Compiler-Generated Communication for Pipelined FPGA Applications*

Heidi E. Ziegler,† Mary W. Hall, and Pedro C. Diniz
University of Southern California / Information Sciences Institute
4676 Admiralty Way, Suite 1001
Marina del Rey, CA 90292
{ziegler, mhall, pedro}@isi.edu

ABSTRACT

In this paper, we describe a set of compiler analyses and an implementation that automatically map a sequential and un-annotated C program into a pipelined implementation, targeted for an FPGA with multiple external memories. For this purpose, we extend array data-flow analysis techniques from parallelizing compilers to identify pipeline stages, required inter-pipeline stage communication, and opportunities to find a minimal program execution time by trading communication overhead with the amount of computation overlap in different stages. Using the results of this analysis, we automatically generate application-specific pipelined FPGA hardware designs. We use a sample image processing kernel to illustrate these concepts. Our algorithm finds a solution in which transmitting a row of an array between pipeline stages per communication instance leads to a speedup of 1.76 over an implementation that communicates the entire array at once.

Categories and Subject Descriptors

B.6.3 [**Design Aids**]: Automatic synthesis; J.6 [**Computer Applications**]: Computer-aided design (CAD)

General Terms

Hardware Design, Performance

Keywords

Pipelining, Parallelizing compiler analysis techniques, Synthesis techniques for configurable computing, High-level and architectural synthesis, Rapid prototyping, FPGAs

1. INTRODUCTION

The use of pipelined execution is an effective method to improve the throughput of a program. FPGA-based computing machines offer a unique opportunity for the realization of custom pipelining structures, matching the definition of the pipeline to the application requirements in terms of pipeline stage definitions, computation that can be overlapped, the data and rates at which it can be communicated as well as the communication placement within the pipeline stages.

The complexity and sophistication of pipelined execution make automatic tools that can analyze sequential applications and derive pipelined implementations extremely desirable. In this paper, we describe a set of compiler analyses to derive, from a sequential algorithm description, the components of a pipelined design, *i.e.,* task parallelism and communication requirements. We use as a foundation parallelizing compiler analyses for array data-flow analysis [3, 6], which we extend to recognize pipelining opportunities and derive communication requirements, for use in mapping to FPGA systems. In the Design Environment for Adaptive Computing TechnOlogy (DEFACTO) [8], we combine these analyses with behavioral synthesis tools to automatically synthesize application-specific pipelines onto a target FPGA-based architecture. The work described in this paper makes the following specific contributions:

- It defines new analyses for characterizing the task parallelism and communication requirements for use in mapping sequential programs to systems of configurable logic.
- It describes an implementation of the analyses and code transformations required to automatically design and synthesize pipelines tailored to sequential program characteristics.
- It presents experimental results for a machine vision application excerpt (MVIS) which demonstrate the use and performance potential of these techniques on an FPGA-based architecture.

The paper is organized as follows. In section 2 we describe the problem we are addressing along with an example that illustrates the approach. Section 3 describes the compiler analysis in more detail. We present the execution times for four communication schemes in section 4. In section 5 we survey related work and conclude in section 6.

*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.
†H. Ziegler is funded by a Boeing Satellite Systems Doctoral Scholars Fellowship.

DAC 2003, June 2–6, 2003, Anaheim, California, USA.
Copyright 2003 ACM 1-58113-688-9/03/0006 ...$5.00.

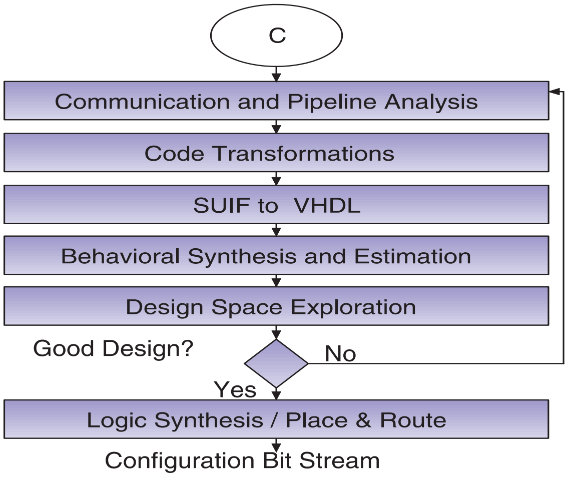
C
Communication and Pipeline Analysis
Code Transformations
SUIF to VHDL
Behavioral Synthesis and Estimation
Design Space Exploration
Good Design?
No
Yes
Logic Synthesis / Place & Route
Configuration Bit Stream

Figure 2: DEFACTO Compilation Flow

2. MOTIVATION AND BACKGROUND

The problem we address in this paper is automatically mapping an application onto an asynchronous pipeline, executing on a configurable architecture [16]. Mapping a pipelined application involves identifying a set of pipeline stages and the associated communication. The compilation goal is to minimize overall execution time, while meeting the storage and computing capacity constraints of the system. We exploit the parallelism in the application without the use of any programmer inserted pragmas or directives.

The compiler approach described in this paper, although generic in the sense of mapping a set of communicating pipeline stages to a configurable architecture, has focused on applications whose computations are specified by sequences of loop nests with intervening statements. These loop nests, not necessarily perfectly nested, compute over array data structures, affine index access functions, defined as linear functions of loop index variables, and constant loop bounds. For the current implementation, we do not map to hardware computations with pointer accesses. Under these assumptions, we have been able to apply our compiler analysis approach to digital image and signal processing kernels and other regular array computations of interest.

2.1 Example

We illustrate the mapping of MVIS, depicted in Figure 1(a). The code is structured as two loop nests. In this example, pipeline stage s_1 corresponds to the computation in the first loop nest, stage s_2 to the second loop nest. S_1 computes the *peak* array; s_2 reads *peak* and computes the arrays *feature_x* and *feature_y* as output. We determine that we must communicate the whole array *peak* from the producer s_1 to the consumer s_2, one row at a time.

3. COMPILER ANALYSIS

The compiler analysis described in this paper is built upon an automatic parallelization system that is part of the Stanford SUIF compiler [1]. Figure 2 depicts the set of compiler analyses implemented specifically for DEFACTO.

The DEFACTO system-level compiler takes an un-annotated, sequential program and applies a set of communication and pipeline analyses based on array data-flow analysis. Previous work on array data-flow analysis has largely focused on identifying loops whose iterations can execute in parallel. This is called *data parallelism*, since the same code is executed in parallel on different data. In this paper, we extend and develop array data-flow analysis that supports pipelining of independent computations, or *task parallelism.* These extended analyses are incorporated into the communication and pipeline analysis phase and include the following:

- Determining which data must be communicated.
- Determining the possible granularities at which data may be communicated.
- Determining the corresponding communication placement points within the program.

Then code transformations are applied to reflect the results of the analysis and the SUIF intermediate format is converted into behavioral VHDL. Estimates from the behavioral synthesis phase are used to evaluate each of the possible granularity solutions in the design space exploration phase; a good design can then be passed onto the logic synthesis and place and route phase.

3.1 Analysis Background

Analyzing communication requirements involves characterizing the relationship between data producers and consumers. This characterization can be thought of as a *data-flow analysis problem.* In compiler terminology, data-flow analysis is the compile-time reasoning about the run-time flow of data values through the program. For decades, data-flow analysis has been used to guide a host of compiler optimizations and has even been incorporated into high-level synthesis tools. For the most part, the analysis has been restricted to scalar variables. Data structures, such as arrays, are treated as a single entity.

Unfortunately, scalar data-flow analysis is too imprecise when optimizing designs that access multi-dimensional array variables, such as commonly occur in multimedia algorithms. To be effective, the analysis must track accesses to individual array elements. For this purpose, we draw on solutions from parallelizing compiler technology to derive *array data-flow analysis* information. Our compiler uses a specific array data-flow analysis, *reaching definitions analysis* [2], to characterize the relationship between array accesses in different pipeline stages [11]. Reaching definitions are variable values that are not *killed*, *i.e.,* not redefined, at another definition point occurring in the data flow from predecessor program points ($p \in pred(s)$) to the current program point s. Definitions occurring at the current program point form a $gen(s)$ set. We typically talk about program points as being *basic blocks.* While in our system we also calculate reaching definitions at the basic block level, we perform a hierarchical analysis that ultimately derives reaching definitions at the level of pipeline stages. For the purposes of deriving communication requirements, the analysis only retains reaching definitions information when a definition in one pipeline stage reaches a use in another pipeline stage. Reaching definitions, $\gamma(s)$, are defined by a set of simultaneous equations represented by Equation 1.

$$\gamma(s) = gen(s) \cup (\cup_{p \in pred(s)} \gamma(p)) \cap \neg killed(s) \qquad (1)$$

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

```
#define IMAGE 32
int u[IMAGE][IMAGE];
int peak[IMAGE][IMAGE];
int feature_x[IMAGE][IMAGE];
int feature_y[IMAGE][IMAGE];
int th, uh1, uh2;

/* stage s1. Apply SOBEL Operator */
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;
      }
  }
}
```

(a) Machine Vision Kernel

$$RDAD_{w,s_1}(peak) = \left\langle \begin{array}{c} 0 \le d1 \le 29 \\ 0 \le d2 \le 29 \end{array} \middle| \langle 1,2\rangle \middle| \langle x,y\rangle \middle| \{3\} \middle| \emptyset \right\rangle$$

$$RDAD_{r,s_2}(peak) = \left\langle \begin{array}{c} 0 \le d1 \le 29 \\ 0 \le d2 \le 29 \end{array} \middle| \langle 1,2\rangle \middle| \langle x,y\rangle \middle| \{4\} \middle| \{3\} \right\rangle$$

$$RDAD_{w,s_2}(feature\_x) = \left\langle \begin{array}{c} 0 \le d1 \le 29 \\ 0 \le d2 \le 29 \end{array} \middle| \langle 1,2\rangle \middle| \langle x,y\rangle \middle| \{5,7\} \middle| \emptyset \right\rangle$$

$$RDAD_{w,s_2}(feature\_y) = \left\langle \begin{array}{c} 0 \le d1 \le 29 \\ 0 \le d2 \le 29 \end{array} \middle| \langle 1,2\rangle \middle| \langle x,y\rangle \middle| \{6,8\} \middle| \emptyset \right\rangle$$

(b) Array Data-Flow Analysis

Total Array-sized Communication

$$CED_{s_1 \to s_2}(peak) = \left\langle \begin{array}{c} 0 \le d1 \le 29 \\ 0 \le d2 \le 29 \end{array} \middle| 0 \middle| 0 \right\rangle$$

Row-sized Communication

$$CED_{s_1 \to s_2}(peak) = \left\langle \begin{array}{c} d1 = x \\ 0 \le d2 \le 29 \end{array} \middle| x \middle| x \right\rangle$$

Element-sized Communication

$$CED_{s_1 \to s_2}(peak) = \left\langle \begin{array}{c} d1 = x \\ d2 = y \end{array} \middle| y \middle| y \right\rangle$$

Best Communication

$$CED_{s_1 \to s_2}(peak) = \left\langle \begin{array}{c} d1 = x \\ 0 \le d2 \le 29 \end{array} \middle| x \middle| x \right\rangle$$

(c) Possible Communication Schemes

```
architecture a_main of main is
  signal dataValid1 : boolean;
  signal gotData0 : boolean;
  signal dPass0 : Arr060000002;

begin
p0: process
  variable peak : Arr022;
  variable uh1, uh2 : SIGNED_INT_32;
  variable dPass0local : Arr060000002;
  variable edge0ready : boolean;
  variable calcDoneInner : boolean;
  variable calcDone : boolean;

begin
  main_loop: loop
    FOR x IN 0 to 28 loop
      wait until clk'event;
      if (gotData0) then
        unSetSignal(dataValid1);
      end if;
      unSetVariable(edge0ready);
      unSetVariable(calcDoneInner);

      calc_loop: loop
        if (edge0ready & NOT calcDone) then
          FOR y IN 0 to 28 loop
            wait until clk'event;
            uh1 := -3 * u(x*32+y) - ..;
            uh2 := 3 * u(x*32+y) + ..;
            peak(x*32+y) := uh1 + uh2;
            dPass0local(y) := peak(x*32+y);
          end loop;
          -- send a row of peak
          dPass0 <= dPass0local;
          setSignal(dataValid1);
          setVariable(calcDoneInner);
        end if;
        wait until clk'event;
        exit calc_loop when calcDoneInner;
      end loop calc_loop;
    end loop;
    setVariable(calcDone);
  end loop main_loop;
end process p0;
```

```
p1: process
  variable peak : Arr022;
  variable feature_x : Arr022;
  variable feature_y : Arr022;
  variable th : SIGNED_INT_32;
  variable calcDoneInner : boolean;
  variable calcDoneOuter : boolean;
  variable dPass0local : Arr060000002;

begin
  main_loop: loop
    wait until clk'event;
    FOR x IN 0 to 28 loop
      wait until clk'event;

      calc_loop: loop
        wait until clk'event;
        if (dataValid1) then
          setSignal(gotData0);
          -- receive a row of peak
          dPass0local := dPass0;
        end if;
        wait until clk'event;
        if (gotData0) then
          FOR y IN 0 to 28 loop
            wait until clk'event;
            peak(x*32+y) := dPass0local(y);
            if (th < peak(x*32+y)) then
              feature_x(x*32+y) := x;
              feature_y(x*32+y) := y;
            else
              feature_x(x*32+y) := 0;
              feature_y(x*32+y) := 0;
            end if;
          end loop;
          setVariable(calcDoneInner);
        end if;
        exit calc_loop when calcDoneInner;
      end loop calc_loop;

    end loop;
  end loop main_loop;
end process p1;
end a_main;
```

(d) VHDL Output

Figure 1: MVIS Kernel Analysis.

Descriptor (RDAD). RDADs are a fundamental extension of Data Access Descriptors (DADs) [6], which were originally proposed to detect the presence of data dependences either for data parallelism or task parallelism, but DADs do not capture sufficient information to automatically generate communication when dependences exist. For this purpose, we have extended DADs to capture reaching definitions information. Second, we have developed RDADs in an existing array data-flow analysis implementation, described by Amarasinghe [3], that derives more precise array sections than that of the DAD definition. For example, it can represent array regions that have holes and some nonlinear array regions. Finally, we have added a tuple containing the dominant induction variable (DIV) for each dimension, ordered according to the array traversal order [6]. A dominant induction variable is defined as the loop index variable that is changing the fastest in a given access expression. In the DAD analysis, the dominant induction variables were used to calculate the traversal order. We relax the DAD restriction on traversal orders in order to increase the opportunity for pipelining and then use the dominant induction variables to aid in selecting communication placement. A full discussion of how we combine standard scalar reaching defintions and array data-flow analysis with task parallelism and pipelining information is beyond the scope of this paper.

We present the RDAD definition in the next section and the definition of another abstraction that captures specific communication requirements between pipeline stages, the Communication Edge Descriptor (CED), in section 3.3. We also show how RDADs are used by the compiler to automatically calculate CEDs in section 3.4.

3.2 RDAD Description

Reaching Definition Data Access Descriptors (RDADs) summarize information about the read and write accesses for array variables in the high-level algorithm description. 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 `for` loop or intervening statements to a pipeline stage, we also associate RDADs with pipeline stages. Loop variables are normalized before calculating the set of associated RDADs.

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* $RDAD_{w,s}(A)$, $\gamma = \emptyset$.

In the following, we use the notation f(RDAD(A)), when selecting a tuple, f, on which to perform an operation. For example, to select the array section α, we write $\alpha(RDAD(A))$.

For this example, we show the calculated RDADs in Figure 1(b). The compiler determines that an access to array *peak*, in statement 3, writes the entire array, as described by

$$\alpha(RDAD_{w,s_1}(peak)) = \left| \begin{array}{l} 0 \leq d1 \leq 29 \\ 0 \leq d2 \leq 29 \end{array} \right|.$$

A read access to *peak* in statement 4 is described by $RDAD_{r,s_2}(peak)$. Similarly, the arrays *feature_x* and *feature_y* are written in statements 5, 6, 7, and 8. For all array accesses in the program, we capture the vector $\tau = \langle 1, 2\rangle$, indicating that dimension 1, the row dimension, varies more slowly than dimension 2. Similarly, we capture the dominant induction variables in $\delta = \langle x, y\rangle$, where x is the DIV corresponding to the row dimension access expression for each RDAD and also the enclosing loop in the nest with loop index variable x. Reaching definitions are retained in RDADs only if they reach outside of a pipeline stage. In the MVIS example, we have one reaching definition, from statement 3 to 4, from $RDAD_{w,s_1}(peak)$ to $RDAD_{r,s_2}(peak)$. We will show how these RDADs are used to calculate the specific communication between the two pipeline stages in section 3.4 and define the Communication Edge Descriptor next.

3.3 CED Description

We define another abstraction, the Communication Edge Descriptor, to describe the communication requirements on each edge connecting two pipeline stages.

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.*

3.4 Determining Communication Requirements

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 s_i and s_j we consider each pair of write and read RDAD tuples, R_i and R_j, where the definition point $\omega(R_i)$ is among the reaching definitions in $\gamma(R_j)$. 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. Once we identify the best granularity, we then calculate the specific communication placement and array data section to be communicated and form a CED. The algorithm is shown in Figure 3.

To calculate the valid set of granularities, the function $calcValidGran(R_i, R_j)$, shown in Figure 3, returns β, a vector of length $\leq m$, with array dimensions from $(1, \cdots, m)$ or $\emptyset$ as elements, ordered from slowest to fastest accessed dimension. The fastest moving, non-empty dimension of β represents the smallest possible communication granularity. The $calcValidGran$ function pairwise compares the input traversal order vectors starting from the slowest moving dimension; if two entries are identical, the corresponding β_k is assigned the same value. Once a non-matching pair of entries is detected, all remaining entries in β are set to $\emptyset$.

```
function calcValidGran($R_i, R_j$)
  for $k = 1$ to $m$ {
    if $\tau_k(R_i) = \tau_k(R_j)$
      $\beta_k = \tau_k(R_i)$
    else $\beta_k \cdots \beta_m = \emptyset$; break; }
  return $\beta$;

function findCommPlace($\beta, R_i, R_j$)
  /* initial condition: communicate whole array */
  $minTime = execTime_{s_i} + execTime_{s_j} +$
      $commTime_{s_i \rightarrow s_j}(\beta_0)$ - $overlapTime_{s_i \rightarrow s_j}(\beta_0)$
  $minDim = 0$;
  for $k = 1$ to $length(\beta)$ {
    $minTime_k = execTime_{s_i} + execTime_{s_j}$ +
      $commTime_{s_i \rightarrow s_j}(\beta_k)$ - $overlapTime_{s_i \rightarrow s_j}(\beta_k)$

    if ($minTime_k \leq minTime$) then {
      $minTime = minTime_k$; $minDim = \beta_k$; }
  }
  return ($\lambda = \delta_{minDim}(R_i), \rho = \delta_{minDim}(R_j)$);


/* Communication Edge Calculation Algorithm */
$\forall s_i, \forall s_j, i \neq j \in Stages, \forall A \in Arrays$,
$\forall R_i \in RDAD_{w,s_i}(A), R_j \in RDAD_{r,s_j}(A)$,

/* check if definition of $A$ in $R_i$ reaches $R_j$ */
if ($\omega(R_i) \cap \gamma(R_j) \neq \emptyset$) then {
  /* calculate communication placement */
  $\beta = calcValidGran(R_i, R_j)$;
  $\{\lambda, \rho\} = calcCommPlace(\beta, R_i, R_j)$);

  /* calculate communicated data */
  $\alpha = \alpha(R_{i,\lambda}) \cap \alpha(R_{j,\rho})$;
}
```

Figure 3: Communication Edge Calculation

The function $findCommPlace(\beta, R_i, R_j)$ returns the communication placement for the send and receive pipeline stages. The first step in this function is to identify the granularity which yields the minimum execution time. The time input to this calculation includes the Monet^{TM} synthesis estimates for individual pipeline stage execution, the known communication time and the time during which computation overlap occurs for a pair of stages at a particular communication granularity. If β_k is a set, we calculate one execution time for the whole set since communication for each entry would be mapped to the same placement points as discussed below.

From the communication granularity, the communication placement is determined by mapping the array dimension from $minDim$ to its associated dominant induction variable in each stage. The communication will occur inside the loop corresponding to the selected dominant induction variable. Based on the array sections accessed in $R_{i,\lambda}$ and $R_{j,\rho}$, subsets of R_i and R_j, respectively, accessed within the pipeline stages, we may also need to perform code transformations, such as peeling, to align the communication. For a β_k that is a set, each dimension is mapped to the same dominant induction variable and thus the same communication placement points.

We also address multiple definitions reaching a single read access. If the definitions are generated within the same pipeline stage, we place the send primitives at the control flow meet point just after the definition points. If multiple definitions are generated in different stages, we place additional logic at the receive point, where the control flow meet point occurs for these definitions. Finally, we calculate the intersection of $\alpha(R_{i,\lambda})$ and $\alpha(R_{j,\rho})$. Intersection on array sections is defined as merging the sets of integer linear inequalities and if a solution exists, simplifying the linear inequality set [3].

3.5 Code Generation

Once the compiler has inserted the communication primitives, the SUIF code is translated into behavioral VHDL, shown in Figure 1(d). For presentation, some of the synchronization details have been abstracted away.

4. EXPERIMENTAL RESULTS

4.1 Implementation Status and Methodology

In this experiment, we evaluate our communication analyses by comparing four different communication schemes. The vector $\beta = \langle 1, 2 \rangle$ indicates that both a row-sized or an element-sized transmission are valid for MVIS, and we can always communicate the whole array, either on or off-chip, in one transmission. In *Array Off-chip*, s_1 completes its total calculation and then communicates the array *peak* to s_2 via external memory. There is no pipelining in this scheme, and additional overhead for the memory accesses; reads take 7 ns and writes 0.1 ns but are pipelined so that there is one memory access per clock cycle. The scheme *Array On-chip* is similar, except the array is communicated on-chip. In *Row*, once s_1 has produced a row of *peak*, the row is communicated immediately to s_2. S_2 consumes the data, such that this computation is overlapped with s_1's computation of the next row. Similarly, in *Element*, s_1 communicates each element as it is produced. There is maximum computation overlap in *Element*. For each instance of on-chip communication, independent of amount of data communicated, *commTime* is approximately two cycles. The CEDs for these schemes are shown in Figure 1(c).

For each scheme, we compile the behavioral VHDL with Monet^{TM} and simulate in ModelSim^{TM} to obtain the total execution time. All behavioral VHDL with the exception of the *Array Off-chip* scheme were generated automatically.

4.2 Results

Figure 4 shows the MVIS execution times for four communication schemes. As expected, when accessing external memory, with no computation overlap, as in *Array Off-chip*, we see the largest program execution time. When we compare the *Array On-chip* and *Row*, we see that there is a 1.76 speedup due to the computation overlap gained from stages s_1 and s_2 executing in parallel. The overhead for communicating one element at a time, in *Element*, is not amortized over the computational overlap. The *Element* scheme therefore does not yield a minimal execution time.

For MVIS, we choose $minDim = 1$, since a row-sized transmission, described by

$$\alpha(CED_{s_1 \rightarrow s_2}(peak)) = \left| \begin{array}{c} d1 = x \\ 0 \leq d2 \leq 29 \end{array} \right|,$$

yields a minimal execution time. We calculate the corresponding communication placement points, $\lambda = \rho = x$. The send primitives are placed inside the enclosing x loop body, just after loop y in s_i and the receive primitives are placed inside the enclosing x loop body, just before loop y in s_j. The CED for the Best Communication is shown in Figure 1(c) and the communication points are shown in (d).

5. RELATED WORK

Previous work on array data flow analysis [6, 14, 3] focused on data dependence analysis but not at the level of

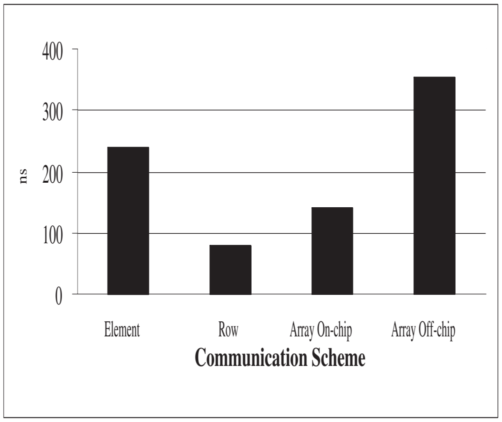

Figure 4: Communication Results

precision required to derive communication requirements for our platform. Parallelizing compiler communication analysis techniques [4, 12] exploited data parallelism.

In [5] Arnold created a software environment to program a set of FPGAs connected to a workstation. [9] focused on compiling for a tightly coupled hybrid FPGA and RISC architecture; Callahan and Wawrzynek [7] used a VLIW-like compilation scheme for the GARP project; both works exploit intra-loop pipelined execution techniques. Goldstein *et al.* [10] describes a custom device that implements an execution-time reconfigurable fabric. Weihardt and Luk [15] describes a set of program transformations to map the pipelined execution of loops with loop-carried dependences onto custom machines.

The approach taken in this paper differs from previously mentioned efforts. Our approach takes un-annotated sequential programs and maps them into a pipelined execution scheme without programmer intervention. Unlike concurrent languages, our approach neither relies on nor exploits concurrent specification behavior. 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.

6. 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, a machine vision application. We presented experimental results, derived automatically by our system. We illustrate how these analyses can improve application performance, as evidenced by the 1.76 speedup we gain over a non-pipelined implementation. Current work focuses on integrating these analyses with automated design space exploration for a single loop [13] and partitioning pipelined implementations over multiple FPGAs.

7. REFERENCES

[1] The Stanford SUIF compilation system. Public Domain Software and Documentation, http://suif.stanford.edu.

[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] 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.

[7] 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.

[8] 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*, Lecture Notes in Computer Science. Springer-Verlag, 2001.

[9] M. Gokhale and J. Stone. Napa C: compiling for a hybrid RISC/FPGA architecture. In *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, pages 126–135, 1998.

[10] 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.

[11] 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.

[12] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Preliminary experiences with the FortranD compiler. In *Proc. Seventh Intl. Conf. Supercomputing*, Portland, Nov 1993.

[13] 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.

[14] 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.

[15] M. Weinhardt and W. Luk. Pipelined vectorization for reconfigurable systems. In *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, pages 52–62, 1999.

[16] 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.