

Coarse-Grain Pipelining on Multiple FPGA Architectures*

Heidi Ziegler**, Byoungro So, Mary Hall, Pedro C. Diniz

University of Southern California / Information Sciences Institute

4676 Admiralty Way, Suite 1001

Marina del Rey, California 90292

{ziegler, bso, mhall, pedro}@isi.edu

Abstract

Reconfigurable systems, and in particular, FPGA-based custom computing machines, offer a unique opportunity to define application-specific architectures. These architectures offer performance advantages for application domains such as image processing, where the use of customized pipelines exploits the inherent coarse-grain parallelism. In this paper we describe a set of program analyses and an implementation that map a sequential and un-annotated C program into a pipelined implementation running on a set of FPGAs, each with multiple external memories. Based on well-known parallel computing analysis techniques, our algorithms perform unrolling for operator parallelization, reuse and data layout for memory parallelization and precise communication analysis. We extend these techniques for FPGA-based systems to automatically partition the application data and computation into custom pipeline stages, taking into account the available FPGA and interconnect resources. We illustrate the analysis components by way of an example, a machine vision program. We present the algorithm results, derived with minimal manual intervention, which demonstrate the potential of this approach for automatically deriving pipelined designs from high-level sequential specifications.

Keywords:

Coarse-grain Pipelining, FPGA-based Custom Computing Machines, Parallelizing Compiler Analysis Techniques.

1. Introduction

The implementation of pipelined execution techniques is an effective method to improve the throughput of a given computing architecture. By dividing the set of operations to be executed into subsets or pipe stages, pipelining increases the number of operations that may execute simultaneously, thus exploiting the implicit parallelism in the sequential application and effectively using available resources. The overall performance improvements come from higher throughput in spite of the fact that the execution time for each individual operation remains unaltered.

In a traditional or synchronous pipeline, the pipe stages are defined such that they contain equal amounts of computation in order to avoid idle processing time that might occur in an unbalanced system. When pipe stages are not easily balanced, due to widely varying

types of operations that make up the computation, an asynchronous pipeline is employed. As a result, the flow of data between neighbors is controlled by a handshaking protocol that triggers data availability.

The asynchronous pipeline provides an ideal processing model on which digital image processing [22] applications execute efficiently. Typically, these applications process multiple images using simple image operators. Examples of common image processing operators include a wide range of stencil operators (*e.g.*, over a fixed $N \times N$ window) or simple thresholding and offsetting computations. The typical application has multiple loop nests inside a main loop for iterating over all data commonly implemented as multi-dimensional arrays.

FPGA-based computing machines offer a unique opportunity for the design of custom pipelining structures matched to each application. One or several loop bodies can be synthesized on each of the FPGAs. In addition, the layout of the stages and their connectivity can be designed to match the application requirements in terms of the relative consumer/producer rates each stage exhibits. Internal FPGA register resources and direct wires can be used to establish high-performance inter-stage communication, avoiding excessive buffer read/write and synchronization operations and thereby increasing overall throughput.

The complexity and sophistication of data orchestration and control of pipelined execution make automatic tools that can analyze sequential applications and derive pipelined implementations extremely desirable. Fortunately, the domain of digital image processing and graphics are a perfect match for existing parallelizing compiler analysis techniques. Using these techniques a compiler and synthesis system can analyze the input sequential code and partition its data and computation among multiple FPGAs for pipelined execution. The compiler analyzes the set of pipeline stages and schedules them onto the target architecture respecting the original program data dependences and the target architecture's FPGA and memory capacity constraints. In so doing, the compiler analysis can derive communication requirements between pipeline stages.

In this paper we describe a set of compiler analyses that address the issues in automatically mapping computations expressed in high-level sequential languages such as C, directly to FPGA-based computing architectures. In particular it makes the following specific contributions:

- It describes an implementation of several parallelizing compiler analysis techniques and transformations required to automatically design platform and application specific pipelines, which have been extended to map computations onto FPGA-based architectures.

* Funded by the Defense Advanced Research Project Agency under contract number F30603-98-2-0113.

** Funded by a Boeing Satellite Systems Doctoral Scholars Fellowship.

- It describes an approach to integrate these various techniques that will allow the compiler to reason about the mapping of computation and automatically derive the corresponding communication and synchronization.
- It presents experimental results for a vision application, demonstrating the use of these techniques for a particular FPGA-based architecture. The pipeline stages are optimized, using estimation techniques to guide the application of loop unrolling to match the producer/consumer rates of the stages while controlling the increase in size of the mapping of stages to FPGAs.

With the growing number of available transistors on a single die, we anticipate the emergence of reconfigurable computing architectures with the ability to incorporate (through soft-cores) various coarse-grain computing elements such as microprocessor cores, and application-specific-engines (ASEs). Enabling the implementation of pipelined execution and the corresponding management of data across many of these computing cores will become an increasingly important issue. The analyses presented in this paper will ultimately allow the automated application mapping for these emerging infrastructures.

The paper is organized as follows. In the section 2 we describe the basic characteristics of the target configurable architecture as well as the terminology for the mapping problem addressed in this paper. In this section we also describe, the mapping solution that the compiler has found for the presented example. Section 3 describes the compiler analysis in detail. We present the results of the generated compiler analysis in section 4 for a set of image processing kernels. In section 5 we survey related work and conclude in section 6.

2. Motivation and Background

We now describe the mapping problem addressed in this paper. We then describe the generic characteristics of the target reconfigurable computing architecture. Finally, we present the mapping of a sample application, inspired by a real life computer vision program.

2.1 Problem Statement

The problem we address in this paper is automatically mapping an application onto an asynchronous pipeline, executing on a configurable architecture as described in section 2.3. We want to exploit the parallelism in the application without the use of any programmer inserted pragmas or directives and also tailor the FPGA configurable logic blocks, by designing an asynchronous pipeline that minimizes the application execution time. Formally, an asynchronous linear pipeline is a set of pipe stages, which performs a fixed function over a stream of data flowing from the first pipe stage to the last, in a linear progression. It contains k pipe stages, where external inputs are fed into the pipeline at stage S_j . The results from a given stage S_i are routed from S_i to S_{i+1} , for all $i = 1, 2, \dots, (k-1)$. The result of the pipelined computation is found at the output of stage S_k .

Mapping a pipelined application across multiple computing elements and memories involves identifying a set of communicating pipeline stages. Each stage must be mapped and scheduled to execute when its data is available, *i.e.*, the stage input data has already been produced and there is enough storage to deposit the resulting stage output. In a pipelined execution scheme, our compilation strategy is

to identify pipe stages, to assign one or more pipe stages to a given computing element and to allocate storage space in the associated storage elements for the input and output data. The compilation goal is to minimize overall completion time, while meeting the storage and computing element capacity constraints of the system.

2.2 Application Domain

The compiler approach described in this paper, although generic in the sense of mapping a set of communicating pipe stages to a reconfigurable architecture with multiple FPGAs, has focused on applications whose computations are specified by sequences of loop nests. These loop nests, not necessarily perfectly nested, compute over array data structures with known dimensions, affine index access functions and constant loop bounds. For the current implementation, we have eliminated the presence of pointers. Despite these restrictions, we have been able to easily express simple digital image and signal processing kernels and other regular array computations of interest to develop our compiler analysis approach.

2.3 Configurable Architectures

We target a reconfigurable computing architecture organized as shown in Figure 1. Each computing element is implemented as a field programmable gate array (FPGA), represented by a square in the figure; each storage element, an external memory or set of memories, connected to one or more FPGAs, represented by a rectangle in the figure. Each computing element may be viewed as a single pipeline stage; each storage element, the connection between two adjacent stages. Alternatively, each FPGA may contain multiple pipe stages. Here registers, internal to the FPGA, are the stage connectors, as it is no longer necessary to route data off-chip and then into the next stage. Circles within each FPGA represent pipe stages and rectangles represent the internal storage on an FPGA.

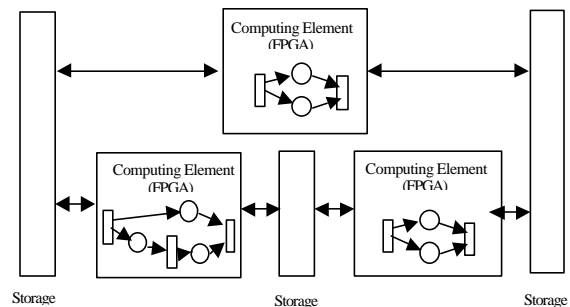


Figure 1. Generic Configurable Architecture.

2.4 Example

We now illustrate the mapping of a sample program. The computation is inspired by a submersible vehicle vision application [16] and is depicted in Figure 2.

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 another control loop (not shown in the figure) that processes a sequence of image frames. The first loop nest extracts image features using the *Sobel* operator. 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.

To map this computation to the configurable architecture described in Figure 1 an FPGA design is synthesized for the digital image operators defined in each of the loop nests. In the following, pipeline stage $S1$ corresponds to the computation in the first loop nest, stage $S2$ to the second and so forth. Figure 3(a) depicts the source code augmented with synchronization primitives to coordinate the pipelined execution between stages. To optimize the individual stages, the compiler applies a set of transformations to exploit operator and memory parallelism and eliminate unnecessary array accesses, as illustrated in Figure 3(b).

The mapping, shown in Figure 3(c), of computation and data to the target platform, illustrates the placement of each pipe stage computation, the data and the renaming of variables. We have used eight logical memories associated with each of the two FPGAs. In practice these logical memories would be mapped to the four physical memories attached to each FPGA. Stage $S1$ generates the values in the $peak$ variable. Stage $S2$ consumes $peak$ and produces $feature_x$ and $feature_y$. The compiler partitions the $peak$ array into four sections and maps each to four internal memories. Next, the compiler maps stage $S3$ to another FPGA. In this scenario, the compiler maps the arrays v , $feature_x$, and $feature_y$ again to four distinct memories. The next section describes the analysis that led to this mapping result.

3. Compilation System Overview

The compiler analysis described in this paper augments an automatic parallelization system that is part of the Stanford SUIF compiler [27]. Figure 4 depicts the set of compiler analyses implemented specifically for our DEFECTO system, a design environment for FPGA-based systems [10]. The SUIF code that is the input to this analysis suite contains data dependence information and also information about each loop nest defined as a unique pipe stage. This code is then analyzed as follows:

- Loop Unrolling Analysis: Determines which loop or loops should be unrolled to expose more memory and operator parallelism while meeting each FPGA capacity constraint.
- Data Reuse Analysis: Determines which data references can be reused across loop iterations and within a loop body.
- Data Layout Analysis: Determines how the data can be laid out in memory to expose more memory access parallelism.
- Communication Analysis: Determines which sections of which array variables need to be communicated. Calculates array access orders, inserts communication and synchronization between pipe stages.
- Pipelining Analysis: Determines pipe stages and matches the production and consumption data rates. Performs on- and off-chip storage management.

A successful compilation and synthesis system must integrate all of the above analyses in a coherent fashion in addition to interfacing with estimation and synthesis tools. The analyses outlined above interact with each other while each considers different trade-offs.

For example, the data reuse analysis attempts to reduce the number of memory accesses by reusing data already fetched from memory into registers. This decreases the time spent on memory accesses at the expense of more internal storage resources. Also, the loop unrolling analysis increases the amount of memory parallelism by exposing more operators. This code expansion can cause the implementation to require a substantially increased percentage of the FPGA capacity.

```

#define IMAGE_SIZE 66
int u[IMAGE_SIZE][IMAGE_SIZE];
int v[IMAGE_SIZE][IMAGE_SIZE];
int ssd[IMAGE_SIZE][IMAGE_SIZE];
int features_x[IMAGE_SIZE][IMAGE_SIZE];
int features_y[IMAGE_SIZE][IMAGE_SIZE];

// Stage 1. Extract features with SOBEL
for(x = 0; x < IMAGE_SIZE-2; x++){
  for(y = 0; y < IMAGE_SIZE-2; y++){
    // u is read only
    peak[x][y] = sobel(u[x][y], u[x+1][y],
                      u[x+2][y], u[x+1][y], u[x+1][y+2],
                      u[x+2][y+1], u[x+2][y+2]);
  }
}

// Stage 2. Select features above threshold
for(x=0; x < IMAGE_SIZE-2; x++){
  for(y=0; y < IMAGE_SIZE-2; y++){
    if(peak[x][y] > threshold){
      features_x[x][y] = x; features_y[x][y] = y;
    } else {
      features_x[x][y] = 0; features_y[x][y] = 0;
    }
  }
}

// Stage 3. Compute Distance Across Images
for(i = 0; i < IMAGE_SIZE-2; i++) {
  for(j=0; j < IMAGE_SIZE-2; j++) {
    ssd[i][j] = 0;
    if((features_x[i][j] != 0){
      ssd[i][j] =
        (u[i][j]-v[i][j])*(u[i][j]-v[i][j]) +
        (u[i][j]-v[i][j+1])*(u[i][j]-v[i][j+1]) +
        (u[i][j]-v[i][j+2])*(u[i][j]-v[i][j+2]) +
        (u[i][j]-v[i+1][j])*(u[i][j]-v[i+1][j]) +
        (u[i][j]-v[i+1][j+1])*(u[i][j]-v[i+1][j+1]) +
        (u[i][j]-v[i+1][j+2])*(u[i][j]-v[i+1][j+2]) +
        (u[i][j]-v[i+2][j])*(u[i][j]-v[i+2][j]) +
        (u[i][j]-v[i+2][j+1])*(u[i][j]-v[i+2][j+1]) +
        (u[i][j]-v[i+2][j+2])*(u[i][j]-v[i+2][j+2]);
    }
  }
}

```

Figure 2. Machine Vision Code Structure

```

signal(host);

// Stage 1. Extract features with SOBEL
for(x= 0; x < IMAGE_SIZE-2; x++){
  for(y= 0; y < IMAGE_SIZE-2; y++){
    // u is read only
    peak[x][y] = ...;
    write(peak[x][y]);
  }
}

// Stage 2. Select features above threshold
for(x=0; x < IMAGE_SIZE-2; x++){
  for(y=0; y < IMAGE_SIZE-2; y++){
    read(peak[x][y]);
    if(peak[x][y] < threshold){
      features_x[x][y] = ...;
      features_y[x][y] = ...;
    } else {
      features_x[x][y] = ...;
      features_y[x][y] = ...;
    }
    send(features_x[x][y]);
    send(features_y[x][y]);
  }
}

// Stage 3. Compute Distance Across Images
for(i = 0; i < IMAGE_SIZE-2; i++) {
  for(j=0; j < IMAGE_SIZE-2; j++) {
    receive(features_x[x][y]);
    receive(features_y[x][y]);
    ssd[i][j] = 0;
    if((features_x[i][j] != 0) &&
       (features_y[i][j] != 0)) {
      ssd[i][j] = ...;
    }
    send(ssd[i][j]);
  }
}
receive(host);

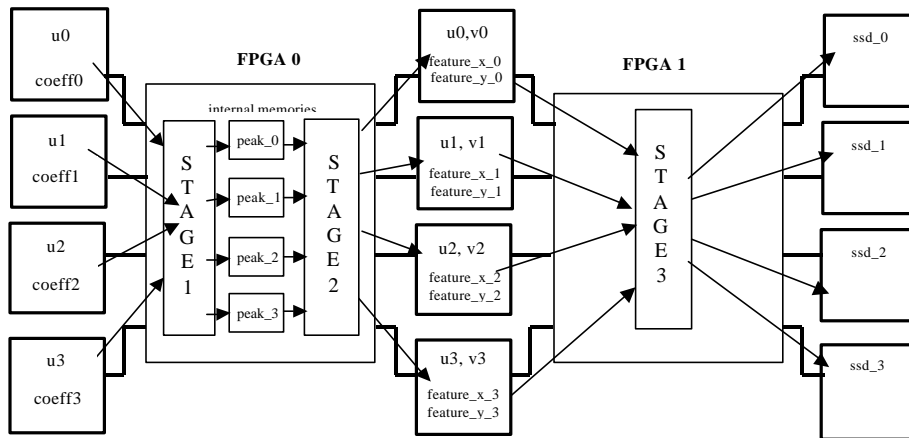
int U0[66][17],U1[66][17],U2[66][17],U3[66][17];
int V0[66][17],V1[66][17],V2[66][17],V3[66][17];
int SSD0[66][17],SSD1[66][17],SSD2[66][17],SSD3[66][17];
int FEATURE_X0[66][17], FEATURE_X1[66][17];
int FEATURE_X2[66][17], FEATURE_X3[66][17];

/* initialize registers
v_0_33,v_1_33,v_2_32,v_0_17,v_1_17,v_2_16,v_3_32,
v_3_16,v_0_32,v_0_16,v_1_32,v_1_32 */
for (i = 0; i <= 63; i++) {
  /* initialize registers v_0_1,v_1_1 */
  for (j = 0; j <= 15; j++) { /* unrolled by 4 */
    /* unroll section 0 */
    ssd_0_0 = 0;
    u_0_0 = U0[i][1+j]; v_2_0 = V2[2+i][1+j];
    if (FEATURE_X0[i][1+j] != 0)
      ssd_0_0 = (u_0_0 - v_0_33)*(u_0_0 - v_0_33) +
        (u_0_0 - v_1_33)*(u_0_0 - v_1_33) +
        (u_0_0 - v_2_32)*(u_0_0 - v_2_32) +
        (u_0_0 - v_0_17)*(u_0_0 - v_0_17) +
        (u_0_0 - v_1_17)*(u_0_0 - v_1_17) +
        (u_0_0 - v_2_16)*(u_0_0 - v_2_16) +
        (u_0_0 - v_0_1) *(u_0_0 - v_0_1) +
        (u_0_0 - v_1_1) *(u_0_0 - v_1_1) +
        (u_0_0 - v_2_0) *(u_0_0 - v_2_0);
    SSD0[i][1+j] = ssd_0_0;
    /* unroll section 1 */
    .....
    /* unroll section 2 */
    .....
    /* unroll section 3 */
    ssd_3_0 = 0;
    u_1_0 = U3[i][1+j]; v_1_0 = V1[2+i][2+j];
    if (FEATURE_X3[i][1+j] != 0)
      ssd_3_0 =
        (u_1_0 - v_3_32)*(u_1_0 - v_3_32) +
        (u_1_0 - v_0_32)*(u_1_0 - v_0_32) +
        (u_1_0 - v_1_32)*(u_1_0 - v_1_32) +
        (u_1_0 - v_3_16)*(u_1_0 - v_3_16) +
        (u_1_0 - v_0_16)*(u_1_0 - v_0_16) +
        (u_1_0 - v_1_16)*(u_1_0 - v_1_16) +
        (u_1_0 - v_3_0)*(u_1_0 - v_3_0) +
        (u_1_0 - v_0_0)*(u_1_0 - v_0_0) +
        (u_1_0 - v_1_0)*(u_1_0 - v_1_0);
    SSD3[i][1+j] = ssd_3_0;
    shift_registers(v_0_0, ..., v_0_33);
    shift_registers(v_1_0, ..., v_1_33);
    shift_registers(v_2_0, ..., v_2_32);
    shift_registers(v_3_0, ..., v_3_32);
  } /* end of for j */
} /* end of for i */

```

(a) Application Mapping to a Two-FPGA Architecture

(b) Optimized Loop Nest in Stage S3



(c) Pipelined Mapping to Target Architecture

Figure 3. Example Transformations and Mapping

3.1 Reuse Analysis and Scalar Replacement

Scalar replacement replaces array references by accesses to temporary scalar variables, so that high-level synthesis will exploit reuse in registers [6]. Our approach to scalar replacement closely matches previous work, which eliminates true dependences when reuse is carried by the innermost loop, for accesses in the *affine* domain with consistent dependences *i.e.*, constant dependence distances. There are, however, two differences: (1) we eliminate unnecessary memory writes and, (2) we exploit reuse across all loops in the nest. The latter difference stems from the observation that many, though not all, algorithms mapped to FPGAs have sufficiently small loop bounds or small reuse distances, and the number of registers that can be configured on an FPGA is sufficiently large, that reuse across multiple loops can be supported. A more detailed description of our scalar replacement and register reuse analysis can be found in [10].

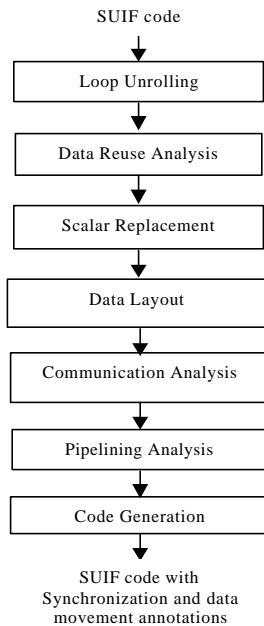


Figure 4. Compilation Flow using SUIF Infrastructure.

Our implementation of scalar replacement includes loop peeling since registers are not yet initialized at the start of the loop execution. Loop peeling removes the first (or last) several iterations of the loop body and makes them separate code before (or after) loop execution. The first iteration of each loop is peeled in order to initialize registers (represented as a comment in Figure 3(b)). For brevity, loop peeling is not shown in this example. In place of the 18 $u[i][j]$ accesses in the first unroll section, each element of array u is loaded into a register, u_0_0 , once, and then that value is reused 18 times in each iteration of the inner loop. Our analysis also exploits data reuse on array v across loops i and j through register shifting. This is shown by the primitive *shift_registers* in the figure. This primitive is meant to illustrate the shifting of each register value to its neighbor, leaving one register empty, to be filled by a memory read on the next iteration. Direct memory accesses to array ssd are

replaced with scalar variables, ssd_0_0 etc., in order to avoid redundant memory write accesses.

3.2 Loop Unrolling

Unrolling [20] is widely used in the computation mapping to expose fine-grain operator and memory parallelism by replicating the logical operations and their corresponding operands in the loop body. 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. Through compiler directed loop unrolling, computations across several iterations can also be scheduled in the same cycle because all the necessary data are ready at the same time. Therefore, unrolling increases the rate at which data is fetched from memory as well as the rate at which data is consumed by the computation. Figure 3(c) shows the unrolled code in which the inner loop j is unrolled three times. Each section is commented with “unroll section”. Only unroll sections zero and three are shown for brevity.

3.3 Custom Data Layout and Array Renaming

Another code transformation lays out the data in the FPGA’s external or internal memories, so as to maximize memory and operator parallelism. Based on the number of available memories for each stage, and the amount of unrolling that has occurred on a given array read or write statement, the arrays are subdivided into a set of smaller arrays. Array elements are distributed cyclically among this set. The code is transformed to reflect the accesses to the distinctly named arrays, as shown in Figure 3(b). Each newly created array is uniquely assigned to a memory. For the example, all the arrays were subdivided into four smaller arrays, each one a quarter the size of the original, since each FPGA has either four internal or external memories and the statements that accessed those memories were unrolled by at least a factor of four. For the interface between stages $S1$ and $S2$, a set of four internal memories was implemented; for the interface between stages $S2$ and $S3$, array elements are distributed cyclically in one dimension across four external memories.

This approach is similar in spirit to the modulo unrolling used in the RAW compiler [4]. However, as compared to modulo unrolling, which is a loop transformation that assumes a fixed data layout, our approach is a data transformation. Further, our current implementation supports a more varied set of custom data layouts. A typical layout is cyclic in at least one dimension of an array, possibly more, but can be block cyclic in the presence of packing and strided accesses. Further discussion of the data layout analysis and transformation can be found in [10].

3.4 Communication and Pipeline Analysis

We perform communication analysis in order to manage the flow of data among the computations on the FPGAs as well as the host. We first calculate which data must be communicated between each pair of pipeline stages. We use array data flow analysis to find arrays accessed within a particular region of the program, capturing precise array elements read and written within each stage [25]. Based on reaching definition information, we intersect the data flow analysis results for each pair of stages that contain associated definition and use points. From this, we determine a partial ordering on the execution and the exact data that must be communicated between stages.

Figure 3(a) illustrates the results of the communication analysis after the compiler has identified the computation to be executed as part of each pipeline stage and to which FPGA these computations should be mapped. In this figure we have used the *read* and *write* primitives for communication that occurs inside the same FPGA whereas we used *send* and *receive* for inter-FPGA data communication. These primitives include synchronization necessary to avoid accessing memory out of order. Notice that the analysis is capable of deriving various levels of granularity of the communication between stages. In the example, we have chosen to insert communication primitives at the finest granularity of single array elements because the overhead of small messages is not significant in our system. Fine-grained communication also maximizes the overlap of the pipeline stages. As communication overhead increases, the same analysis can be used to trade-off communication granularity for parallelism.

We can enhance the performance of the architecture by creating a pipelined execution scheme. Unlike traditional architectures, we must create the pipeline datapath and map the communication on to it. We use the characteristics of the communication to drive the datapath definition. We examine the array access patterns and compare those of the producer and consumer pipeline stages to see if they are the same [3]. This ordering allows us to determine the exact communication granularity. We then determine the communication points, the send point for the producer and the receive point for the consumer, and insert annotations, that will be processed by a later compiler pass. We communicate the produced data as soon as possible in order to match the consumption rate of the next stage. For arrays that are produced and consumed according to a common traversal order, communication occurs just after the producer instruction; when the traversal orders do not match, the communication occurs only after the complete dataset has been produced, for example, at the termination of the producer pipe stage.

Once we decide the computation and data partition, as discussed in the next section, we determine whether data will be communicated via the shared memory or an on-chip buffer. We assume data to be communicated between stages, executing on different FPGAs are written back to a shared memory by the producing FPGA and then read from memory by the consuming FPGA. An alternate implementation, to be investigated in future work, would be the use of direct links between communicating FPGAs. In Figure 3(c), arrays *feature_x* and *feature_y* are examples of data that are communicated via a shared memory, from stages *S2* to *S3*. For data flowing between stages executing on the same FPGA, local, on-chip buffers are used to store the produced data until it can be consumed. The communication of array *peak*, from stages *S1* to *S2*, is an example of this form of communication. Synchronization among the involved pipe stages is required for both external and internal memory usage. Figure 3(a) shows the insertion of the read/write primitives for on-chip and send/receive for off-chip communication. We used the original code in this figure for brevity; in the implementation, the communication points are inserted in the optimized code shown in Figure 3(b).

3.5 Computation and Data Partitioning

In this phase, the compiler partitions both the data and computation across FPGAs and memories while taking into account the FPGA and memory capacity constraints as well as pipeline performance.

Generated under the assumption that all FPGA capacity was available to an individual stage, we use the performance results to identify the loop stage likely to become a bottleneck in the pipeline. We then adjust the loop unroll factor for other stages to match the producer/consumer rates across the entire pipeline. Using the revised relative sizes, we then apply a greedy algorithm to map the stages to FPGAs. For this example, stage *S3* has the largest space requirement after being unrolled further to match its consumption rate to the production rate of *S2*. It is therefore assigned to FPGA1 while *S1* and *S2* are assigned to FPGA0. Data associated with each stage must be partitioned to memories available to each corresponding FPGA. We assume the program working set fits well within the memory capacity constraints. Read only data, such as array *u*, is replicated to decrease the amount of data that is communicated between stages.

3.6 Code Generation

Once the compiler has determined what the source code for each of the pipe stages should be, it generates a set of pipe stage definitions, along with the required data that needs to be communicated between stages and writes this information out as annotations. Using these annotations, each stage is translated into behavioral VHDL using a tool we developed in the context of the DEFACTO project [10]. The communication, occurring between each pipe stage, is translated into corresponding host functions that copy data to and from the corresponding memories, if applicable, along with synchronization signals to each FPGA indicating when it is safe to initiate execution. We rely on a wealth of infrastructure for supporting local memory operations [8] by inserting additional library calls as needed and we then use a commercial synthesis flow to generate the FPGA bit-stream file.

4. Experimental Results

4.1 Implementation Status and Methodology

We have implemented the analyses described in the previous section (with the exception of the computation and data partitioning discussed in section 3.5) as part of the DEFACTO system. The DEFACTO system combines parallelizing compiler technology from the Stanford SUIF [27] compiler with commercial synthesis tools to automate design space exploration in mapping sequential standard C code to FPGA-based systems. Analyses and transformations are applied within the SUIF system, and the code is partitioned between what is to execute on a conventional microprocessor and what is to be implemented in hardware on one or more FPGAs.

The code to be mapped to FPGAs is translated into behavioral VHDL. Code to run on the microprocessor is translated into C, and includes calls to initialize data, retrieve results and initiate computation on the FPGAs. Next we use Mentor Graphics' MonetTM [19] behavioral synthesis tool to generate RTL VHDL, followed by logic synthesis using Mentor Graphics' LeonardoSpectrumTM [18] tool, and subsequently Xilinx Foundation tools for the place-and-route phase.

Automated design space exploration (for example, to derive the optimal unroll factor) is accelerated 1000 times by using area and speed estimates from MonetTM rather than fully synthesizing each design under consideration [10]. We define a metric, *Balance* [26], to guide the design space exploration. *Balance* for a particular loop nest is defined by the ratio of the data fetch rate, or effective

bandwidth to and from memory, to the data consumption rate, or the total computational delay in one iteration of a loop.

If *Balance* is close to one, which is the ideal case, both memories and FPGAs are busy. If *Balance* is less than one, the loop nest needs data at a higher rate than the architecture can provide. This kind of loop is memory bound, and the on-chip space for computation logic may be too excessive to be useful. On the other hand, if *Balance* is greater than one, the loop is compute bound, and the space for on-chip data storage may be unnecessarily used.

Assuming computation and memory access can be overlapped, *Balance* permits the design space exploration to limit unroll factors. For example, if a design is memory bound and memory bandwidth is fully utilized, there will be little or no performance advantage to increasing the unroll factor, and in fact, the increase in design complexity might degrade the achievable clock rate.

To validate this approach, in previous work [10] we have compared the Monet™ behavioral VHDL synthesis estimates with the output of RTL VHDL synthesis and place-and-route. In all cases, the total number of clock cycles remains the same. However, the target clock rate can degrade for larger unroll factors due to increased routing complexity. Similarly, space can also increase, more than linearly with the unroll factors. For a set of application kernels, these estimation discrepancies, while not negligible, never influenced the selected design. This is because our algorithm favors small unroll factors where there are significant increases in parallelism; even with small degradations in clock rate, increased unrolling yields overall performance gains.

In previous work, we have presented algorithms for mapping individual loop nests onto a single FPGA with one or more external memories [10], [25]. For this experiment, we extend our previous work with an implementation of communication and pipelining analysis to derive a mapping solution for multiple FPGAs. For several individual loop nests, we have demonstrated fully automated design mapping, from C specification to execution on the Annapolis Wildstar™ board [1].

The following set of results is derived automatically for each of the three pipeline stages from the example in Section 2. Each pipeline stage has access to four memories, and the data for each stage is automatically mapped to these memories to exploit memory parallelization. For the purposes of this experiment, we assume that communication and memory access throughput are the same: one cycle for either a read or write. This optimistically assumes full pipelining of memory accesses, which is possible but not guaranteed on the Annapolis Wildstar™.

Our implementation of communication and pipelining analysis automatically derived the communication shown in Figure 3(a), including identifying the placement of communication to enable pipelining between stages. We have not yet implemented computation and data partitioning as described in Section 3.5, but we plan to generalize the solution for the example presented here.

4.2 Results

The following tables show behavioral synthesis estimates (augmented by compiler models) of the compiler-optimized pipeline stages for each of the three stages from the vision application in Figure 2. These results were derived automatically by our system. Tables 1 through 3 contain three rows. The first row shows the

Balance achieved by the given solution. The second row provides the number of clock steps on a Xilinx Virtex™ 1000-BG560 FPGA [29]. The third row provides an estimate of space; this number does not directly correspond to configurable logic blocks on the Virtex; in our experience with place-and-route following Monet™ estimation, we have determined that a number above about 32000 exceeds the capacity of the Virtex. The results are presented for different unroll factors of the innermost loop for these three stages. Note that in Tables 1 and 3, an unroll factor of 32 represents full unrolling due to loop peeling used to initialize registers for scalar replacement.

Table 1. Stage S1 Results for Different Unroll Factors

Unroll Factor	1	2	4	8	16	32
Balance	1.50	1.50	1.50	1.25	1.13	0.95
Cycles	12344	6297	3177	2653	2385	2554
Space	12470	9745	15619	25583	45889	79928

Table 2. Stage S2 Results for Different Unroll Factors

Unroll Factor	1	2	4	8	16	32	64
Balance	1.00	1.00	1.00	0.83	0.75	0.71	0.69
Cycles	12352	6208	3136	3136	3136	3136	3136
Space	249	327	384	532	717	1116	1470

Table 3. Stage S3 Results for Different Unroll Factors

Unroll Factor	1	2	4	8	16	32
Balance	2.00	1.67	2.00	1.63	1.44	1.34
Cycles	16632	10266	8229	6699	5919	5420
Space	16817	31954	45912	83340	132656	262054

The columns that are in bold face type represent the desired unroll factor. To see how we arrived at this result, let us first consider the optimal unroll factors selected for each individual stage. Without unrolling, stage *S1* is compute bound. The best solution for stage *S1* in isolation is the balanced solution obtained with an unroll factor of 16. Stage *S2* without unrolling is balanced, but performance improves up to the unroll factor of four, where the memory bandwidth is fully utilized, which we call the *saturation point*. Beyond the saturation point, the design is memory bound, and performance does not improve. In stage *S3*, the design is compute bound because, as shown in Figure 3(b), scalar replacement has eliminated most of the memory accesses. By fully unrolling the loop nest, the design is nearly balanced.

To arrive at the final solution, we make several observations:

- Stage *S3* is the slowest stage, so it is possible to reduce unroll factors for stages *S1* and *S2* without slowing down overall

performance. This leads us to select unroll factors of four for both stages $S1$ and $S3$.

- Stage $S3$ is much larger than the other stages, so it is placed on an FPGA by itself. Stages $S1$ and $S2$ are placed on the same FPGA.
- Accordingly, the v , $feature_x$ and $feature_y$ arrays are placed in memories that may be shared by stages $S2$ and $S3$. Array u is replicated.

Assuming this implementation meets the capacity constraints of the FPGA, we derive the mapping shown in Figure 3(c).

Note that on a Virtex, capacity may be limited for stage $S3$, and an unroll factor of two is the upper limit. In this case, we would also limit the unroll factor for stages $S1$ and $S2$ to a factor of two as well.

We also performed a similar experiment on an IDCT type kernel consisting of two pipeline stages. For each stage, the optimal unroll factor was one, corresponding to a *Balance* of 1.00. Our analysis determines that the access patterns for the communicated arrays are dissimilar. Due to the fact that the arrays must be communicated, we may reorganize the data during communication. We may also further tailor the unroll amounts, and the points and granularity of communication, based on the communication latencies. These analyses are the subject of current [9] and future work.

5. Related Work

Software pipelining has long been used as a way to improve computational throughput and resource utilization [24]. The Splash-2 project [2] created an integrated software environment to program a set of FPGAs connected to a high-end workstation. Unlike our automated system, it incorporated manual programming and partitioning of the application into a pipelined execution scheme.

Gokhale and Stone [13] focused on compiling for a tightly coupled hybrid FPGA and RISC architecture; Callahan and Wawrzynek [5] used a VLIW-like compilation scheme for their GARP project; both works exploit pipelined execution techniques. Unlike the work described in this paper, these efforts have focused on intra-loop pipelining rather than coarse-grain pipelined execution.

PipeRench [15] devices attempt to solve the slow FPGA problem by implementing a fabric that is reconfigurable during execution, using a virtually unlimited number of pipeline stages (stripes). The compiler takes as input, an application written in the dataflow intermediate language, and then uses operator decomposition and recombination to create a pipeline reconfigurable execution scheme. The analysis is fine-grained and the stripe resources are predefined.

Weinhardt and Luk [28] describe a set of program transformations to map the pipelined execution of loops with loop-carried dependences onto custom machines using a pipeline control unit. They focus exclusively on a single FPGA implementation and for a single loop and not on the coarse-grained execution of loop nests. They do not address the issues of automation, or analysis and do not present a compiler algorithm for their implementation.

Other languages address the need to explicitly target pipelined execution. The RaPiD architecture and its associated language, the RaPiD-C [7] language, uses the *sloop* construct to map the different iterations of a loop to the RaPiD architecture nodes – a linear array of functional elements – so that outer loops can be

executed in pipelined fashion. The SpecC [11] language uses the *pipe* keyword to indicate when a sequence of statements, considered in an infinite loop, should be executed in pipelined mode. Variables associated with a *pipe* can also have the *pipelined* attribute to indicate that multiple storage elements should be allocated to them, one for each stage of the pipeline.

The Streams-C [14] high level language follows the Communication Sequential Processes (CSP) parallel programming model. The implementation is a combination of annotations and library function calls. The programmer uses the annotations to declare processes, streams and signals. A process is an independently executing object with a process body that is a C routine and signals are used to synchronize their execution. The programmer also defines the streams and associates a set of input/output ports with each process. The programmer also defines the properties of each data stream and explicitly introduces the appropriate *read* and *write* operations in the code.

The approach taken in this paper differs from the previously mentioned efforts in several respects. First, our approach takes unannotated sequential imperative 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. Second, instead of focusing on intra-loop pipelining techniques that optimize resource utilization, we have focused on coarse-grain inter-loop pipelining, mapping different loop nests to different computational elements. Our focus has been on increased throughput through coarse-grain instruction-level-parallelism, which we believe is a natural match for digital image processing data intensive and streaming applications.

The Cameron research project [23] is a system that compiles programs written in a single-assignment subset of C called SA-C into dataflow graphs and then into synthesizable VHDL. The resulting compiled code has both a component that executes on a traditional processor as well as on computing architectures with FPGA devices. SA-C also defines *reduction* and *windowing* operators for two-dimensional array variables. These operators are directly translated into predefined library routines for the target FPGAs. Like in our approach, the SA-C compiler includes loop-level transformations such as loop unrolling and tiling in particular when windowing operators are present in a loop. However, the application of these loop-level transformations is controlled by *pragmas* rather than using the behavioral synthesis estimation.

In the context of DSP programs, Murthy and Shuvra [21] have addressed data storage for synchronous data flow specifications. They have focused on merging the requirements of many buffers when the data does not need to be accessed instantly. As such, their work is a refinement of the high-level and coarse-grained approach described in this paper. We focus on imperative program specifications and not on a subset of data-flow. They do not mention pipelining.

Gokhale and Stone [12] automatically allocate an array to a particular memory using a technique, called implicit enumeration, coupled with pragmas in the code that must be inserted by the user. They calculate the costs, based on scheduling conflicts, for putting each array in each memory. They try to find an enumeration that minimizes the costs. This effort differs from our approach. Unlike the algorithmic exhaustive search described in [12], we use a simple

greedy algorithm to determine a mapping that minimizes communication between the pipeline stages.

6. Conclusion

In this paper, we have described how parallelizing compiler technology can be adapted and integrated to automatically derive, from sequential C specifications, pipelined implementations for systems with multiple FPGAs and memories. We have described our implementation of these analyses in the DEFACTO system, and demonstrated this approach with a case study, a vision algorithm. We have presented experimental results derived automatically by our system to illustrate how these analyses can be integrated. Our current work focuses on fully integrating these analyses, and demonstrating automatically-derived pipelined implementations on our target architecture.

References

- [1] *WildstarTM Reconfigurable Computing Engines. User's Manual R3.3*, Annapolis Microsystems Inc., 1999.
- [2] J. Arnold, "The splash 2 software environment", in *Proc. IEEE Symp. on FPGAs for Custom Computing Machines*, pp. 88-93, 1993.
- [3] V. Balasundaram and K. Kennedy, "A technique for summarizing data access and its use in parallelism enhancing transformations", in *Proc. ACM Conf. on Programming Languages Design and Implementation*, pp. 41-53, 1989.
- [4] R. Barua, W. Lee, S. Amarasinghe and A. Agarwal, "Maps: a compiler-managed memory system for raw machines", in *Proc. 26th Int. Symp. on Comp. Arch.*, pp. 4-15, 1999.
- [5] T. Callahan and J. Wawrzynek, "Adapting software pipelining for reconfigurable computing," in *Proc. Int. Conf. on Compilers, Arch. and Synthesis for Embedded Systems*, pp. 57-64, 2000.
- [6] S. Carr and K. Kennedy, "Improving the ratio of memory operations to floating-point operations in loops", *ACM Trans. Programming Languages and Systems*, vol. 15(3), pp. 400-462, 1994.
- [7] D. Cronquist, P. Franklin, S. Berg and C. Ebeling, "Specifying and compiling applications for RaPiD", in *Proc. IEEE Symp. on FPGAs for Custom Computing Machines*, pp. 116-125, 1998.
- [8] P. Diniz and J. Park, "Automatic synthesis of data storage and control structure for FPGA-based computing engines", in *Proc. IEEE Symp. on FPGAs for Custom Computing Machines*, pp. 91-100, 2000.
- [9] P. Diniz and J. Park, "Data reorganization engines for the next generation of system-on-a-chip FPGAs", in *Proc. 10th ACM Int. Symp. on FPGAs*, pp. 237-244, 2002.
- [10] P. Diniz, M. Hall, J. Park, B. So, H. Ziegler, "Bridging the gap between compilation and synthesis in the DEFACTO system", in *Proc. 14th Workshop on Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2001.
- [11] D. D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao, *SPEC C: Specification Language and Design Methodology*, Boston: Kluwer Academic Publishers, 2000.
- [12] M. Gokhale and J. Stone, "Automatic allocation of arrays to memories in FPGA processors with multiple memory banks", in *Proc. IEEE Symp. on FPGAs for Custom Computing Machines*, pp. 63-69, 1999.
- [13] M. Gokhale and J. Stone, "Napa C: compiling for a hybrid RISC/FPGA architecture", in *Proc. IEEE Symp. on FPGAs for Custom Computing Machines*, pp. 126-135, 1998.
- [14] M. Gokhale, J. Stone, J. Arnold and M. Kalinowski, "Stream-oriented FPGA computing in Streams-C high level language", in *Proc. IEEE Symp. on FPGAs for Custom Computing Machines*, pp. 49-58, 2000.
- [15] S. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. Taylor and R. Laufer, "PipeRench: a coprocessor for streaming multimedia acceleration", in *Proc. of 26th Intl. Symp. on Comp. Arch.*, pp. 28-39, 1999.
- [16] N. Gracias, José Santos-Victor, "Underwater video mosaics as visual navigation maps.", in *Computer Vision and Image Understanding*, vol. 79(1), pp. 66-91, 2000.
- [17] M. Hall, S. Amarasinghe, B. Murphy, S. Liao and M. Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler," in *Proc. of Supercomputing*, pp. 1-26, 1995.
- [18] *LeonardoSpectrumTM*, v2000 Rev.1.d, Mentor Graphics Inc., 2000.
- [19] *MonetTM*, Mentor Graphics Inc., 1999.
- [20] S. Muchnik, *Advanced Compiler Design and Implementation*, San Francisco: Morgan Kaufmann Publishers, 1997.
- [21] P. Murthy and S. Bhattacharyya, "A buffer merging technique for reducing memory requirements of synchronous dataflow specifications," in *Proc. Int. Symp. on System Synthesis*, pp. 78-84, 1999.
- [22] J. G. Proakis and D. G. Manolakis, *Digital Signal Processing: Principles, Algorithms, and Applications*, 2nd ed., MacMillan Coll. Div., 1992.
- [23] R. Rinker, M. Carter, A. Patel, M. Chawathe, C. Ross, J. Hammes, W. Najjar and W. Bohm, "An automated process for compiling dataflow graphs into reconfigurable hardware," in *IEEE Trans. on VLSI Systems*, vol. 9(1), pp. 130-139, 2001.
- [24] J. Rutenberg, W. Lichtenstein, G. Gao, A. Stouchinin, "Software pipelining showdown: optimal vs. heuristic methods in a production compiler", in *Proc. ACM Conf. on Programming Languages Design and Implementation*, pp. 1-11, 1996.
- [25] B. So, H. Ziegler, and M. Hall, "Parallelization and locality analysis for adaptive computing systems," in *Proc. Parallel Architectures and Compilation Techniques, Workshop on Reconfigurable Computing*, pp. 27-34, 1999.
- [26] B. So, M. Hall, and P. Diniz, "A compiler approach to fast design space exploration in FPGA systems," in *Proc. ACM Conf. on Programming Languages Design and Implementation*, June 2002.

- [27] *The Stanford SUIF Compilation System*, Public Domain Software and Documentation available at <http://suif.stanford.edu>.
- [28] M. Weinhardt and W. Luk., "Pipelined vectorization for reconfigurable systems", in *Proc. IEEE Symp. of FPGAs for Custom Computing Machines*, pp. 52-62, 1999.
- [29] *Virtex-II 1.5v FPGA complete data sheet*, ds031(v1.7), XILINX Inc., 2001.