Array Replication to Increase Parallelism in Applications Mapped to Configurable Architectures

Heidi E. Ziegler, Priyadarshini L. Malusare and Pedro C. Diniz

University of Southern California / Information Sciences Institute
4676 Admiralty Way, Suite 1001
Marina del Rey, California, 90292
{ziegler,priya,pedro}@isi.edu

Abstract. Configurable architectures, with multiple independent on-chip RAM modules, offer the unique opportunity to exploit inherent parallel memory accesses in a sequential program by not only tailoring the number and configuration of the modules in the resulting hardware design but also the accesses to them. In this paper we explore the possibility of array replication for loop computations that is beyond the reach of traditional privatization and parallelization analyses. We present a compiler analysis that identifies portions of array variables that can be temporarily replicated within the execution of a given loop iteration, enabling the concurrent execution of statements or even non-perfectly nested loops. For configurable architectures where array replication is essentially free in terms of execution time, this replication enables not only parallel execution but also reduces or even eliminates memory contention. We present preliminary experiments applying the proposed technique to hardware designs for commercially available FPGA devices.

1 Introduction

Emerging computing architectures now have multiple computing cores and multiple memory modules such as discrete and programmable register files as well as RAM blocks. For example, field-programmable gate arrays (FPGAs) allow designers to define an arbitrary set of registers and customize the topology of internal RAM blocks [12] to suit the data and computational needs of the computation. Other programmable architectures simply allow for the arrangement of registers and fine-grain functional units to create tailored pipelined architectures [5]. Overall these flexible architectures provide ample opportunities to exploit data parallelism as well as coarse and fine-grain parallelism.

Unfortunately, mapping sequential applications to these architectures is a difficult task. Programmers must explicitly manage the mapping and organization of arrays among the rich set of storage resources, configurable register sets and on and off-chip memories, if they are to fully exploit the architectural benefits of configurable devices. The wide range of design choices faced by the programmer makes it desirable to develop automated analysis and mapping tools that

can navigate certain characteristics of the design space, in particular, the data dependences found in common sequential imperative programs.

In this paper we focus on *array privatization* and *array replication* techniques to enable compilers to uncover parallelism opportunities in sequential computations that are traditionally impeded by both *anti* and *output-dependences*. We focus on array privatization not across loop iterations but within the same loop iteration. It focuses on the analysis of non-perfectly nested loops by determining *anti-dependences* between a sequence of nested loops in a *control loop*.

When two computations, that execute serially, access the same array location, reading its previous value and then writing a new value into the location, this gives rise to an *anti-dependence* between them. Similarly when two computation use the same location to store consecutive values that are otherwise independent creates an *output-dependence*. These dependences can be eliminated by creating a copy of the array, that each computation freely accesses. Each computation uses a distinct memory location to write and read a value, and in the absence of *true-dependences* between these loops nest, they can execute concurrently within the same iteration of the *control loop*.

This concurrent execution, however, raises the issue of memory contention when two or more concurrently executing loop nests access the same array region, *i.e.,* the loops exhibit *input-dependences*. To overcome this memory contention, we take advantage of the flexibility of memory mapping in configurable architectures by creating copies of shared array variables. By accessing the array copies, the parallel loop nests can therefore execute concurrently due to the absence of *anti-dependences* but also be contention-free. When the original computation exhibits *loop-carried true-dependences* (*i.e.,* values written in a given iteration that are read in a later iteration of a loop), the transformed code must update the array copies (not necessarily all of them) when the concurrent execution terminates to ensure that subsequent computations proceed with the correct values.

This transformation explores a space-time tradeoff. In order to eliminate *anti-, output-* and *input-dependences*, the implementation requires additional memory space. In addition, some execution time overhead is incurred in updating the copies to enforce the original program data dependences. The analysis abstractions, in cooperation with estimates of memory space usage, allow for an effective algorithm to manage this tradeoff and adjust, possibly dynamically, the performance of the implementation in response to available resources.

In this paper we evaluate the replication and privatization transformations when mapping a set of computations to a configurable computing device, a Xilinx Virtex™ FPGA. We simulate the transformed code as a concurrently executing hardware design, thereby revealing the effects on performance and the corresponding cost of storage.

This paper makes the following specific contributions:

- Describes the application of *array replication* and *array privatization* transformations to take advantage of the flexibility of configurable architectures.

- Extends existing array data-flow analysis to identify opportunities for concurrent execution of entire loops when arrays are replicated and temporarily privatized.
- Presents experimental results of our array replication algorithm when applied to a sample set of image processing computations for specific mappings to an FPGA device.

Preliminary results reveal that a modest increase of storage for private and replicated data leads to hardware designs that exhibit respectable execution time speedups, making this approach feasible when storage space is not a limiting factor in the design.

With the increase in VLSI device capacity and the emergence of computing architectures that have multiple computing units on the same die and a very rich set of configurable storage structures, the placement and layout of data will become increasingly important if applications are to fully exploit the true potential of internal data bandwidth and computational units.

This paper is structured as follows. Section 2 illustrates a motivating example for array replication. Section 3 describes the compiler analyses and a data replication algorithm. In section 4 we present preliminary experimental results of the application of the proposed analyses to a set of multimedia computations targeting an FPGA configurable device. We discuss related work in section 5 and then conclude in section 6.

2 Example

We now present an example showing how array replication (or copying) eliminates *anti-* and *output-dependences* thereby enabling concurrent execution of loops. This example also illustrates the elimination of *input-dependences* (*i.e.,* when two loops access arrays that are stored in the same memory module) that reduces memory contention introduced by concurrency. The computation is illustrated in figure 1 and consists of an outer `i` loop with three loop nests, `L1`, `L2` and `L3` nested within. Each of these three loop nests access a two-dimensional array variable `A` using affine subexpressions. The first two loop nests `L1` and `L2` read two consecutive rows of the array whereas the third loop nest `L3` writes the array row read by the first loop nest in the same iteration of `i` and in iteration `i+1` by the second loop nest.

Within loop `i`, one cannot execute loops `L1` and `L2` concurrently with loop `L3`, since there is an *anti-dependence* between `L3` and the other loops. Iterations of the `i` loop also cannot be executed concurrently given the *loop-carried true-dependence* between `L3` and `L2`. As such, privatization of `A` is therefore not possible either [11].

A way to enable concurrent execution of all loop nests during the execution of each iteration of the `i` loop is to create a copy of array `A` named `A_3`, which `L3` can update locally while loops `L1` and `L2` read from the original array `A`. We call this transformation where the array is being replicated with respect to the loop nest that writes it, a *partial replication*. At the end of concurrent

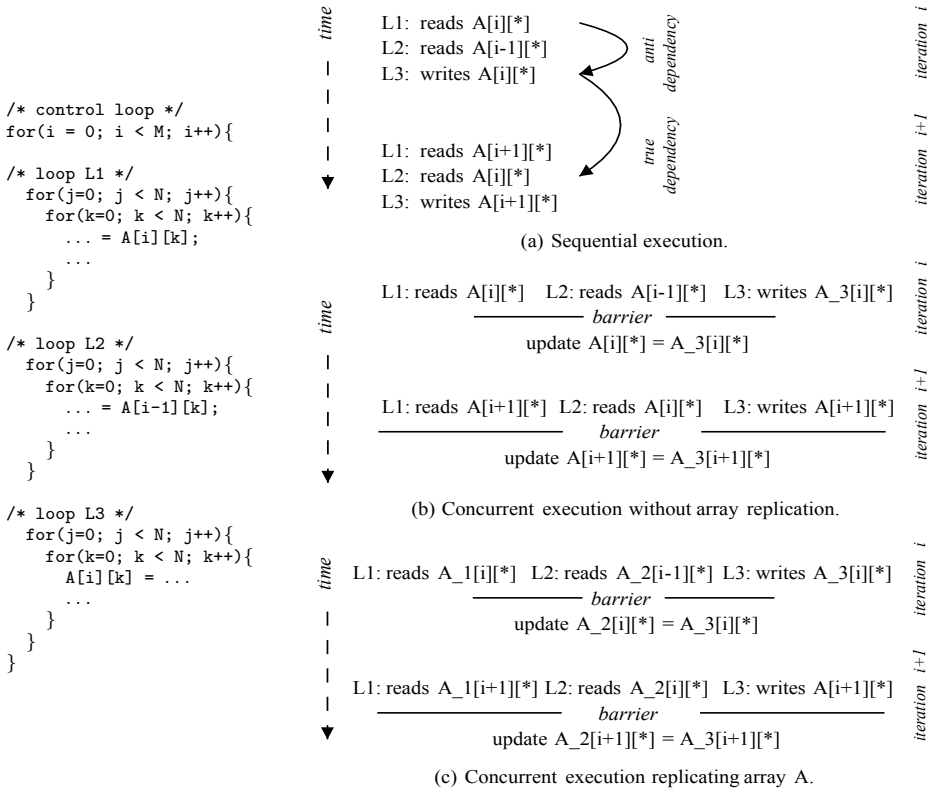

Fig. 1. Example computation and illustrative sequential and concurrent execution.

execution of loops `L1` through `L3` within one iteration of the `i` loop, we insert a synchronization *barrier* and then update the original array `A` with the new values generated by loop `L3`. This concurrent execution is illustrated in figure 1(b) and the corresponding parallel code is depicted in figure 2(a).

Due to the concurrent execution of the three loop nests, there is now memory contention on array `A` by the loops `L1` and `L2`. In an architecture with memory modules with a limited number of memory ports and in the absence of careful scheduling of read operations the execution of each loop will possibly stall for data. To alleviate the memory contention, we further replicate array `A` and assign these new arrays `A_1` and `A_2` to two memories that can be accessed in parallel by loops `L1` and `L2`. In this extended replication transformation, called *full replication*, we create copies that are local to the loops that both read and write the arrays.[1] We trade decreased execution time for increased array storage. In addition, the implementation must update the arrays to ensure data consis-

[1] There are additional degrees of replication with respect to the loops that read a given array. Furthermore, this need to replicate to reduce memory access contention interacts with other transformations such as custom data-layout enabled by loop unrolling as described in [9].

```
for(i = 0; i < M; i++){ /* control loop */

  begin par
  {
    for (j=0; j < N; j++){ /* loop L1 */
      for (k=0; k < N; k++){
        ... = A[i][k];
        ...;
      }
    }
  }

  {
    for (j=0; j < N; j++){ /* loop L2 */
      for (k=0; k < N; k++){
        ... = A[i-1][k];
        ...;
      }
    }
  }

  {
    for (j=0; j < N; j++){ /* loop L3 */
      for (k=0; k < N; k++){
        A_3[j][k] = ...;
        ...;
      }
    }
  }
  end par
  /* update original A */
  for (k=0; k < N; k++){
    A[i][k] = A_3[i][k];
  }
}
```

(a) Transformed code with partial replication

```
for(i = 0; i < M; i++){ /* control loop */

  begin par
  {
    for (j=0; j < N; j++){ /* loop L1 */
      for (k=0; k < N; k++){
        ... = A_1[i][k];
        ...;
      }
    }
  }

  {
    for (j=0; j < N; j++){ /* loop L2 */
      for (k=0; k < N; k++){
        ... = A_2[i-1][k];
        ...;
      }
    }
  }

  {
    for (j=0; j < N; j++){ /* loop L3 */
      for (k=0; k < N; k++){
        A_3[j][k] = ...;
        ...;
      }
    }
  }
  end par
  /* update A_2 */
  for (k=0; k < N; k++){
    A_2[i][k] = A_3[i][k];
  }
}
```

(b) Transformed code with full replication

Fig. 2. Transformed example computation

tency. While updating complete arrays is a safe and conservative approach, in actuality, only array elements that correspond to *loop-carried true-dependences* need to be updated. In our example and given that the array section written by `L3` is read only by `L2` in the next iteration of the `i` loop, the implementation only needs to update the array `A_2` associated with `L2` and not `A_1` associated with `L1`. In other words the definition of the array row written by `L3` reaches `L2` but not `L1`. Figure 2(b) depicts the transformed code after the replication of these arrays and the corresponding concurrent execution is illustrated in figure 1(c).

While the inclusion of a copy operation is likely to decrease performance benefits of such transformations in a classical architecture, in the context of configurable architectures, it has little if any impact on overall execution time. When the implementation of the computation in `L3` has to issue a write operation to a specific memory module with a configurable number of read and write ports, one can specify a multi-port write operation to occur synchronously to many memory modules without any performance penalty.

This example illustrates the kind of computation the array privatization and replication analysis described in this paper is designed to handle. First, we focus on non-perfectly nested loops with intra-iteration *anti-dependences* and *true-dependences* to recognize computations that can execute concurrently by the

introduction of one copy to the loop nest that modifies sections of an array. These values must then be copied back into the original array or other copies at the end of the execution of the parallel code region. Second, we introduce array copies to eliminate memory contention during the concurrent execution of multiple loop nests, thereby eliminating memory contention by exploiting the memory bandwidth available in architectures with configurable storage units.

3 Compiler Analysis

We now describe the compiler analysis and basic abstractions used to determine the opportunities for array replication with the goal of executing loop nests concurrently while reducing memory contention caused by accessing shared arrays. In this section we focus on imperfectly nested loops that manipulate array references. Whereas our analysis can be very precise for arrays that have affine array access functions, it can also handle, with loss of precision, references that are very irregular, *i.e.,* array-based indirect accesses.

3.1 Basic Abstractions and Auxiliary Functions

This analysis focuses on imperfectly nested loops where the outermost loops i_1 through i_k in the nest are perfectly nested. The i_k loop in the nest has a loop body that consists of a sequence of loop nests, each of which is a perfectly nested loop as well. We name the i_k as the *control loop* and build a control-flow-graph CFG corresponding to its body where each node corresponds to a loop nest. For the example in section 2, the CFG is a linear sequence of loop nests `L1` through `L3`, with loop `i` as the *control loop*. The corresponding CFG and dependences between the nodes are illustrated in figure 3.

For each loop nest, corresponding to a node n_k in the CFG, we define the *upwards-exposed read* and *write* regions for a given array `A` denoted by $\texttt{ER}(\texttt{A}, \texttt{n}_\texttt{k})$ and $\texttt{WR}(\texttt{A}, \texttt{n}_\texttt{k})$ respectively. The accessed array region is described by a set of linear inequalities. Given that each loop nest may be enclosed by a *control loop*, the corresponding dimension in the linear inequality will consist of symbolic information. A simple, yet effective implementation restriction is to limit the analysis to loops with single-induction variable affine subexpressions making the presence of index variables of the control loop simple. Figure 3 depicts the CFG of the control loop for the example in section 2, along with the relevant exposed-read and write region abstractions for the array `A`.

Using these abstractions, the compiler can compute data dependences between nodes of the CFG uncovering *anti-*, *input-*, *output-* and *true-dependences* by determining if the intersection between $\texttt{ER}(\texttt{A}, \texttt{n}_\texttt{i})$ and $\texttt{WR}(\texttt{A}, \texttt{n}_\texttt{j})$ between nodes n_i and n_j corresponding to the same array are non-empty. For instance, an *anti-dependence* exists between loops n_i and n_j due to array variables `A` iff $\{\texttt{WR}(\texttt{A}, \texttt{n}_\texttt{i}) \cap \texttt{ER}(\texttt{A}, \texttt{n}_\texttt{j}) \; with \; i > j\} \neq \emptyset$. In some cases the intersection will yield symbolic variables corresponding to the loops of the nest and the dependence test must conservatively assume dependence. In addition, we also define

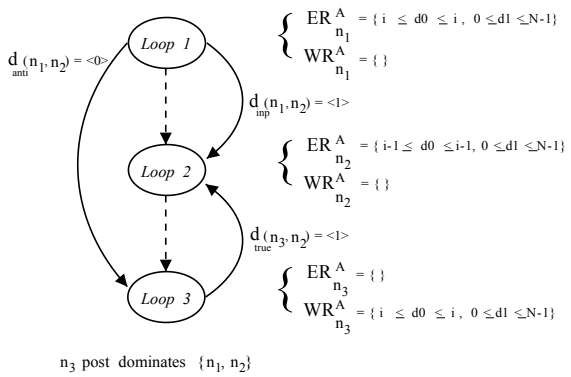

Fig. 3. Control flow graph and dependence information for the example code.

a dependence distance for each dependence type. For the example in section 2, there is a *loop-carried true-dependence* on the control loop `i` with a distance of 1 between the nodes corresponding to the loops `L2` and `L3` since `L3` writes the i^{th} row of the array `A` which is read by `L2` on the subsequent iteration of `i`.

3.2 Algorithm for Detecting Replication

Using the abstractions for data accesses, `ER` and `WR`, as well as the δ data dependence distance information, we now describe a compiler algorithm that determines opportunities for parallel execution of the loop nests that make up the body of the control loop. The algorithm also determines which arrays can be replicated to mitigate memory contention resulting from concurrent execution.

The algorithm, shown in figure 4, is structured into 5 main steps. In the first step the algorithm extracts the control loop `i` and the CFG corresponding to the enclosed loops. In the second step, for each node n_k, the algorithm computes `ER` and `WR` for each array variable `A`. In step three, the algorithm computes the dependence distances between every pair of nodes. In step four, the algorithm determines the opportunities for concurrent execution of the nodes within the same iteration of the `i` loop. The basic idea of this step is to identify a straight-line sequence of nodes such that the last node of the sequence exhibits an *anti-dependence* with the other nodes but there are no *true-* or *output-dependences* for that same iteration.[2] The set of nodes that meet this data dependence and control dependence criteria are gathered in a *parallel region* corresponding to the new node named `parallel(`n_k`)`. The compiler creates an array copy corresponding to this parallel node in order to eliminate *anti-dependences* and inserts synchronization code at the beginning and end of the parallel region so that values in the original array are updated with the value generated by n_k.

[2] Extending this simple algorithm to regions of the CFG with control-flow leads to several code generation complications.

```
Step 1. Extract control loops and coarse-grain control flow graph
        extract control loops i_0, ..., i_k and CFG;

Step 2. Determine exposed read and write information for each loop
        for all nodes n_i ∈ CFG
          for all arrays A n_i manipulates
            compute ER(A, n_i) and WR(A, n_i);

Step 3. Compute dependence types and distances
        for all pairs of nodes (n_i,n_j) ∈ CFG
          compute δ_type(n_i, n_j) <, =, > ⟨x⟩ where x is distance

Step 4. Identify parallel regions
        for all nodes n_k ∈ CFG s.t. WR(A, n_k) ≠ ∅ do
          if(numPreds(n_k) > 1) then
            parallel(n_k) = ∅;
            continue;
          R = {n_k};
          n = preds(n_k);
          while (n ≠ entry OR numSuccs(n) = 1) do
            if ((n_i ∉ R) AND (ER(A, n_i) ≠ ∅) AND (δ_true(n_i, n_k) = ⟨0⟩)) then
              R = R + {n_i}
            end if;
          end while;
          parallel(n_k) = R;
          insert fork before firstNode(parallel(n_k));
          insert join barrier after lastNode(parallel(n_k));
          end for;
        end for;

Step 5. Reduce contention by replicating arrays
        for all parallel regions of CFG do
          // Partial Replication case
          insert update array variable A for WR(A, n_k);
          if (FullReplication) then
            selectNumberCopies(parallel(n_k));
            for all n_j ∈ parallel(n_k) do
              update copy of A that has δ_true(n_j, n_k) > ⟨0⟩;
              for all arrays B replicate array for which δ_input(n_i, n_j) = ⟨0⟩;
            end for all;
          end if;
        end for;
```

Fig. 4. Parallelism detection and replication algorithm.

In step five, the algorithm identifies which array should be replicated for each parallel region. In this step the algorithm must decide how many copies to insert for each array variable and which copies need to be updated due to *true-dependences* across iterations of the control loop. In its simplest form, *partial replication*, there is a single copy for each parallel region that corresponds to a single node writing to an array variable. In the *full replication* variation, the algorithm generates one copy per each node that reads the array variable as well. Rather then updating all array copies, the algorithm only updates copies using the reaching definitions across loop iterations which is captured by loop-carried dependence information [13]. To this effect the algorithm determines which nodes, and for each array variable, exhibit a *loop-carried true dependence*, at the control-loop level. The particular value of the dependence distance of the control loop indicates the number of iterations across which the values need to

be updated in the original array location or copies. For the shortest distance of 1, the values must be updated at the end of the current iteration to be used in the subsequent iteration. However, if the distance is longer, one can delay the update and overlap it with the execution of another iteration thereby hiding its cost.

In this description we have statically determined which nodes of the CFG and therefore which loop nests operate on copies of the array using an external function, `selectCopies(parallel(`$\mathtt{n_k}$`))`. We foresee a more sophisticated algorithm, possibly dynamic, in which the need to replicate is selected at run-time depending on execution conditions.

3.3 Granularity of Replication

The algorithm described above can be augmented to allow the compiler to uncover opportunities for fine-grain replication by observing the order (in terms of array dimensions) in which multiple loop nests access the same array variables. In the example in figure 1 during parallel execution all loop nests access shared arrays in the same order, therefore array replication can occur at the finest granularity of an element.[3] Then concurrently executing loop nests only require 1 element of replicated data in the array copy. As soon as a loop nest has finished processing a given element, another element of the array can be copied. In addition the updates for copies can also proceed at a finer granularity as long as the iterations of the various concurrent loops execute synchronously. A similar analysis approach has been developed in the context of choosing the granularity of multiple communicating computations executing in a pipelined fashion [13].

In addition to requiring less storage space, at an increase in synchronization cost, this strategy also allows for the updates of copies to be executed concurrently with the parallel execution of the loop nests with the proper synchronization. This strategy reduces the execution time overhead of copy updating and substantially reduces the storage overhead.

The presence of irregular data access patterns, *i.e.,* non-affine does not pose a fundamental problem for the analysis outlined here. Rather then being able to determine exactly the array sections that need to be replicated in the case of a finer-grain synchronization, the analysis settles for replication at the next computational level at which the irregular data access pattern has been absorbed in a specific array dimension.

4 Experimental Results

We now describe the experimental methodology and results for the manual application of the analysis and program transformations to a set of kernels.

[3] The finest granularity may not be the best choice as additional execution time overhead might not be amortized over the small data size.

4.1 Methodology

We applied the analysis algorithm described in section 3 and evaluated the benefits and drawbacks using 3 synthetic kernels `hist`, `bic` and `lcd`.

The `hist` kernel is composed of 3 nested loops inside a single control loop with a total of 15 lines of C code. Each of the inner loop nests in `hist` manipulates 3 distinct array variables exhibiting anti-dependences among the last loop nest and the first two nests. There is a true dependence between the first and second loop nests preventing them from being executed concurrently even when anti-dependences are removed by replication. Nevertheless, the second and third loop nests can be executed concurrently.

The `bic` kernel is composed of 4 loop nests inside a single control loop with a total of 50 lines of C code. Each of the inner loop nests manipulates 4 array variables. This kernel exhibits intra-iteration anti-dependences among the four loop nests and an output dependence between the last two nests. Replicating a single array variable, however, will enable the concurrent execution of the first three loop nests.

The `lcd` kernel is composed of 3 loop nests inside a single control loop with a total of 20 lines of C code. Each of the inner loop nests manipulates 2 array variables. This kernel exhibits only intra-iteration anti-dependences among the last loop nest and the first two loop nests allowing the three loop nests to be executed concurrently via replication of a single array variable.

After we apply the analysis outline in section 3, we manually translate each of these kernels into behavioral VHDL and simulate the execution of the control loop using the Monet™ [7] behavioral synthesis tool. From this simulation, we obtain the execution time of each loop nest, in clock cycles at a given frequency, assuming each loop nest executes sequentially. Using the number of clock cycles obtained via the Monet simulation, we then use a simple discrete event simulator to determine the parallel execution time when one or more of the arrays have been replicated, thereby allowing for concurrent execution as well as reduced memory contention. This simulator allows us to determine the waiting time of each loop nest in the control loop as well as the overall percentage of time the execution spends stalled for memory operations. In our experiments we did not consider software pipelining execution techniques as they further increase the memory contention thereby skewing the replication results to be even more favorable to the application of the technique presented here. In these results we assume that every RAM is dual ported, with a one read and one write port that can be accessed in parallel and assigned the latency of every read and write operation to be 3 clock cycles.

4.2 Results

We now describe the results in terms of execution time reduction due to parallelism and the impact on memory space usage for each kernel. The *original* version is simply the kernel executing in a sequential fashion without any replication or parallel execution. The *partial replication* version corresponds to the

introduction of array copies for eliminating anti-dependences. In this version parallel loops may still access shared data. Finally, the *full replication* version includes copies of the array variables to decrease memory contention.

Table 1 summarizes the results in terms of execution time for each kernel and each analysis variation. For the *partial* and *full replication* versions, we have included the cost of performing the update operations after the parallel regions execute. The table indicates the amount of time each transformed kernel spends doing computation (comp. columns), updating the copies if any (update columns), stalling for memory (stall columns) and the overall percentage reduction (red. columns) of the total execution time taking into account the copy operations which execute sequentially after the parallel region executes.

Kernel	*Original Code*					*Partial Replication*					*Full Replication*				
	comp.	update	total	stall	red. %	comp.	update	total	stall	red. %	comp.	update	total	stall	red. %
`hist`	1.86	0	1.86	0	–	1.29	0.07	1.36	0	26.9	1.29	0.07	1.36	0	26.9
`bic`	131.1	0	131.1	0	–	77.8	4.11	81.9	36.9	37.5	65.55	4.11	69.66	0	46.8
`lcd`	61.44	0	61.44	0	–	49.15	4.10	53.25	24.58	13.3	24.57	4.10	28.68	0	53.3

Table 1. Execution time results (cycles in thousands).

As can be seen, there is a sharp decrease in the execution time in the *partial replication* code versions due to parallel execution ranging from 13% to 37%. This reduction simply reflects the concurrent execution of loop nests as revealed by comparing the values in the comp. columns for the *original* and *partial replication* versions. The results for the *partial replication* versions also reveal the opportunity to reduce execution time since the *stall time* values are substantial in the case of `bic` and `lcd`. For `hist` there is no stall time in the *partial replication* version given that only two loop nests execute concurrently and one of them updates a local copy. By aggressively replicating data in the *full replication* versions, the execution time is subsequently reduced leading to overall speedups between 1.37 and 2.1 over the original code version.

Table 2 depicts the space requirements for each code version. For each kernel and respective code version, we describe the number and size (in terms of number of array elements) the code uses along with the total space in bytes and percentage increase over the *original* code version.

Reflecting the opportunity for replication, the space requirements increase monotonically between the *partial* and *full replication* code versions. In the case of the `lcd` and `hist` kernels there is a substantial increase in memory usage close to 100%. While this increase may seem extreme, we note that these figures are biased by the fact that we do not take into account other kernel data structures. This effect is apparent in the `bic` kernel where due to the fact that this kernel

Kernel	*Original Code*			*Partial Replication*			*Full Replication*		
	Array Info	Total Size (KBytes)	Incr. (%)	Array Info	Total Size (KBytes)	Incr. (%)	Array Info	Total Size (KBytes)	Incr. (%)
`hist`	$1 \times (64\ by\ 64)$ $3 \times (64)$	17.15	—	$2 \times (64\ by\ 64)$ $3 \times (64)$	33.56	95.5	$2 \times (64\ by\ 64)$ $3 \times (64)$	33.56	95.5
`bic`	$6 \times (64\ by\ 64)$	98, 30	—	$7 \times (64\ by\ 64)$	114.7	16.7	$10 \times (64\ by\ 64)$	163.8	66.7
`lcd`	$2 \times (64\ by\ 64)$	32.77	—	$3 \times (64\ by\ 64)$	49.15	50.0	$4 \times (64\ by\ 64)$	65.54	100.0

Table 2. Space requirements results.

manipulates a larger number of arrays that are not replicated, the percentage increase of space requirements is much smaller.

4.3 Discussion

These preliminary results indicate that the execution overhead of updating array copies can be negligible, allowing full exploitation of the concurrent execution performance benefits. The results also reveal that memory contention, even with a small number of concurrent tasks can be substantial. In this scenario, the fully replicated variation allows for the elimination of memory contention, and further improve execution performance. Overall fully replicated code versions achieve speedups between 1.4 and 2.1 with a maximum increase in memory usage by a factor of 2.

Although there are other execution techniques, such as pipelining, these results reveal that using replication techniques a compiler can eliminate anti-dependences enabling substantial increases in execution speed at modest increases in memory space requirements. This experience reveals that replication can be a valuable technique for parallel performance when memory space is not at a premium.

5 Related Work

In this section we discuss related work in the areas of array data-flow analysis, privatization, storage reuse and replication.

Array Privatization/Renaming and Data-flow Analysis Array privatization determines that a variable assigned within the loop is used only in the same iteration in which it is assigned [4, 6]. Renaming is designed to allow for concurrent operations that have output and anti-dependences but where there is no flow of values between statements of a loop nest. It has been used mainly for scalar variables as for arrays the additional memory costs make it very unprofitable for traditional high-end architectures. Array data-flow analysis [3, 10] focuses on data dependence analysis that is used to determine the privatization requirements as well as the conditions for parallelization.

Replication for Shared Memory Multiprocessor Systems Many compilers targeting shared memory systems replicate data to enable concurrent read accesses [1] and further [8] investigates adaptive replication in order to reduce synchronization overheads that may ultimately degrade performance.
Memory Parallelism There have been many approaches to improve memory parallelism. In particular, for FPGAs, [9] introduces a novel data and code transformation called *custom data layout*. After applying scalar replacement to reduce the number of memory accesses, this transformation is applied to partition the remaining array accesses across available memories.

The approach described in this paper differs from these efforts in many respects. First, and unlike traditional privatization analyses, we relax the conditions for privatization allowing anti-dependences both within the same iteration as well as across iterations of the control loop. Array renaming is the technique used in our first transformation to expose concurrency across multiple loop nests[2]. We augment this transformation with replication (or copying) to increase the memory bandwidth and hence eliminate contention. Despite the similarities our combined renaming and replication transformations allow for values to flow across iterations of the control loop whereas simple renaming has been used within the same loop nest. Second, data layout techniques typically work in combination with loop-based transformations such as loop unrolling to expose more parallel accesses when the unrolled body reveals references with data access patterns that are disjoint in space. The transformations described here are clearly orthogonal to these two approaches. Lastly, the approach described here is geared towards non-perfectly nested loops where an outermost control loop or loops need to be executed sequentially due to true loop-carried dependences but each loop nested within can execute concurrently.

The approach described here takes advantage of the fact that configurable architectures can mitigate several sources of replication overhead typically not possible in traditional computing architectures. First, the number and connectivity of memory units can be tailored to the exact number of array copies. Second, the spatial nature of the execution in configurable architecture allows the execution of the copy/update operations without substantially instruction overhead. Furthermore it is possible to perform a single write operation to multiple memories simultaneously thereby updating more than one array copy.

6 Conclusion

Configurable architectures offer the potential for customized storage structures. This flexibility enables the application of low overhead data replication and privatization techniques to mitigate or even eliminate memory contention issues in concurrent loop execution where shared data are accessed. In this paper we have presented a simple array data-flow analysis algorithm to uncover the opportunities for array replication and temporary privatization in computations expressed as non-perfectly nested loops. The experimental results, for a set of kernels targeted to commercially available FPGA devices, reveal that a modest increase in

storage for private and replicated data leads to hardware designs that exhibit small speedups. These results make this approach feasible when chip capacity for data storage is available.

References

1. F. Allen, M. Burke, R. Cytron, J. Ferrante, W. Hsieh, and V. Sarkar. A Framework for Determining Useful Parallelism. In *Proc. Intl. Conf. Supercomputing*, ACM, pages 207–215, 1988.
2. R. Allen and K. Kennedy. Automatic Translation of Fortran Programs to Vector Form. 9(4):491–542, 1987.
3. V. Balasundaram and K. Kennedy. A technique for summarizing data access and its use in parallelism enhancing transformations. In *Proc. ACM Conf. Programming Languages Design and Implementation*, pages 41–53, 1989.
4. R. Eigenmann, J. Hoeflinger, Z. Li, and D. Padua. Experience in the AutomaticParallelization of four Perfect Benchmark Programs. In *Proc. 4th Workshop Languages and Compilers for Parallel Computing*, LNCS. Springer-Verlag, 1991.
5. S. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. Taylor, and R. Laufer. PipeRench: a coprocessor for streaming multimedia acceleration. In *Proc. 26th Intl. Symp. Comp. Arch.*, pages 28–39, 1999.
6. Z. Li. Array privatization for parallel execution of loops. In *Proc. ACM Intl. Conf. Supercomputing*, 1992.
7. Mentor Graphics Inc. *Monet*^{TM}, 1999.
8. M. Rinard and P. Diniz. Eliminating Synchronization Bottlenecks in object-based Programs using Adaptive Replication. In *Proc. Intl. Conf. Supercomputing*, ACM, pages 83–92, 1999.
9. B. So, M. Hall, and H. Ziegler. Custom Data Layout for Memory Parallelism. In *Proc. Intl. Symp. Code Gen. Opt.*, pages 291–302, March 2004.
10. C.-W. Tseng. Compiler optimizations for eliminating barrier synchronization. In *Proc. Fifth Symp. Principles and Practice of Parallel Programming*, volume 30(8) of *ACM SIGPLAN Notices*, pages 144–155, 1995.
11. P. Tu and D. Padua. Automatic Array Privatization. In *Proc. 6th Workshop Languages and Compilers for Parallel Computing*, LNCS. Springer-Verlag, 1993.
12. Xilinx Inc. *Virtex-II Pro*^{TM} *Platform FPGAs: introduction and overview*, DS083-1(v2.4.1) edition, March 2003.
13. H. Ziegler, M. Hall, and P. Diniz. Compiler-generated Communication for Pipelined FPGA applications. In *Proc. 40th Design Automation Conference*, June 2003.