Search Space Properties for Mapping Pipelined FPGA Applications*

Heidi Ziegler, Mary Hall, and Byoungro So

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

Abstract. This paper describes an automated approach to hardware design space exploration, through a collaboration between parallelizing compiler technology and high-level synthesis tools. In previous work, we described a compiler algorithm that optimizes individual loop nests, expressed in C, to derive an efficient FPGA implementation. In this paper, we describe a global optimization strategy that maps multiple loop nests to a pipelined FPGA implementation. The global optimization algorithm automatically transforms the computation to incorporate explicit communication and data reorganization between pipeline stages, and uses metrics to guide design space exploration to consider the impact of communication and to achieve balance between producer and consumer data rates across pipeline stages. In a case study with a machine vision application, we find that on-chip communication greatly reduces the number of memory accesses, and thus results in designs that are less memory bound.

1 Introduction

The extreme flexibility of Field Programmable Gate Arrays (FPGAs), coupled with the widespread acceptance of Hardware Description Languages (HDLs) such as VHDL or Verilog, has made FPGAs the medium of choice for fast hardware prototyping and a popular vehicle for the realization of custom computing machines that target multi-media applications. Unfortunately, developing programs that execute on FPGAs is extremely cumbersome, demanding that software developers also assume the role of hardware designers.

In this paper, we describe a new strategy for automatically mapping from high-level algorithm specifications, written in C, to efficient coarse-grain pipelined FPGA designs. In previous work, we presented an overview of DEFACTO, the system upon which this work is based, which combines parallelizing compiler technology in the Stanford SUIF compiler with hardware synthesis tools [12]. In [20] we presented an algorithm for mapping a single loop nest to an FPGA and a case study [27] describing the communication and partitioning analysis necessary for mapping a multi-loop program to multiple FPGAs. In this paper, we combine the optimizations applied to individual loop nests with analyses and

* This work is funded by the National Science Foundation (NSF) under Grant CCR-0209228 and the Defense Advanced Research Project Agency under contract number F30603-98-2-0113.

optimizations necessary to derive a globally optimized mapping for multiple loop nests. This paper focuses on the mapping to a single FPGA, incorporating more formally ideas from [27] such as the use of matching producer and consumer rates to prune the search space.

As the logic, communication and storage are all configurable, there are many degrees of freedom in selecting the most appropriate implementation of a computation, constrained by chip area. Further, due to the complexity of the hardware synthesis process, the performance and area of a particular design cannot be modelled accurately in a compiler. For this reason, the optimization algorithm involves an iterative cycle where the compiler generates a high-level specification, synthesis tools produce a partially synthesized result, and estimates from this result are used to either select the current design or guide generation of an alternative design. This process, which is commonly referred to as *design space exploration*, evaluates what is potentially an exponentially large search space of design alternatives. As in [20], the focus of this paper is a characterization of the properties of the search space such that exploration considers only a small fraction of the overall design space.

To develop an efficient design space exploration algorithm for a pipelined application, this paper makes several contributions:

- Describes the integration of previously published communication and pipelining analyses [26] and the single loop nest design space exploration algorithm [20].
- Defines and illustrates important properties of the design space for the global optimization problem of deriving a pipelined mapping for multiple loop nests.
- Exploits these properties to derive an efficient global optimization algorithm for coarse-grained pipelined FPGA designs.
- Presents the results of a case study of a machine vision kernel that demonstrate the impact of on-chip communication on improving the performance of FPGA designs.

The remainder of the paper is organized as follows. In the next section we present some background on FPGAs and behavioral synthesis. In section 3, we provide an overview of the previously published communication analysis. In section 4, we describe the optimization goals of our design space exploration. In section 5 we discuss code transformations applied by our algorithm. We present the search space properties and a design space exploration algorithm in section 6. We map a sample application, a machine vision kernel in section 7. Related work is surveyed in section 8 and we conclude in section 9.

2 Background

We now describe FPGA features of which we take advantage and we also compare hardware synthesis with optimizations performed in parallelizing compilers. Then we outline our target application domain.

2.1 Field Programmable Gate Arrays and Behavioral Synthesis

FPGAs are a popular vehicle for rapid prototyping. Conceptually, FPGAs are sets of reprogrammable logic gates. Practically, for example, the Xilinx Virtex

family of devices consists of 12,288 device *slices*; each slice in turn is composed of 2 look-up tables (LUTs), able to implement an arbitrary logic function of 11 boolean inputs and 6 outputs [25]. Two slices form a configurable logic block. These blocks are interconnected in a 2-dimensional mesh. As with traditional architectures, bandwidth to external memory is a key performance bottleneck in FPGAs, since it is possible to compute orders of magnitude more data in a cycle than can be fetched from or stored to memory. However, unlike traditional architectures, FPGAs allow the flexibility to devote its internal configurable resources either to storage or to computation.

```
#define IMAGE 16
int u[IMAGE][IMAGE]; v[IMAGE][IMAGE];
int peak[IMAGE][IMAGE];
int feature_x[IMAGE][IMAGE];
int feature_y[IMAGE][IMAGE];
int th, uh1, uh2;

/* stage s1. Apply Prewitt Edge Detector */
for(x = 0; x < IMAGE-3; x++){
  for(y = 0; y < IMAGE-3; y++){
1.   uh1= -3*u[x][y] - ···;
2.   uh2= 3*u[x][y] + ···;
3.   peak[x][y] = (uh1 + uh2);
  }
}

/* stage s2. Find Features - threshold */
for(x = 0; x < IMAGE-3; x++){
  for(y = 0; y < IMAGE-3; y++){
4.   if(peak[x][y] > th){
5.     feature_x[x][y] = x;
6.     feature_y[x][y] = y;
     } else {
7.     feature_x[x][y] = 0;
8.     feature_y[x][y] = 0;
     }
  }

for(x = 0; x < IMAGE-2; x++){
  for(y = 0; y < IMAGE-2; y++){
9.   if(feature_x[x][y] != 0)
10.    ssd[x][y] =
         (u[x][y]-v[x][y+1])*(u[x][y]-v[x][y+1])
         ···;
  }

}
}
```

```
for(x = 0; x < IMAGE-3; x+=2){
  for(y = 0; y < IMAGE-3; y+=2){
    if (th < peak[x][y]) {
      feature_x_1_0_0 = x;
      feature_y_1_1_0 = y;
    } else {
      feature_x_1_0_0 = 0;
      feature_y_1_1_0 = 0;
    }
    feature_y[x][y] = feature_y_1_1_0;
    feature_x[x][y] = feature_x_1_0_0;
  }
}

for(x = 0; x < (IMAGE-3)/2; x+=2){
  for(y = 0; y < (IMAGE-3)/2; y+=2){
1.   uh1= -3*u[x][y] - ···;
2.   uh2= 3*u[x][y] + ···;
3.   peak[x][y] = (uh1 + uh2);

4.   uh1= -3*u[x+1][y] - ···;
5.   uh2= 3*u[x+1][y] + ···;
6.   peak[x+1][y] = (uh1 + uh2);

7.   uh1= -3*u[x][y+1] - ···;
8.   uh2= 3*u[x][y+1] + ···;
9.   peak[x][y+1] = (uh1 + uh2);

10.  uh1= -3*u[x+1][y+1] - ···;
11.  uh2= 3*u[x+1][y+1] + ···;
12.  peak[x+1][y+1] = (uh1 + uh2);
  }
}
```

Fig. 1. MVIS Kernel with Scalar Replacement and Unroll and Jam for S2.

To configure an FPGA, designers download a bitstream file with information to configure a set of slices in the FPGA as well as the routing. Using hardware description languages such as VHDL or Verilog, designers can specify the desired functionality at a high level of abstraction known as a behavioral specification as opposed to a low level or structural specification. By using a behavioral specification, designers avoid committing to a particular hardware implementation.

The process of taking a behavioral specification and generating a low level hardware specification is called *behavioral synthesis*. While low level optimizations such as binding, allocation and scheduling are performed during synthesis, only a few high level optimizations may be performed. For example, the behavioral synthesis compiler performs local optimizations, such as loop unrolling, but only when directed by the programmer. The behavioral specification is synthesized into a Register Transfer Level specification and then this used as input to a place and route tool which generates the device configuration file.

2.2 Target Application Domain

Due to their customizability, FPGAs are commonly used for applications that have significant amounts of fine-grain parallelism and possibly can benefit from non-standard numeric formats (*e.g.,* reduced data widths). Specifically, multimedia applications, including image and signal processing on 8-bit and 16-bit data, respectively, offer a wide variety of enormously popular applications that map well to FPGAs.

For example, a fundamental image processing algorithm consists of scanning a multi-dimensional image performing computation on a given pixel value and all its neighbors. Typically images are represented as multi-dimensional array variables, and the computation is expressed as a loop nest. Such applications exhibit abundant concurrency as well as temporal reuse of data. Examples of kernel computations that fall into this category include image correlation, Laplacian image operators, erosion/dilation operators and edge detection.

Fortunately, this domain of applications maps well to the capabilities of current parallelizing compiler analyses. Parallelizing compilers are most effective in the *affine* domain, where array subscript expressions are linear functions of the loop index variables and constants [24]. In the work described in this paper, we restrict input programs to loop nest computations on array and scalar variables (no pointers), where all subscript expressions are affine with a fixed stride. The loop bounds must be constant.[1] We support loops with control flow, but to simplify control and scheduling, the generated code always performs conditional memory accesses.

We illustrate the concepts discussed in this paper using a synthetic benchmark, a machine vision kernel, depicted in Figure 1. For clarity, we have omitted some initialization and termination code as well as some of the numerical complexity of the algorithm. The code is structured as three loop nests nested inside

[1] Non-constant bounds could potentially be supported by the algorithm, but the generated code and resulting FPGA designs would be much more complex. For example, behavioral synthesis would transform a `for` loop with a non-constant bound to a `while` loop in the hardware implementation.

another control loop (not shown in the figure) that process a sequence of image frames. The first loop nest extracts image features using the Prewitt edge detector. The second loop nest determines where the peaks of the identified features reside. The last loop nest computes a sum square-difference between two consecutive images (arrays u and v). Using the data gathered for each image, another algorithm would estimate the position and velocity of the vehicle.

3 Communication and Pipelining Analyses

A key advantage of parallelizing compiler technology over behavioral synthesis is the ability to perform data dependence analysis on array variables. Analyzing communication requirements involves characterizing the relationship between data producers and consumers. This characterization can be thought of as a *data-flow analysis problem*. Our compiler uses a specific array data-flow analysis, *reaching definitions analysis* [2], to characterize the relationship between array accesses in different pipeline stages [15]. This analysis is used for the following purposes:

- Mapping each loop nest or straight line code segment to a pipeline stage.
- Determining which data must be communicated.
- Determining the possible granularities at which data may be communicated.
- Selecting the best granularity from this set.
- Determining the corresponding communication placement points within the program.

We combine reaching definitions information and array data-flow analysis for data parallelism [3] with task parallelism and pipelining information and capture it in an analysis abstraction called a Reaching Definition Data Access Descriptor (RDAD). RDADs are a fundamental extension of Data Access Descriptors (DADs) [7], which were originally proposed to detect the presence of data dependences either for data parallelism or task parallelism. We have extended DADs to capture reaching definitions information as well as summarize information about the read and write accesses for array variables in the high-level algorithm description, capturing sufficient information to automatically generate communication when dependences exist. Such RDAD sets are derived hierarchically by analysis at different program points, *i.e.,* on a statement, basic block, loop and procedure level. Since we map each nested loop or intervening statements to a pipeline stage, we also associate RDADs with pipeline stages.

Definition 1 *A Reaching Definition Data Access Descriptor, RDAD(A), defined as a set of 5-tuples* $\langle\ \alpha \mid \tau \mid \delta \mid \omega \mid \gamma\ \rangle$*, describes the data accessed in the* m*-dimensional array* A *at a program point* s*, where* s *is either a basic block, a loop or pipeline stage.* α *is an array section describing the accessed elements of array* A *represented by a set of integer linear inequalities.* τ *is the traversal order of* α*, a vector of length* $\leq m$*, with array dimensions from* $(1, \cdots, m)$ *as elements, ordered from slowest to fastest accessed dimension. A dimension traversed in reverse order is annotated as* $\bar{i}$*. An entry may also be a set of dimensions traversed at the same rate.* δ *is a vector of length* m *and contains the dominant induction variable for each dimension.* ω *is a set of definition or use points for which* α

captures the access information. γ is the set of reaching definitions. We refer to $RDAD_{r,s}(A)$ as the set of tuples corresponding to the reads of array A *and $RDAD_{w,s}(A)$ as the set of writes of array* A *at program point s. Since writes do not have associated reaching definitions, for all $RDAD_{w,s}(A)$, $\gamma = \emptyset$.*

After calculating the set of RDADs for a program, we use the reaching definitions information to determine between which pipeline stages communication must occur. To generate communication between pipeline stages, we consider each pair of write and read RDAD tuples where an array definition point in the sending pipeline stage is among the reaching definitions in the receiving pipeline stage. The communication requirements, *i.e.,* placement and data, are related to the *granularity of communication*. We calculate a set of valid granularities, based on the comparison of traversal order information from the communicating pipeline stages, and then evaluate the execution time for each granularity in the set to find the best choice. We define another abstraction, the Communication Edge Descriptor (CED), to describe the communication requirements on each edge connecting two pipeline stages.

$$RDAD_{w,s_1}(peak) = \left\langle \begin{array}{l} 0 \le d1 \le 13 \\ 0 \le d2 \le 13 \end{array} \middle| \langle 1,2 \rangle \middle| \langle x,y \rangle \middle| \{3\} \middle| \emptyset \right\rangle$$

$$RDAD_{r,s_1}(u) = \left\langle \begin{array}{l} 0 \le d1 \le 13 \\ 0 \le d2 \le 16 \end{array} \middle| \langle 1,2 \rangle \middle| \langle x,y \rangle \middle| \{1,2\} \middle| \emptyset \right\rangle$$

$$RDAD_{r,s_2}(peak) = \left\langle \begin{array}{l} 0 \le d1 \le 13 \\ 0 \le d2 \le 13 \end{array} \middle| \langle 1,2 \rangle \middle| \langle x,y \rangle \middle| \{4\} \middle| \{3\} \right\rangle$$

$$RDAD_{w,s_2}(feature\_x) = \left\langle \begin{array}{l} 0 \le d1 \le 13 \\ 0 \le d2 \le 13 \end{array} \middle| \langle 1,2 \rangle \middle| \langle x,y \rangle \middle| \{5,7\} \middle| \emptyset \right\rangle$$

$$RDAD_{w,s_2}(feature\_y) = \left\langle \begin{array}{l} 0 \le d1 \le 13 \\ 0 \le d2 \le 13 \end{array} \middle| \langle 1,2 \rangle \middle| \langle x,y \rangle \middle| \{6,8\} \middle| \emptyset \right\rangle$$

$$RDAD_{r,s_3}(feature\_x) = \left\langle \begin{array}{l} 0 \le d1 \le 13 \\ 0 \le d2 \le 13 \end{array} \middle| \langle 1,2 \rangle \middle| \langle x,y \rangle \middle| \{9\} \middle| \{5,7\} \right\rangle$$

$$RDAD_{r,s_3}(u) = \left\langle \begin{array}{l} 0 \le d1 \le 13 \\ 0 \le d2 \le 13 \end{array} \middle| \langle 1,2 \rangle \middle| \langle x,y \rangle \middle| \{10\} \middle| \emptyset \right\rangle$$

$$RDAD_{r,s_3}(v) = \left\langle \begin{array}{l} 0 \le d1 \le 13 \\ 0 \le d2 \le 13 \end{array} \middle| \langle 1,2 \rangle \middle| \langle x,y \rangle \middle| \{10\} \middle| \emptyset \right\rangle$$

$$RDAD_{w,s_3}(ssd) = \left\langle \begin{array}{l} 0 \le d1 \le 13 \\ 0 \le d2 \le 13 \end{array} \middle| \langle 1,2 \rangle \middle| \langle x,y \rangle \middle| \{10\} \middle| \emptyset \right\rangle$$

Fig. 2. MVIS Kernel Communication Analysis.

Definition 2 *A Communication Edge Descriptor (CED), $CED_{s_i \rightarrow s_j}(A)$, defined as a set of 3-tuples $\langle\ \alpha \mid \lambda \mid \rho\ \rangle$, describes the communication that must occur between two pipeline stages s_i and s_j. α is the array section, represented by a set of integer linear inequalities, that is transmitted on a per communication instance. λ and ρ are the communication placement points in the send and receive pipeline stages respectively.*

Figure 2 shows the calculated RDADs for pipeline stages $S1$ and $S2$, for arrays *peak*, *feature_x* and *feature_y*. These are used as input to calculate the set of CEDs. Figure 3 shows the set of CEDs representing the communication between pipeline stages $S1$ and $S2$. By inspection of the RDADs, we see that there is a reaching definition for array *p*eak from pipeline stage $S1$ to $S2$. This implies that communication must occur between these two stages. From the RDAD traversal order tuples, $\tau = \langle\ 1,\ 2\ \rangle$ we can see that both arrays are accessed in the same order in each stage and we may choose from among all possible granularities, e.g. whole array, row, and element. We calculate a CED for each granularity, capturing the data to be communicated each instance and the communication placement. We choose the best granularity, based on total program execution time, and apply code transformations to reflect the results of the analysis. The details of the analysis are found in [26].

Total Array-sized Communication

$$CED_{s_1 \rightarrow s_2}(peak) = \langle \begin{array}{c} 0 \leq d1 \leq 13 \\ 0 \leq d2 \leq 13 \end{array} \Big| 0 \Big| 0 \rangle$$

Element-sized Communication

$$CED_{s_1 \rightarrow s_2}(peak) = \langle \begin{array}{c} d1 = x \\ d2 = y \end{array} \Big| y \Big| y \rangle$$

Row-sized Communication

$$CED_{s_1 \rightarrow s_2}(peak) = \langle \begin{array}{c} d1 = x \\ 0 \leq d2 \leq 13 \end{array} \Big| x \Big| x \rangle$$

Best Communication

$$CED_{s_1 \rightarrow s_2}(peak) = \langle \begin{array}{c} d1 = x \\ 0 \leq d2 \leq 13 \end{array} \Big| x \Big| x \rangle$$

Fig. 3. MVIS Kernel Communication Analysis.

4 Optimization Strategy

In this section, we set forth our strategy for solving the global optimization problem. We briefly describe the criteria, behavioral synthesis estimates, and metrics used for local optimization, as published in [20, 19] and then describe how we build upon these to find a global solution.

A high-level design flow is shown in Figure 4. The shaded boxes represent a collection of transformations and analyses, discussed in the next section, that may be applied to the program. At a high-level, the design space exploration algorithm involves selecting parameters for a set of transformations for the loop nests in a program. By choosing specific unroll factors and communication granularities for each loop nest or pair of loop nests, we partition the chip capacity and ultimately the memory bandwidth among the pipeline stages. The generated VHDL is input into the behavioral synthesis compiler to derive a performance and area estimates for each loop nest. From this information, we use *balance*

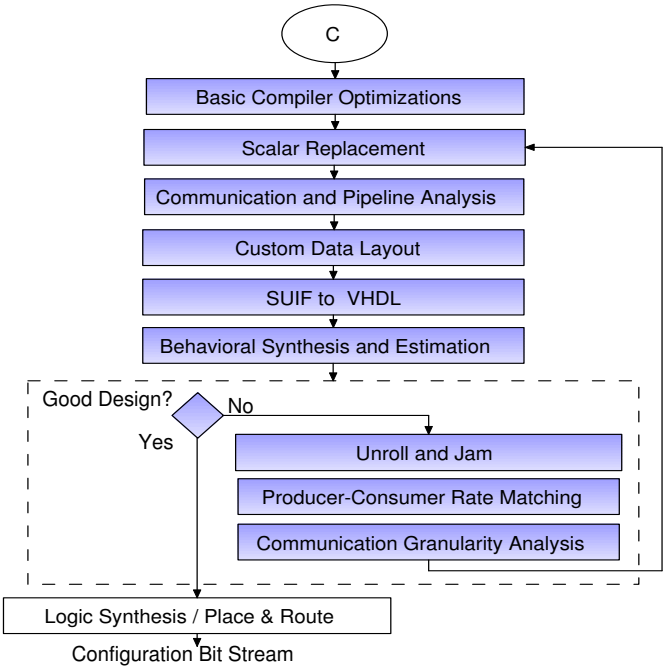

Fig. 4. High-Level Design Flow

and *efficiency*, along with our 2 optimization criteria to tune the transformation parameters to obtain the best performance and also determine if we have a good design.

The two optimization criteria, for mapping a single loop nest,

1. The design's execution time should be minimized.
2. The design's space usage, for a given performance, should be minimized.

are still valid for mapping a pipelined computation to an FPGA but the way in which we calculate the input and evaluate these criteria has changed. The $area(d)$ of design d, related to criterion 2, is a summation of the individual behavioral synthesis estimates of the FPGA area used for the data path, control and communication for each pipeline stage in this design. The $time(d)$ of design d, related to criterion 1, is a summation of the behavioral synthesis estimates for each pipeline stage of the number of cycles it takes to run to completion, including the time used to communicate data and excluding time saved by the overlap of communication and computation.

5 Transformations

We define a set of transformations, widely used in conventional computing, that permit us to adjust computational and memory parallelism in FPGA-based systems through a collaboration between parallelizing compiler technology and high-level synthesis. To meet the optimization criteria set forth in the previous

section, we have reduced the optimization process to a tractable problem, that of selecting a set of parameters, for local transformations applied to a single loop nest or global transformations applied to the program as a whole, that lead to a high-performance, balanced, and efficient design.

5.1 Transformations for Local Optimization

Unroll and Jam Due to the lack of dependence analysis in synthesis tools, memory accesses and computations that are independent across multiple iterations must be executed in serial. Unroll and jam [9], where one or more loops in the iteration space are unrolled and the inner loop bodies are fused together, is used to expose fine-grain operator and memory parallelism by replicating the logical operations and their corresponding operands in the loop body. Following unroll-and-jam, the parallelism exploited by high-level synthesis is significantly improved.

Scalar Replacement. This transformation replaces array references by accesses to temporary scalar variables, so that high-level synthesis will exploit reuse in registers. Our approach to scalar replacement closely matches previous work [9]. There are, however, two differences: (1) we also eliminate unnecessary memory writes on output dependences; and, (2) we exploit reuse across all loops in the nest, not just the innermost loop. We peel iterations of loops as necessary to initialize registers on array boundaries. Details can be found in [12].

Custom Data Layout This code transformation lays out the data in the FPGA's external memories so as to maximize memory parallelism. The compiler performs a 1-to-1 mapping between array locations and virtual memories in order to customize accesses to each array according to their access patterns. The result of this mapping is a distribution of each array across the virtual memories such that opportunities for parallel memory accesses are exposed to high-level synthesis. Then the compiler binds virtual memories to physical memories, taking into consideration accesses by other arrays in the loop nest to avoid scheduling conflicts. Details can be found in [21].

5.2 Transformations for Global Optimization

Communication Granularity and Placement With multiple, pipelined tasks (*i.e.,* loop nests), some of the input/output data for a task may be directly communicated on chip, rather than requiring reading and/or writing from/to memory. Thus, some of the memory accesses assumed in the optimization of a single loop nest may be eliminated as a result of communication analysis.

The previously-described communication analysis selects the communication granularity that maximizes the overlap of communication and computation, while amortizing communication costs over the amount of data communicated. This granularity may not be ideal when other issues, such as on-chip space constraints, are taken into account. For example, if the space required for on-chip buffering is not available, we might need to choose a finer granularity of communication. In the worst case, we may move the communication off-chip altogether.

Data Reorganization On-chip As part of the single loop solution, we calculated the best custom data layout for each accessed array variable, allowing for

a pipeline stage to achieve its best performance. When combining stages that access the same data either via memory or on-chip communication on the same FPGA, the access patterns for each stage may be different and thus optimal data layouts may be incompatible. One strategy is to reorganize the data between loop nests to retain the locally optimal layouts. In conventional systems, data reorganization can be very expensive in both CPU cycles and cache or memory usage, and as a result, usually carries too much overhead to be profitable. In FPGAs, we recognize that the cost of data reorganization is in many cases quite low. For data communicated on-chip between pipeline stages that is already consuming buffer space, the additional cost of data reorganization is negligible in terms of additional storage, and because the reorganization can be performed completely in parallel on an FPGA, the execution time overhead may be hidden by the synchronization between pipeline stages. The implementation of on-chip reorganization involves modifying the control in the finite state machine for each pipeline stage, which is done automatically by behavioral synthesis; the set of registers containing the reorganized array will simply be accessed in a different order. The only true overhead is the increased complexity of routing associated with the reorganization; this in turn would lead to increased space used for routing as well as a potentially slower achieved clock rate.

6 Search Space Properties

The optimization involves selecting unroll factors, due to space and performance considerations, for the loops in the nest of each pipeline stage. Our search is guided by the following observations about the impact of the unroll factor and other optimizations for a single loop in the nest. In order to define the global design space, we discuss the following observations:

Observation 1 *As a result of applying communication analysis, the number of memory accesses in a loop is non-increasing as compared to the single loop solution without communication.*

The goal of communication analysis is to identify data that may be communicated between pipeline stages either using an on or off-chip method. The data that is may now be communicated via on-chip buffers would have been communicated via off-chip memory prior to this analysis.

Observation 2 *Starting from the design found by applying the single loop with communication solution, the unroll factors calculated during the global optimization phase will be non-increasing.*

We start by applying the single loop optimizations along with communication analysis. We assume that this is the best balanced solution in terms of memory bandwidth and chip capacity usage. We also assume that the ratio of performance to area has the best efficiency rating as compared to other designs investigated during the single loop exploration phase. Therefore, we take this result to be the worst case space estimate and the best case performance achievable by this stage in isolation; unrolling further would not be beneficial to finding the best design.

Observation 3 *When the producer and consumer data rates for a given communication event are not equal, we may decrease the unroll factor of the faster pipeline stage to the point at which the rates are equal. We assume that reducing the unroll factor does not cause this pipeline stage to become the bottleneck of the pipeline.*

When comparing two pipeline stages between which communication occurs, if the rates are not matched, the implementation of the faster stage may be using an unnecessarily large amount of the chip capacity while not contributing to the overall performance of the program. This is due to the fact that performance is limited by the slower pipeline stage. We may choose a smaller unroll factor for the faster stage such that the rates match. Since the slower stage is the bottleneck, choosing a smaller unroll factor for the faster stage does not affect the overall performance of the pipeline until the point at which the faster stage becomes the slower stage.

Finally, if a pipeline stage is involved in multiple communication events, we must take care to decrease the unroll factor based on the constraints imposed by all events. We do not reduce the unroll factor of a stage to the point that it becomes a bottleneck.

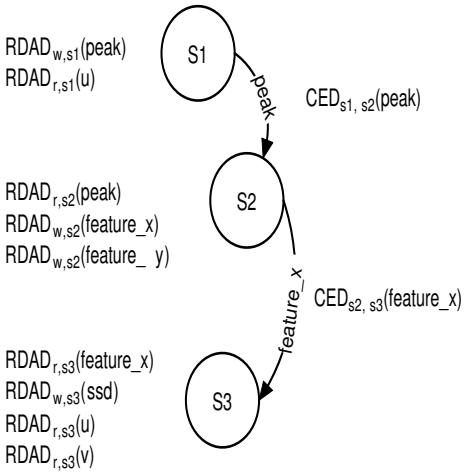

Fig. 5. MVIS Task Graph

6.1 Optimization Algorithm

At a high-level, the design space exploration algorithm involves selecting parameters for a set of transformations for the loop nests in a program. By choosing specific unroll factors and communication granularities for each loop nest or pair of loop nests, we partition the chip capacity and ultimately the memory bandwidth among the pipeline stages. The generated VHDL is input into the behavioral synthesis compiler to derive a performance and area estimates for each loop nest. From this information, we can tune the transformation parameters to obtain the best performance.

The algorithm represents a multiple loop nest computation as an acyclic task graph to be mapped onto a pipeline with no feedback. To simplify this discussion, we describe the task graph for a single procedure, although interprocedural task graphs are supported by our implementation. Each loop nest or computation between loop nests is represented as a node in the task graph. Each has a set of associated RDADs. Edges, each described by a CED, represent communication events between tasks. There is one producer and one consumer pipeline stage per edge. The task graph for the MVIS kernel is shown in Figure 5. Associated with each task is the current best hardware implementation, with both area and performance estimates, and balance and efficiency metrics.

1. We apply the communication and pipelining analyses to 1) define the stages of the pipeline and thus the nodes of the task graph and 2) identify data which could be communicated from one stage to another and thus define the edges of the task graph.
2. In reverse topological order, we visit the nodes in the task graph to identify communication edges where producer and consumer rates are out of balance. From Observation 3, if reducing a producer or consumer rate does not cause a task to become a bottleneck in the pipeline, we may modify it.
3. We compute the area of the resulting design, which we currently assume is the sum of the areas of the single loop nest designs, including the communication logic and buffers. If the space utilization exceeds the device capacity, we employ a greedy strategy to reduce the area of the design. We select the largest task in terms of area, and reduce its unroll factor.
4. Repeat steps two and three until the design meets the space constraints of the target device.

Our initial algorithm employs a greedy algorithm to reduce space constraints, but other heuristics may be considered in future work, such as reducing space of tasks not on the critical path, or using the balance and efficiency metrics to suggest which tasks will be less impacted by reducing unroll factors.

7 Experiments

We have implemented the loop unrolling, the communication analysis, scalar replacement, data layout, the single loop design space exploration and the translation from SUIF to behavioral VHDL such that these analyses and transformations are automated. Individual analysis passes are not fully integrated, requiring minimal hand intervention.

We examine how the number of memory accesses has changed when comparing the results of the automated local optimization and design space exploration with and without applying the communication analyses. In Table 1 we show the number of memory accesses in each pipeline stage before and after applying communication analysis. The rows entitled Accesses Before and After are the results without and with communication analysis respectively. As a result of the communication analysis, the number of memory accesses greatly declines

for all pipeline stages. In particular, for pipeline stage $S2$, the number of memory accesses goes to zero because all consumed data is communicated on-chip from stage $S1$ and all produced data is communicated on-chip to stage $S3$. This should have a large impact on the performance of the pipeline stage. For pipeline stages $S1$ and $S3$, the reduction in the number of memory accesses may be sufficient to transform the pipeline stage from a memory bound stage into a compute bound stage. This should also improve performance of each pipeline stage and ultimately the performance of the total program.

Table 1. Memory Access Reduction.

Pipeline Stage	1	2	3
Accesses Before	49	117	45
Accesses After	2	0	6

From the design space exploration for each single loop, we would choose unroll factors of 4, 4, and 2 for pipeline stages $S1, S2$, and $S3$. This is based on both the best balance numbers and area as explained in [27].

We then apply the design space exploration with global optimizations. Since the sum of the areas, 306K Monet space units, for the implementation for all three pipeline stages with the previously mentioned unroll factors is larger than the total area of the chip (150K), we must identify one or more pipeline stages for which to decrease the unroll factors. We apply the second step of our algorithm, which matches producer and consumer rates throughout the pipeline. Since $S3$ is the bottleneck when comparing the rates between stages $S2$ and $S3$, we know that we may reduce the unroll factor of stage $S2$ to 2 without affecting the pipeline performance. Then, our algorithm will detect a mismatch between stages $S1$ and $S2$. Again, we may decrease the unroll factor of stage $S1$ from 4 to 2 without affecting performance. Then we perform the analyses once again on each pipeline stage, using the new unroll factor of 2 for all pipeline stages. The size of the resulting solution is 103K Monet units. We are now within our space constraint.

In summary, by eliminating memory accesses through scalar replacement and communication analysis, and by then matching producer and consumer data rates for each pipeline stage, we were able to achieve a good mapping while eliminating large parts of the search space.

8 Related Work

In this section we discuss related work in the areas of automatic synthesis of hardware circuits from high-level language constructs, array data-flow analysis, pipelining and design space exploration using high-level loop transformations.

8.1 Synthesizing High-Level Constructs

Languages such as VHDL and Verilog allow programmers to migrate to configurable architectures without having to learn a radically new programming paradigm. Efforts in the area of new languages include Handel-C [18]. Several researchers have developed tools that map computations to reconfigurable custom computing architectures [23], while others have developed approaches to mapping applications to their own reconfigurable architectures that are not FPGAs,

e.g., RaPiD [10] and PipeRench [14]. The two projects most closely related to ours, the Nimble compiler and work by Babb *e*t al. [6], map applications in C to FPGAs, but do not perform design space exploration.

8.2 Design Space Exploration

In this discussion, we focus only on related work that has attempted to use loop transformations to explore a wide design space. Other work has addressed more general issues such as finding a suitable architecture (either reconfigurable or not) for a particular set of applications (*e*.g., [1]). Derrien/Rajopadhye [11] describe a tiling strategy for doubly nested loops. They model performance analytically and select a tile size that minimizes the iteration's execution time. Cameron's estimation approach builds on their own internal data-flow representation using curve fitting techniques [17].

8.3 Array Data-Flow Analysis

Previous work on array data flow analysis [7, 22, 3] focused on data dependence analysis but not at the level of precision required to derive communication requirements for our platform. Parallelizing compiler communication analysis techniques [4, 16] exploited data parallelism.

8.4 Pipelining

In [5] Arnold created a software environment to program a set of FPGAs connected to a workstation; Callahan and Wawrzynek [8] used a VLIW-like compilation scheme for the GARP project; both works exploit intra-loop pipelined execution techniques. Goldstein *et al.* [14] describes a custom device that implements an execution-time reconfigurable fabric. Weinhardt and Luk [23] describes a set of program transformations to map the pipelined execution of loops with loop-carried dependences onto custom machines. Du *et al.* [13] provide compiler support for exploiting coarse-grained pipelined parallelism in distributed systems.

8.5 Discussion

The research presented in this paper differs from the efforts mentioned above in several respects. First the focus of this research is in developing an algorithm that can explore a wide number of design points, rather than selecting a single implementation. Second, the proposed algorithm takes as input a sequential application description and does not require the programmer to control the compiler's transformations. Third, the proposed algorithm uses high-level compiler analysis and estimation techniques to guide the application of the transformations as well as evaluate the various design points. Our algorithm supports multi-dimensional array variables absent in previous analyses for the mapping of loop computations to FPGAs. Fourth, instead of focusing on intra-loop pipelining techniques that optimize resource utilization, we focus on increased throughput through task parallelism coupled with pipelining, which we believe is a natural match for image processing data intensive and streaming applications. Finally, we use a commercially available behavioral synthesis tool to complement the parallelizing compiler techniques rather than creating an architecture-specific

synthesis flow that partially replicates the functionality of existing commercial tools. Behavioral synthesis allows the design space exploration to extract more accurate performance metrics (time and area used) rather than relying on a compiler-derived performance model. Our approach greatly expands the capability of behavioral synthesis tools through more precise program analysis.

9 Conclusion

In this paper, we describe how parallelizing compiler technology can be adapted and integrated with hardware synthesis tools, to automatically derive, from sequential C programs, pipelined implementations for systems with multiple FPGAs and memories. We describe our implementation of these analyses in the DEFACTO system, and demonstrate this approach with a case study. We presented experimental results, derived, in part, automatically by our system. We show that we are able to reduce the size of the search space by reasoning about the maximum unroll factors and matching producer and consumer rates. While we employ a greedy search algorithm here, we plan to investigate tradeoffs between and effects of adjusting unroll factors for pipeline stages both on and off the critical path. Once our design is within the space constraints of the chip capacity, we will continue to search for the best allocation of memory bandwidth. Finally, we may find that we are able to reduce the number of memory accesses to the point that we may want to unroll a loop further than the original single loop solution to meet our optimization criteria.

References

1. Santosh Abraham, Bob Rau, Robert Schreiber, Greg Snider, and Michael Schlansker. Efficient design space exploration in PICO. Technical report, HP Labs, 1999.
2. A. Aho, R. Sethi, and J. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley Publishing, 1988.
3. S. Amarasinghe. *Parallelizing Compiler Techniques Based on Linear Inequalities*. PhD thesis, Dept. of Electrical Engineering, Stanford University, Jan 1997.
4. S. Amarasinghe and M. Lam. Communication optimization and code generation for distributed memory machines. In *Proc. ACM Conf. Programming Languages Design and Implementation*, pages 126–138, Albuquerque, 1993.
5. J. Arnold. The Splash 2 software environment. In *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, pages 88–93, 1993.
6. J. Babb, M. Rinard, C. Moritz, W. Lee, M. Frank, R. Barua, and S. Amarasinghe. Parallelizing applications into silicon. In *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, pages 70–81, 1999.
7. 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.
8. T. Callahan and J. Wawrzynek. Adapting software pipelining for reconfigurable computing. In *Proc. Intl. Conf. Compilers, Architectures and Synthesis for Embedded Systems*, pages 57–64, Nov 2000.
9. S. Carr and K. Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions Programming Languages and Systems*, 16(6):400–462, 1994.

10. D. Cronquist, P. Franklin, S. Berg, and C. Ebeling. Specifying and compiling applications for RaPiD. In *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, pages 116–125, 1998.
11. Steven Derrien, Sanjay Rajopadhye, and Susmita Sur Kolay. Combined instruction and loop parallelism in array synthesis for FPGAs. In *Proc. 14th Intl. Symp. System Synthesis*, pages 165–170, 2001.
12. P. Diniz, M. Hall, J. Park, B. So, and H. Ziegler. Bridging the gap between compilation and synthesis in the DEFACTO system. In *Proc. 14th Workshop Languages and Compilers for Parallel Computing*, LNCS. Springer-Verlag, 2001.
13. Wei Du, Renato Ferreira, and Gagan Agrawal. Compiler support for exploiting coarse-grained pipelined parallelism. In *to appear in Proc. Super Computing*, ACM SIGPLAN Notices, Nov. 2003.
14. 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.
15. M. Hall, S. Amarasinghe, B. Murphy, S. Liao, and M. Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Proc. Ninth Intl. Conf. Supercomputing*, pages 1–26, 1995.
16. S. Hiranandani, K. Kennedy, and C.-W. Tseng. Preliminary experiences with the FortranD compiler. In *Proc. Seventh Intl. Conf. Supercomputing*, Portland, Nov 1993.
17. W. Najjar, D. Draper, A. Bohm, and R. Beveridge. The Cameron project: high-level programming of image processing applications on reconfigurable computing machines. In *Proc. 7th Intl. Conf. Parallel Architecturs and Compilation Techniques - Workshop Reconfigurable Computing*, 1998.
18. I. Page and W. Luk. Compiling occam into FPGAs. In *Field Programmable Gate Arrays*, pages 271–283. Abigdon EE and CS Books, 1991.
19. B. So, P.C. Diniz, and M.W. Hall. Using estimates from behavioral synthesis tools in compiler-directed design space exploration. In *Proc. 40th Design Automation Conference*, June 2003.
20. B. So, M. Hall, and P. Diniz. A compiler approach to fast design space exploration in FPGA-based systems. In *Proc. ACM Conf. Programming Languages Design and Implementation*, pages 165–176, June 2002.
21. B. So, H. Ziegler, and M. Hall. A compiler approach for custom data layout. In *Proc. 14th Workshop Languages and Compilers for Parallel Computing*, July, 2002.
22. 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.
23. M. Weinhardt and W. Luk. Pipelined vectorization for reconfigurable systems. In *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, pages 52–62, 1999.
24. M. Wolfe. *Optimizing Supercompilers for Supercomputers*. Addison, 1996.
25. Xilinx Inc. *Virtex-II Pro^{TM} Platform FPGAs: introduction and overview*, DS083-1(v2.4.1) edition, March 2003.
26. H. Ziegler, M. Hall, and P. Diniz. Compiler-generated communication for pipelined FPGA applications. In *Proc. 40th Design Automation Conference*, June 2003.
27. H. Ziegler, B. So, M. Hall, and P. Diniz. Coarse-grain pipelining on multiple FPGA architectures. In *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, April 2002.