$. That is, we choose the candidate arrays for duplication in the increasing order of INC_k, until all the arrays are duplicated or duplicating more arrays would exceed the allowable memory size constraint. Here, h is the number of arrays that we choose during this process. Fig. 5 (on page 59) gives the algorithm (named Algorithm II) that selects the arrays to duplicate. After the selection is performed, we use the algorithm in Fig. 4 to duplicate the selected arrays.

Example. An example of duplication under memory constraint is shown in Fig. 7. By checking array `A` and array `B`, in Fig. 7(a) against our assumptions, we can determine that `A` can use in-place duplication and `B` needs to be fully duplicated. If there is no memory constraint, the original program could be transformed to the one given in

```
int A(100),B(100);
for i=0,98
  A(i+1)=A(i)+a;
  B(i+1)=B(i+1)+B(i);
```

(a) Original program

```
int A(102),B(100),B'(100);
for i=0,98
  A(i+1)=A(i)+a;
  A(i+3)=A(i+2)+a;
  if A(i+1)!=A(i+3)
    error();
  B(i+1)=B(i+1)+B(i);
  B'(i+1)=B'(i+1)+B'(i);
  if B(i+1)!=B'(i+1);
    error();
```

(b) Without memory constraint

```
int A(102),B(100);
for i=0,98
  A(i+1)=A(i)+a;
  A(i+3)=A(i+2)+a;
  if A(i+1)!=A(i+3)
    error();
  B(i+1)=B(i+1)+B(i);
```

(c) With an allowable increase of 10 array locations

Fig. 7. Example for duplication under memory constraint

Fig. 7(b). Now let us assume that we impose a memory constraint such that we cannot use more than 10 extra array locations for storing the duplicates. We use X_1 to represent array `A` and X_2 to represent array `B`. To determine the memory expansion due to array `A`, we proceed as follows:

$$F_{1,0} = [1];\ \mathcal{F}_{1,0}(\boldsymbol{I}) = i+1;\ F_{1,1} = [1];\ \mathcal{F}_{1,1}(\boldsymbol{I}) = i;\ \boldsymbol{d_{1,1}} = [1];\ \boldsymbol{d_{1,l_{max}}} = \boldsymbol{d_{1,1}} = [1];$$
$$\boldsymbol{L_1} = \boldsymbol{d_{1,l_{max}}} + [1] = [2];\ \ \boldsymbol{P_1} = F_{1,0} \cdot \boldsymbol{L_1} = [2];\ \ INC_1 = (100+2) - 100 = 2.$$

On the other hand, `B` needs to be fully duplicated. Thus, we have $INC_2 = 100$. Since $INC_1 < INC_2$, we first consider duplicating `A`, which is possible since $INC_1 < 10$. However, we cannot add `B` to the list of arrays to be duplicated since $INC_1 + INC_2 > 10$. To sum up, `A` is duplicated, whereas `B` is not duplicated. Fig. 7(c) gives the transformed code.

4 Extensions

Recall that, in Section 3.1, we listed three assumptions so that our in-place duplication could be used. In this section, we discuss the needed extensions to our base approach if some of these assumptions are to be relaxed. Note that Assumption 1 cannot be relaxed, since our approach would not work on an array that does not satisfy this assumption (i.e., if this assumption fails, we cannot put an upper bound on the extra memory space required). On the other hand, our approach can be extended to work on arrays that do not satisfy Assumption 2 or Assumption 3 (instead of just using full duplication for them).

4.1 Relaxing Assumption 3

Assumption 3 presented in Section 3.1 requires that there is only one write reference to the array being considered. Let us now consider the case where there are two write references, $\mathcal{F}_{k,0}$ and $\mathcal{F}_{k,1}$, for the array X_k being considered, and X_k satisfies Assumption 1 and Assumption 2. In this case, there are two possible scenarios for these two write references: either there is a data reuse between them, or there is no data reuse between them.

If there is a data reuse between these two write references, our algorithm can deal with this case with little modification. Without loss of generality, we assume that there is a data reuse from $\mathcal{F}_{k,0}$ to $\mathcal{F}_{k,1}$, and the reuse distance vector is $\boldsymbol{d_{k,1}}$. That is, $\mathcal{F}_{k,1}(\boldsymbol{I} + \boldsymbol{d_{k,1}}) = \mathcal{F}_{k,0}(\boldsymbol{I})$ $(\boldsymbol{d_{k,1}} \succeq [0\ \ 0\ \cdots\ 0]^T)$. This scenario is illustrated in Fig. 8. Comparing Fig. 2 and Fig. 8, we see that one can use the same strategy in determining the location to store the duplicate for $X_k(\mathcal{F}_{k,0}(\boldsymbol{I}))$. In fact, we can treat $\mathcal{F}_{k,1}$ the same way as we treat read references. This is because we are certain that $X_k(\mathcal{F}_{k,0}(\boldsymbol{J}^+))$ is not touched in the original loop until iteration $\boldsymbol{J}^+$ executes.

On the other hand, if there is no data reuse between these two write references, one can treat them as two different arrays. In this case, the references to X_k can be divided into two groups, based on to which write reference they have data reuse:

$$\mathcal{F}^1_{k,0},\ \mathcal{F}^1_{k,1},\ \ldots,\ \mathcal{F}^1_{k,M^1_k};$$
$$\mathcal{F}^2_{k,0},\ \mathcal{F}^2_{k,1},\ \ldots,\ \mathcal{F}^2_{k,M^2_k}.$$

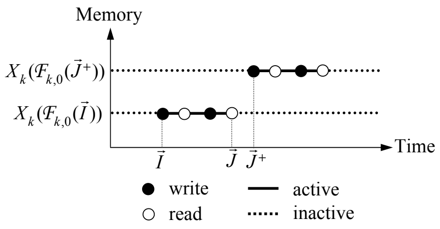
Memory
$X_k(\mathcal{F}_{k,0}(\vec{J}^+))$
$X_k(\mathcal{F}_{k,0}(\vec{I}))$
Time
$\vec{I}$
$\vec{J}$ $\vec{J}^+$
● write
○ read
active
inactive

Fig. 8. Determining the memory location to store the duplicate for $X_k(\mathcal{F}_{k,0}(\boldsymbol{I}))$ when there are two write references to X_k

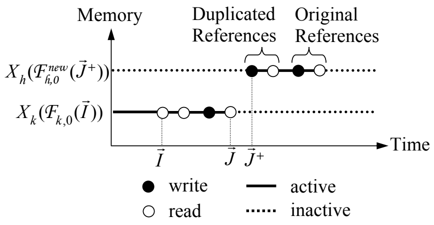

Fig. 9. Determining the memory location to store the duplicate for $X_k(\mathcal{F}_{k,0}(\boldsymbol{I}))$ in another array X_h

```
int A(N),B(N);
for i=0,N-2
  A(i+1)=A(i)+a;
  B(i+1)=A(i+1)+B(i);
  A(i)=B(i+1)/2;
```

(a) Original program

```
int A(N),A'(N),B(N),B'(N);
for i=0,N-2 {
  A(i+1)=A(i)+a;
  A'(i+1)=A'(i)+a;
  if A(i+1)!=A'(i+1)
    error();
  B(i+1)=A(i+1)+B(i);
  B'(i+1)=A'(i+1)+B'(i);
  if B(i+1)!=B'(i+1)
    error();
  A(i)=B(i+1)/2;
  A'(i)=B'(i+1)/2;
  if A(i)!=A'(i)
    error();
}
```

(b) Full duplication

```
int A(N+2),B(N+2);
for i=0,N-2 {
  A(i+1)=A(i)+a;
  A(i+3)=A(i+2)+a;
  if A(i+1)!=A(i+3)
    error();
  B(i+1)=A(i+1)+B(i);
  B(i+3)=A(i+3)+B(i+2);
  if B(i+1)!=B(i+3)
    error();
  A(i)=B(i+1)/2;
  A(i+2)=B(i+3)/2;
  if A(i)!=A(i+2)
    error();
```

(c) In-place duplication

Fig. 10. Example for multiple write references to the same array

Notice that there is a data reuse from $\mathcal{F}^i_{k,0}$ to $\mathcal{F}^i_{k,l}$ for $i = 1, 2$. The array elements accessed by these two groups do not overlap (since, otherwise, $\mathcal{F}^1_{k,0}$ and $\mathcal{F}^2_{k,0}$ would have data reuse); therefore, choosing an array element within one group as the location of a duplicate does not affect any access in the other group. This essentially means that we can treat the two groups as two different arrays, and select the locations for duplicates independently. The only modification to Algorithm I would be combining the array expansion results from these two groups together. For example, if our approach requires expanding the ith dimension of X_k by $|p^1_{k,i}|$ for the first group, and by $|p^2_{k,i}|$ for the second group, the final result is that the ith dimension is expanded by $max(|p^1_{k,i}|, |p^2_{k,i}|)$.

If there are more than two write references to array X_k in the loop, we can deal with them in a similar fashion. Specifically, we first divide the references into groups such that the references (in the same group) have data reuses between them, and the references in different groups are independent from each other. Then, we process each group separately as if it is a different array. Specifically, for each group X^i_k, we determine the write reference that have data reuse to all other references in its group and the reuse distances are non-negative. We represent such write reference as $\mathcal{F}^i_{k,0}$. This can also be expressed as $\forall 0 \leq l \leq M^i_k : \mathcal{F}^i_{k,l}(\boldsymbol{I} + \boldsymbol{d^i_{k,l}}) = \mathcal{F}^i_{k,0}(\boldsymbol{I})$ and $\boldsymbol{d_{k,l}} \succeq [0\ 0\ \cdots\ 0]^T$. After this, we can process this group using Algorithm I. Fig. 10 gives an example of how in-place duplication works when there are two write references in the same loop to the same array. There are two different arrays accessed in the code. Fig. 10(b) gives the full duplication version. Assume that X_1 represents `A` and X_2 represents `B`. Array `B` satisfies all the assumptions, and we can apply in-place duplication to it using Algorithm I. On the other hand, Array `A` satisfies Assumption 1 and Assumption 2, but does not satisfy Assumption 3 since there are two write references to it (`A(i+1)` and `A(i)`). Consequently, we need to use the strategy discussed above for in-place duplication for array `A`.

For the two write references to array `A`, namely, `A(i+1)` and `A(i)`, we can determine that `A(i)` has data reuse with `A(i+1)` based on data reuse analysis. Therefore, we represent `A(i+1)` as $X_1(\mathcal{F}_{1,0}(\boldsymbol{I}))$. Now, we can apply Algorithm I to `A`:

$$F_{1,0} = [1];\ \mathcal{F}_{1,0}(\boldsymbol{I}) = i+1;\ F_{1,1} = [1];\ \mathcal{F}_{1,1}(\boldsymbol{I}) = i;\ F_{1,2} = [1];\ \mathcal{F}_{1,2}(\boldsymbol{I}) = i+1;$$
$$F_{1,3} = [1];\ \mathcal{F}_{1,3}(\boldsymbol{I}) = i; \boldsymbol{d_{1,1}} = [1];\ \boldsymbol{d_{1,2}} = [0];\ \boldsymbol{d_{1,3}} = [1];$$
$$\boldsymbol{d_{1,l_{max}}} = max(\boldsymbol{d_{1,1}}, \boldsymbol{d_{1,2}}, \boldsymbol{d_{1,3}}) = \boldsymbol{d_{1,1}} = [1].$$

Based on this, we can calculate $\boldsymbol{L_1}$ and $\boldsymbol{P_1}$ as follows:

$$\boldsymbol{L_1} = \boldsymbol{d_{1,l_{max}}} + [1] = [2] \quad \text{and} \quad \boldsymbol{P_1} = F_{1,0} \cdot \boldsymbol{L_1} = [2].$$

Therefore, we find that the duplicate of `A(i+1)` is stored in `A(i+3)`, and the duplicate of `A(i)` is stored in `A(i+2)`. Fig. 10(c) gives the transformed code when both `A` and `B` are duplicated using in-place duplication.

4.2 Relaxing Assumption 2: Global Duplication

If an array X_k does not satisfy Assumption 2, this means that there exist some array elements that are used before they are written in the loop. Such locations need to be considered active from the beginning of the loop, and we cannot use them as duplicates for other array elements. Therefore, we are not able to use in-place duplication for such an array. However, as long as X_k satisfies Assumption 1, it is still possible to avoid full duplication using a different approach, which we discuss in this subsection. This approach reuses the locations in some other array (X_h) to store the duplicates for X_k, and is referred to as "global duplication".

For an array X_h to be used to store the duplicates for X_k, it needs to satisfy the following two conditions:

1. X_h should have the same number of dimensions as X_k.
2. X_h should satisfy all the three assumptions listed in Section 3.1.

Note that such an array X_h itself can benefit from in-place duplication, and in our approach, we always apply in-place duplication first. Therefore, when we try to use the locations in X_h to store duplicates of the elements of X_k, we need to take X_h's in-place duplication into account as well. Fig. 9 illustrates an example scenario. After X_h's in-place duplication, the references to X_h are doubled due to references to duplicates. We use $\mathcal{F}_{h,l}^{new}$ to denote both the original references and the references created by duplication. We have:

$$\begin{aligned} \mathcal{F}_{h,l}(\boldsymbol{I}) + \boldsymbol{P_h} &= \mathcal{F}_{h,l}^{new}(\boldsymbol{I}), && \text{for duplicated references;} \\ \mathcal{F}_{h,l}(\boldsymbol{I}) &= \mathcal{F}_{h,l+M_h}^{new}(\boldsymbol{I}), && \text{for original references;} \\ M_h^{new} &= 2M_h. \end{aligned}$$

```
int A(N),B(N);
for i=0,N-2
  A(i+1)=A(i)+a;
  B(i+1)=B(i+1)+B(i);
```

(a) Original program

```
int A(N+2),B(N),B'(N);
for i=0,N-2 {
  A(i+1)=A(i)+a;
  A(i+3)=A(i+2)+a;
  if A(i+1)!=A(i+3)
    error();
  B(i+1)=B(i+1)+B(i);
  B'(i+1)=B'(i+1)+B'(i);
  if B(i+1)!=B'(i+1)
    error();
}
```

(b) In-place duplication

```
int A(N+4),B(N);
for i=0,N-2 {
  A(i+1)=A(i)+a;
  A(i+3)=A(i+2)+a;
  if A(i+1)!=A(i+3)
    error();
  B(i+1)=B(i+1)+B(i);
  A(i+5)=A(i+5)+A(i+4);
  if B(i+1)!=A(i+5)
    error();
}
```

(c) Global duplication

Fig. 11. Example application of global duplication

For simplicity, we assume that all references to X_k have data reuses with each other. (if this assumption is not satisfied, we use the strategy discussed in Section 4.1 by dividing references into groups). In this case, we can find a reference, denoted as $\mathcal{F}_{k,0}$, from which all other X_k references have data reuses. We follow the approach described in Algorithm I to calculate $\boldsymbol{d_{k,l_{max}}}$ and $\boldsymbol{L_k}$. We know at this point that the array element $X_k(\mathcal{F}_{k,0}(\boldsymbol{I}))$ will not be used from iteration $\boldsymbol{I}+\boldsymbol{L_k}$ onwards. Therefore, we can use the X_h array element $X_h(\mathcal{F}^{new}_{h,0}(\boldsymbol{I}+\boldsymbol{L_k}))$, which is written at iteration $\boldsymbol{I}+\boldsymbol{L_k}$ for the first time in the loop, to store the duplicate for $X_k(\mathcal{F}_{k,0}(\boldsymbol{I}))$. To calculate the location of the duplicate for $X_k(\mathcal{F}_{k,i}(\boldsymbol{I}))$, we first represent it as $X_k(\mathcal{F}_{k,0}(\boldsymbol{I}-\boldsymbol{d_{k,i}}))$. Therefore, the duplicate of $X_k(\mathcal{F}_{k,i}(\boldsymbol{I}))$ is stored in $X_h(\mathcal{F}^{new}_{h,0}(\boldsymbol{I}-\boldsymbol{d_{k,i}}+\boldsymbol{L_k}))$.

In order to determine how much X_h needs to be expanded, we calculate $\boldsymbol{P_k}$, i.e., the difference between $\mathcal{F}^{new}_{h,0}(\boldsymbol{I}+\boldsymbol{L_k})$ and $\mathcal{F}^{new}_{h,0}(\boldsymbol{I})$:

$$P_k = \mathcal{F}^{new}_{h,0}(\boldsymbol{I}+\boldsymbol{L_k}) - \mathcal{F}^{new}_{h,0}(\boldsymbol{I}) = F^{new}_{h,0} \cdot \boldsymbol{L_k}.$$

Assuming that $\boldsymbol{P_k} = [p_{k,1} \quad p_{k,2} \ \cdots \ p_{k,D_k}]^T$ and the original size of the ith dimension of X_k is $N_{k,i}$, the ith dimension of X_k needs to be expanded by $|p_{k,i}|$ units to $N_{k,i} + |p_{k,i}|$.

Fig. 11 gives an example application of global duplication. In this example, we use in-place duplication for array A. Without the use of global duplication, array B needs to be fully duplicated as shown in Fig. 11(b). In the case of global duplication, array B uses the available space in array A to store its duplicates, and Fig. 11(c) gives the transformed code. If $\text{N} = 100$, by using in-place duplication, we can reduce the extra memory space from 100% (in the full duplication case) to 51%. By using global duplication, on the other hand, this number is further reduced to 2%.

5 Experimental Evaluation

5.1 Setup

In this section, we present an experimental evaluation of the approach discussed in this paper. To evaluate the effectiveness of our approach, we implemented it within an optimizing compiler [22] and performed experiments with several array based benchmarks. The average increase due to our approach in compilation times of the original

Table 2. Benchmarks used in this study

SpecFP2000			Embedded Applications		
Benchmark	**Brief Description**	**Input**	**Benchmark**	**Brief Description**	**Input**
171.swim	Shallow Water Modeling	Ref. Input	atr	Network Address Translation	1.47MB
172.grid	Multi-Grid Solver	Ref. Input	bss	Signal Deconvolution	3.07MB
177.mesa	3D Graphic Library	Ref. Input	encr	Digital Signature for Security	1.88MB
179.art	Image Recognition/Neural Networks	Ref. Input	img-seg6	Embedded Image Segregation	2.61MB
183.equake	Seismic Wave Propagation Simulation	Ref. Input	usonic	Feature-Based Area Estimation	4.36MB
188.ammp	Computational Chemistry	Ref. Input	wood04	Color-Based Surface Inspection	5.28MB

programs was about 220%. Table 2 lists the benchmarks used in this study. Our benchmarks are divided into two groups. The first group contains the C benchmarks from the SpecFP2000 suite [17] (plus two FORTRAN benchmarks, of which we were able to generate the C versions by hand), whereas the second group are representative applications from the domain of embedded computing. We collected the applications in the second group from different sources. For each group of benchmarks in Table 2, the second column gives a brief description of each benchmark and the last column shows the size of the total data manipulated by each benchmark.

5.2 Results

Fig. 12 shows the effectiveness of in-place and global duplication in reducing the memory requirements due to enhanced reliability. Each bar in this figure represents the extra memory demand of the corresponding approach (in-place or global), as a fraction of the extra memory demand of a scheme that duplicates all the array data in the application (i.e., full duplication). As can be seen from this bar-chart, our approaches save significant memory space with respect to the full duplication of all array data. The average savings brought by the in-place duplication scheme are 30.2% and 36.4% for the SpecFP2000 benchmarks and the embedded applications, respectively. The corresponding savings with the global duplication scheme are 39.3% and 44.2%. We see that, except for two benchmarks (177.mesa and atr), the global duplication scheme brings savings over the in-place duplication scheme, as the former has the flexibility of using other arrays for creating duplicates of the elements of a given array.

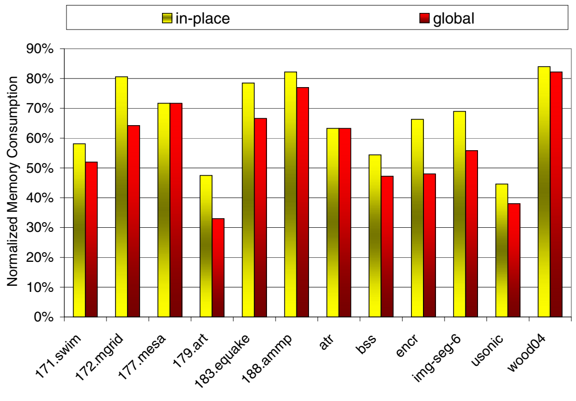

Fig. 12. Memory requirements of our duplication schemes

Note that, in the results presented above, all array elements have been duplicated (some recycling the memory locations of the elements that passed their last uses). Our next set of experiments measure the success of the in-place duplication approach that operates with memory constraints (see Section 3.3). The results are given in Fig. 13 with different memory constraints. Specifically, each point on the x-axis gives the maximum allowable increase in the size of the data manipulated by the original programs. The y-axis, on the other hand, gives the percentage of array elements duplicated by our approach. We see from these results that, our approach is successful in utilizing available extra memory space for duplication. In fact, even with an extra 5% memory space, it is able to duplicate about 16% of the array elements on the average. When we increase the extra available memory space reserved for duplicates to 40%, the average percentage of duplication becomes 76%.

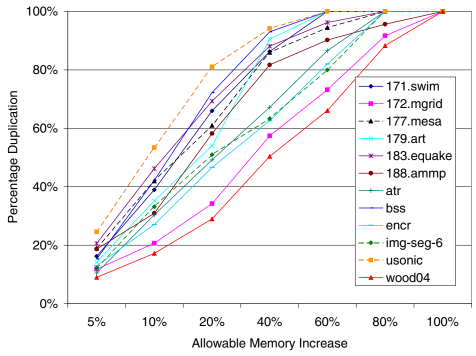

Fig. 13. Duplication under memory constraints

Although our focus in this paper is on memory space savings and reliability, it is also important to consider the impact of our approach on execution cycles. To determine the execution cycles taken by our approach, we simulated the benchmark codes using SimpleScalar [15]. The simulated architecture is a two-issue embedded processors with 16KB instruction and data caches. The access latencies for both the caches are 1 cycle, and a miss penalty of 100 cycles is assumed. The graph in Fig. 14 gives execution cycles for our two schemes as a fraction of the execution cycles taken by the full duplication strategy. Note that the full duplication scheme almost doubles the execution cycles of the original codes (i.e., those without any protection). Two observations can be made from this graph. First, both the schemes perform better than the full duplication based approach in terms of execution cycles. The main reason for this is that the reduction in data space requirements reduces capacity misses and this in turn reduces execution cycles. Second, the difference between our two schemes is less than one would expect, given the fact that global can reuse (and save) more memory space than in-place. The reason for the small difference between the two is the increased number of conflict misses with the global

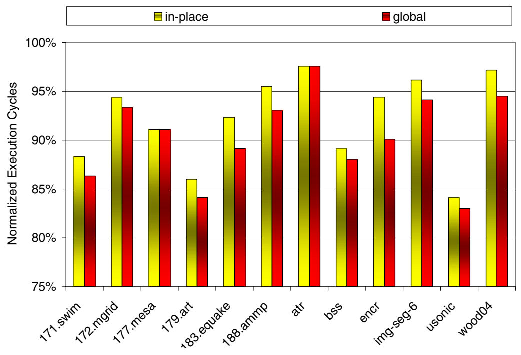

Fig. 14. Performance of our duplication schemes

scheme, due to the additional irregularity created by reusing different locations in the same loop iteration.

6 Related Work

6.1 Memory Reuse

There exist several prior studies that reduce the memory footprint of array-based programs. Wolfe [24] presented a technique called array contraction to optimize programs for a vector architecture. Lefebvre and Feautrier [8] proposed a method for reducing the memory overhead brought by full data expansion in automatic parallelization of loop-based static control programs. Song et al [16] proposed an algorithm that combines loop shifting, loop fusion, and array contraction to reduce memory usage and improve data locality. Wilde and Rajopadhye [21] studied memory reuse using a polyhedral model that performs static lifetime computations of variables. Strout et al [19] presented a schedule-independent storage mapping technique that reduces data space consumption but introduces no dependences other than those implied by flow dependences. Unnikrishnan et al [20] used a loop-based program transformation technique to reduce lifetimes of array elements. Their objective is to reduce the cases where the lifetimes of array elements overlap so that the storage requirement can be reduced. The common point between our approach and these prior studies is that all of them exploit variable lifetime information extracted by the compiler. The main difference is that we use this information for reducing the additional memory space demand due to enhanced reliability against soft errors, rather than reducing the original memory demand of the application. Also, most of these prior studies optimize for a single array at a time and operate under some additional constraints such as maintaining a certain degree of parallelism. In comparison, our global duplication scheme can reuse the space available in other arrays for storing the duplicates of the elements of a given array.

6.2 Software Approach to Transient/Permanent Errors

Software techniques for fault detection and recovery have been studied by prior research. Huang and Abraham [7] proposed Algorithm-Based Fault Tolerance (ABFT) to ensure the reliability of matrix operations. Roy-Chowdhury [13] extended the ABFT framework to a parallel processing environment. Oh and McCluskey [10] proposed Selective Procedure Call Duplication (SPCD) to improve system reliability. SPCD analyzes the procedure-call behavior of the program, and determines whether to duplicate the statements of a procedure or duplicate the procedure call. Rebaudengo et al [12] and Nicolescu et al [9] proposed systematic approaches for introducing redundancy into programs to detect errors in both data and code. Their approach demonstrated good error detection capabilities, but it also introduced considerable memory overheads due to full duplication for all variables. Our approach, in contrast, tries to minimize the memory overhead and retains the same degree of reliability that would be provided by full duplication. Audet et al [2] presented an approach for reducing a program's sensitivity to transient errors by modifying the program structure, without introducing redundancy into the program. Although this approach introduces almost no extra memory overhead, it cannot provide the same degree of reliability that would be provided by full duplication. Benso et al [3] presented a similar work that improves the reliability of a C code by code reordering. They do not consider memory optimization through array reuse. Shirvani et al [14] used software-implemented error detection and correction (EDAC) code to provide protection against transient errors. Several prior studies targeted at specific platforms. Gong et al [5,6] proposed a compiler-assisted approach to fault detection in regular loops for distributed-memory systems. Their approach focuses on performance issues, and does not consider memory consumption. In comparison, our objective in this work is to reduce memory overheads.

7 Concluding Remarks

Many embedded systems operate under multiple constraints such as limited memory size, limited battery power, real-time performance, reliability, and security. Consequently, in optimizing for one constraint, one should be very careful in controlling the impact of doing so on other constraints. Motivated by this observation, this paper presents a memory space conscious compiler-based approach that targets improving reliability of array-based programs against soft errors, a form of transient errors. The idea is to reuse the memory locations of inactive array elements (i.e., the elements that have reached their last uses) as placeholders for the duplicates of the actively used array elements. We present two specific algorithms based on this idea, and test their effectiveness using a set of twelve array-based applications. Our experimental evaluation demonstrates that our approach is successful in reducing the extra memory demand due to improved reliability.

References

1. R. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491-542, October 1987.
2. D. Audet, S. Masson, and Y. Savaria. Reducing fault sensitivity of microprocessor-based systems by modifying workload structure. In *Proc. IEEE International Symposium in Defect and Fault Tolerant in VLSI Systems*, 1998.
3. A. Benso, S. Chiusano, P. Prinetto, and L. Tagliaferri. A C/C++ source-to-source compiler for dependable applications. In *Proc. International Conference on Dependable Systems and Networks*, pp. 71-78, June 2000.
4. F. Catthoor, K. Danckaert, C. Kulkarni, E. Brockmeyer, P. G. Kjeldsberg, T. V. Achteren, and T. Omnes. *Data Access and Storage Management for Embedded Programmable Processors*. Kluwer Academic Publishers, 2002.
5. C. Gong, R. Melhem and R. Gupta. Compiler assisted fault detection for distributed memory systems. In *Proc. 1994 Scalable High Performance Computing Conference*, Knoxville, TN, 1994.
6. C. Gong, R. Melhem, and R. Gupta. Loop transformations for fault detection in regular loops on massively parallel systems. *IEEE Transaction on Parallel and Distributed Systems*, 7(12):1238-1249, December 1996.
7. K. H. Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers*, vol. C-33, pp. 518-528, June 1984.

8. V. Lefebvre and P. Feautrier. Automatic storage management for parallel programs. *Research Report PRiSM 97/8*, France, 1997.
9. B. Nicolescu and Raoul Velazco. Detecting soft errors by a purely software approach: method, tools and experimental results. In *Proc. Design, Automation and Test in Europe Conference and Exhibition*, Munich, Germany, March 2003.
10. N. Oh and E. J. McCluskey. Error detection by selective procedure call duplication for low energy consumption. *IEEE Transactions on Reliability*, 51(4):392-402, December 2002.
11. W. Pugh, D. Wonnacott. An exact method for analysis of value-based array data dependences. In *Proc. the 6th International Workshop on Languages and Compilers for Parallel Computing*, 1993.
12. M. Rebaudengo, M. Sonza Reorda, M. Violante, P. Cheynet, B. Nicolescu, and R. Velazco. System safety through automatic high-level code transformations: an experimental evaluation. In *Proc. IEEE Design Automation and Testing in Europe*, Munich, Germany, March 13-16, 2001.
13. Amber Roy-Chowdhury. Manual and compiler assisted methods for generating error detecting parallel programs. *Ph.D thesis*, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, 1996.
14. P. P. Shirvani, N. Saxena, and E. J. McCluskey. Software-implemented EDAC protection against SEUs. *IEEE Transaction on Reliability*, 49(3):273-284, September 2000.
15. http://www.simplescalar.com.
16. Y. Song, R. Xu, C. Wang, and Z. Li. Data locality enhancement by memory reduction. In *Proc. the 15th ACM International Conference on Supercomputing*, June 2001.
17. http://www.spec.org/osg/cpu2000/CFP2000/.
18. G. R. Srinivasan. Modeling the cosmic-ray-induced soft-error rate in integrated circuits: an overview. *IBM Journal of Research and Development*, 40(1):77–89, January 1996.
19. M. Strout, L. Carter, J. Ferrante, and B. Simon. Schedule-independent storage mapping in loops. In *Proc. ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.
20. P. Unnikrishnan, G. Chen, M. Kandemir, M. Karakoy, and I. Kolcu. Loop transformations for reducing data space requirements of resource-constrained applications. In *Proc. International Static Analysis Symposium*, June 11-13, 2003.
21. D. Wilde and S. Rajopadhye. Memory reuse analysis in the polyhedral model. *Parallel Processing Letters,* 1997.
22. R. Wilson et al. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Notices*, 29(12):31-37, December 1994.
23. M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In Proc. *ACM SIGPLAN Conference on Programming Language Design and Implementation,* pp. 30-44, June 1991.
24. M. J. Wolfe. *High Performance Compilers for Parallel Computing*, Addison-Wesley Publishing Company, 1996.
25. J. F. Zeigler. Terrestrial cosmic rays. *IBM Journal of Research and Development*, 40(1):19–39, January 1996.