Bridging the Gap between Compilation and Synthesis in the DEFACTO System[1]

Pedro Diniz, Mary Hall, Joonseok Park, Byoungro So, Heidi Ziegler[2]
University of Southern California / Information Sciences Institute
4676 Admiralty Way, Suite 1001
Marina del Rey, California 90292
{pedro,mhall,joonseok,bso,ziegler}@isi.edu

Abstract. The DEFACTO project - a Design Environment For Adaptive Computing TechnOlogy - is a system that maps computations, expressed in high-level languages such as C, directly onto FPGA-based computing platforms. Major challenges are the inherent flexibility of FPGA hardware, capacity and timing constraints of the target FPGA devices, and accompanying speed-area trade-offs. To address these, DEFACTO combines parallelizing compiler technology with behavioral VHDL synthesis tools, obtaining the complementary advantages of the compiler's high-level analyses and transformations and synthesis' binding, allocation and scheduling of low-level hardware resources. To guide the compiler in the search of a *good* solution, we introduce the notion of ***balance*** between the rates at which data is fetched from memory and accessed by the computation, combined with ***estimation*** from behavioral synthesis. Since FPGA-based designs offer the potential for optimizing memory-related operations, we have also incorporated the ability to ***exploit parallel memory accesses*** and ***customize memory access protocols*** into the compiler analysis.

1. Introduction

The extreme flexibility of field programmable gate arrays (FPGAs), coupled with the widespread acceptance of hardware description languages (HDL) such as VHDL or Verilog, have made FPGAs the medium of choice for fast hardware prototyping and a popular vehicle for the realization of custom computing machines. Programming these reconfigurable systems, however, is an elaborate and lengthy process. The programmer must master all of the details of the hardware architecture, partitioning both the computation to each of the computing FPGAs and the data to the appropriate memories.

The standard approach to this problem requires that the programmer manually translate the program into an HDL representation, applying his or her knowledge of the FPGA specifics, identifying available parallelism and performing the necessary code transformations at that time. Using commercially available synthesis tools, typically the programmer can specify the maximum number of components of a given kind and/or the maximum area or clock period of the resulting design. The tool then determines the feasibility of the design by negotiating area and speed trade-offs, and generates a realizable design. Throughout the process, the burden is on the programmer to ensure that the intermediate representation of the program is kept consistent with the original version.

We believe the way to make programming of FPGA-based systems more accessible is

[1]Funded by the Defense Advanced Research Project Agency under contract # F30602-98-2-0113
[2]Funded through a Boeing Satellite Systems Doctoral Scholars Fellowship

to offer a high-level imperative programming paradigm, such as C, coupled with new compiler technology oriented towards FPGA designs. Programmers retain the advantages of a simple computational model via a high-level language but rely on powerful compiler analyses to identify parallelism, provide high-level loop transformations, and automate most of the tedious and error-prone mapping tasks. This paper describes preliminary experiences with DEFACTO - a Design Environment For Adaptive Computing TechnOlogy[7]. The DEFACTO system integrates compilation and behavioral synthesis to automatically map computations expressed in C to FPGA-based computing engines.

In DEFACTO, we explore the interaction between compiler transformations, most notably loop transformations, and their hardware realizations on FPGAs using commercially available synthesis tools. Since the hardware implementation is bound in terms of capacity, the compiler transformations must be driven by the space constraints and timing requirements for the design. Also, high latency and low bandwidth to external memory, relative to the computation rate, are often performance bottlenecks in FPGA-based systems, just as in conventional architectures. A compiler using high-level analysis information such as data dependence analysis, is in a privileged position to make decisions about which code transformations will lead to good hardware designs. To this end, we introduce the notion of ***balance*** to select the appropriate set of transformations.

A balanced solution is one where data is fetched from memory at approximately the same rate as it is consumed by the computation. To determine *balance*, we must examine the output of synthesis for a candidate design. Due to the complexity of resource binding and other by-products of synthesis, the compiler cannot accurately predict *a priori* the performance and space characteristics of the resulting design. Completely synthesizing a design is prohibitively slow (hours to days) and further, the compiler must try several designs to arrive at a good solution. For these reasons, we exploit ***estimation*** from behavioral synthesis tools such as Mentor Graphics' Monet^{TM}[15] to determine specific hardware parameters (*e.g.*, size and speed) with which the compiler can quantitatively evaluate the application of a high-level transformation to derive a balanced and feasible implementation of the loop nest computation. The compilation system uses data dependence and related analyses in combination with estimation information from a commercially available behavioral synthesis tool for the rapid evaluation of implementation options to steer a successful implementation of a selected portion of the computation to hardware. DEFACTO performs several code transformations that are useful for tailoring code to space constraints, such as ***loop tiling***, and ***loop unrolling***.

An additional focus of optimization in the DEFACTO system is to decrease latency and increase the effective bandwidth to external memory. While many locality optimizations used in conventional systems can be applied (*e.g.*, loop permutation, tiling, unrolling, scalar replacement), new optimizations are possible such as ***parallelization of memory accesses*** and ***customization of memory access protocols.*** Further, the configurability of FPGA hardware leads to new decision procedures for applying existing transformations. In this paper we describe how these techniques borrowed and adapted from compiler technology have been combined with a commercial behavioral synthesis tool, to automatically derive balanced and optimized designs on FPGA-based architectures.

The rest of this paper is organized as follows. In the next section we compare parallelizing compiler technology with behavioral synthesis technology used in commercially available tools. Section 3 presents an overview of the DEFACTO system. Section 4 describes analyses to uncover data reuse and exploit this knowledge in the context of FPGA-based data storage structures. Section 5 describes a transformation strategy for automating design space exploration in the presence of multiple loop transformations. This strategy uses a specific performance metric that takes data reuse and the space and time estimates derived by behavioral synthesis estimation. Section 6 describes the application of data partitioning and customization of memory controllers to substantially reduce the costs of accessing external memory from the FPGA. We survey related work in section 7. In section 8 we describe the current status of the DEFACTO system implementation and present some preliminary conclusions of its applicability to simple kernel applications.

2. Comparing Parallelizing Compiler Technology and Behavioral Synthesis

Behavioral synthesis tools map computations expressed in hardware-oriented programming languages, such as VHDL or Verilog, to FPGA and ASIC hardware designs. Behavioral specifications, as opposed to lower level logic or structural specifications, specify their computations without committing to a particular hardware implementation. In the same way that behavioral synthesis has raised the level of abstraction from logic synthesis, permitting designers to develop much more complex designs than previously possible, the goal of the DEFACTO system is to raise the level of abstraction even higher, to the C application level. While at first glance it may seem obvious that a compiler can straightforwardly generate correct VHDL code from a C input (in fact, some commercial systems already do this for restricted C dialects), what is less obvious is that existing compiler technology, developed for parallelization and optimizing memory hierarchies, can automate important optimizations that are difficult for a human designer to perform and exceed the capabilities of today's synthesis tools.

While there are some similarities between the optimizations performed in these two technologies, in many ways they offer complementary capabilities, as shown in Table 1.

Behavioral Synthesis	Parallelizing Compiler
Optimizations only on scalar variables	Optimizations on scalars and arrays
Optimizations only inside loop body	Optimizations inside loop body and across loop iterations
Supports user-controlled loop unrolling	Analysis guides automatic loop transformations
Manages registers and inter-operator communication	Optimizes memory accesses; evaluates trade-offs of different storage, on- and off-chip
Considers only single FPGA	System-level view: multiple FPGAs, memories
Performs allocation, binding and scheduling of hardware resources	No knowledge of hardware implementation of computation

Table 1: Comparison of Capabilities.

Behavioral synthesis performs three core functions: (1) binding operators and registers in the specification to hardware implementations (*e.g.*, selecting a ripple-carry adder to implement an addition); (2) resource allocation (*e.g.*, deciding how many ripple-carry adders are needed); and, (3) scheduling operations in particular clock cycles. In addition, behavioral synthesis supports some optimizations, but relies heavily on programmer pragmas to direct some of the mapping steps. For example, after loop unrolling, the tool will perform extensive optimizations on the resulting inner loop body, such as parallelizing and pipelining operations and minimizing registers and operators to save space. However, deciding the unroll amount is left up to the programmer.

The key advantage of parallelizing compiler technology over behavioral synthesis is the ability to perform data dependence analysis on array variables, used as a basis for parallelization, loop transformations and optimizing memory accesses. This technology permits optimization of designs with array variables, where some data resides in off-chip memories. Further, it enables reasoning about the benefits of code transformations (such as loop unrolling) without explicitly applying them.

In DEFACTO, we combine these technologies, as will be described in the next section. Behavioral VHDL specifications permit use of variables and high-level control constructs such as loops and conditions, reducing the gap between parallelizing compiler technology and logic synthesis tools.

3. Overview of DEFACTO Compilation and Synthesis System

3.1 Design Flow

Figure 1 shows the steps of automatic application mapping in the DEFACTO compiler. In the first step, the DEFACTO compiler takes an algorithm description written in C or FORTRAN, and performs pre-processing and several common optimizations. In the second step, the optimized code is partitioned into what will execute in software on the host and what will execute in hardware on the FPGAs. The following steps are loop

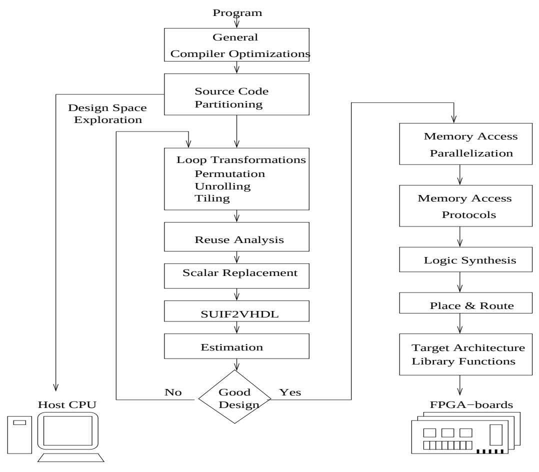

Figure 1. DEFACTO Design Flow

transformations and analyses to optimize the parallelism and data locality. Loop permutation[16] changes the nesting order of loops; we use it in DEFACTO to move loops that carry the most data reuse to the innermost loops where it can be better exploited. Loop unrolling[16] duplicates the loop body. Tiling[21] divides computation and data that are allocated to an FPGA into blocks. The next step is data reuse analysis, which identifies multiple accesses to the same memory location. Scalar replacement[6] makes the data reuse explicit by using the same register for read/write references that access the same memory location. SUIF2VHDL translates from SUIF to VHDL, which is the input to the synthesis process.

When the final computation part of the behavioral VHDL design has been derived, we optimize memory accesses by reorganizing data, and customizing memory access protocols. The resulting design is sent on to logic synthesis and place-and-route. Invocations to library functions are added to the portion of the code that will execute in software to load FPGA configurations and memories, synchronize and retrieve results.

A major issue in the automation process in DEFACTO is to understand how much and which set of loop transformations to apply and what are the good metrics by which to evaluate the resulting design. Our approach uses the estimation features of commercially available synthesis tools, which deliver fast (and presumably reasonably accurate) estimates for a given design. The synthesis tool provides area estimates and the number of clock cycles required for its scheduling. Using this information, DEFACTO refines the set of constraints for the design and evaluates what the resulting implementation estimate is. This estimation helps the DEFACTO system to quickly explore area and speed trade-offs in the design space.

3.2 Target Architectures

The compilation techniques designed for DEFACTO focus on targeting board-level systems similar to the WildStar/PCI board from Annapolis Micro Systems, depicted in Figure 2. Such systems consist of multiple interconnected FPGAs; each can access its own local memories. A larger shared system memory and general-purpose processor (GPP) are directly connected to the FPGA (this connection varies significantly across boards). The GPP is responsible for orchestrating the execution of the FPGAs by managing the flow of control and data from local memory to shared system memory in the application.

We chose a board-level architecture as a target for DEFACTO because it can be assembled with commodity parts, and commercial systems composed of such components are available. We expect the compilation approaches to form a solid foundation for a multi-board system, and also be applicable to system-on-a-chip devices that have multiple independent memory banks, each with configurable logic. The WildStar/PCI board in Figure 2 consists of three Xilinx Virtex FPGAs parts with up to a million gates each. One FPGA serves as a controller; the other two FPGAs are connected by a 64-bit channel, and each has two local SRAMs with 32-bit dedicated channels. The three FPGAs share additional system memory. When logic is effectively mapped to the FPGAs, the rate data

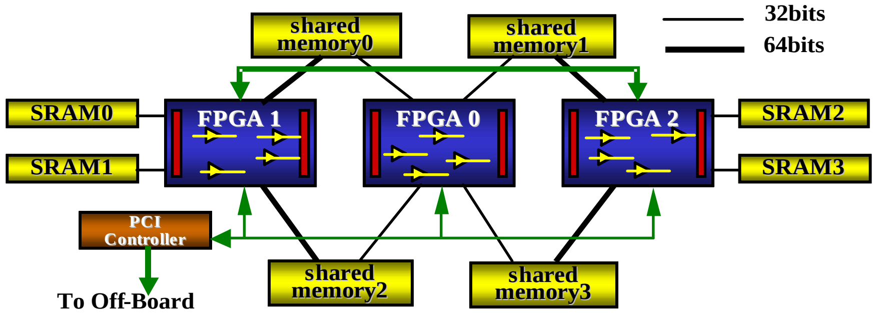

Figure 2. Target Architecture Example - WildStar/PCI Board

is consumed by the computation is significantly higher than the rate the local memories can provide making it desirable to exploit temporal reuse within the FPGA.

3.3 Target Applications and Examples

Image processing applications are one of the primary domains for the DEFACTO system. These applications typically manipulate large volumes of fine-grain data that, despite recent increases in FPGA capacity, must be mapped to external memories and then streamed through the FPGAs for processing. These applications typically combine the manipulation data organized as images in a regular fashion. They also tend to combine bit-level operations with sliding-window techniques making them amenable for data dependence and data reuse analysis which are a significant part of the focus of the DEFACTO compilation flow. In the next section we introduce these analyses and present a running example - a binary image correlation kernel application - that fits this class of applications. In addition, the example offers opportunities for the application of loop level transformations, such as loop unrolling and tiling, and because of its complexity, requires that the compiler negotiate the space-time trade-off.

4. Analyses and Transformations to Exploit Data Reuse

We next describe data reuse analysis for FPGA-based computing engines. While the core of this analysis uses data dependence information developed for parallelization and memory hierarchy optimizations in conventional architectures, FPGAs have no cache to reduce the latency of memory accesses. As a result, reuse analysis and optimizations are even more critical to our domain. Further, reuse analysis must precisely capture exact dependence distances so that data reuse can be exploited in compiler-controlled hardware structures, such as in internal registers. Thus, reuse analysis and associated transformations in our context more closely resemble what is required in register allocation for conventional architectures[6], rather than the large body of prior work on exploiting reuse in cache.

There are also several distinctions in the transformations that are performed in FPGA-based systems. The number of registers is not fixed as in a conventional architecture, so it is possible to tailor the number of registers to the requirements of the design. Second, customized hardware structures, not just registers, can be used to exploit the reuse in a way that maps to a time- and space-efficient design. The remainder of this section describes these differences.

4.1 Data Reuse Analysis

Data reuse analysis, as implemented in our compiler, identifies reuse opportunities, and builds a reuse graph for a given loop nest. *A reuse graph* G=(V,E) includes all reuse-carrying array references (V) and the reuse edges (E) between them. *Reuse edges* are represented by the reuse distance between two array references. The *reuse distance* is defined by the difference in iteration counts between the dependent references. The reuse distance is different from the dependence distance in two ways. First, only input and true dependences are candidates for data reuse. Our analysis also considers output dependences as candidates for redundant write access elimination, which also reduces memory accesses. Anti-dependences are not considered, as they do not represent data reuse. Secondly, as reuse analysis must precisely capture the distance between dependences, we do not include dependences with an *inconsistent distance*, where the iteration counts between the dependent references are not always constant. These references require a non-constant number of registers to exploit reuse. Further, the candidate array references for data reuse must be in the affine domain. In other words, the array access expression is composed of linear functions of loop indices and constants.

The reuse distance is represented in the same form as dependence vectors, *i.e.* a vector with the same number of elements as the number of loops in the loop nest. For example, a reuse distance of (1, 2) means that there are two loops in the nest and the reuse occurs in one iteration of the outer loop and two iterations of the inner loop. *A loop-independent data reuse* represented by a vector of all zeros, occurs when the same array element is referenced multiple times in the same iteration of the loop nest. A *reuse chain* includes all the array references that access the same memory location, a set of reuse instances. A *reuse instance* includes two array references, one source and one sink, and the reuse distance vector (reuse edge) between the two array references. References with no incoming reuse edges are called *generators*; each reuse chain has one generator. A generator provides the data that is used by all the references in the reuse chain.

Figure 3(a) shows an example, organized as a four-deep loop nest with a conditional accumulation. At each iteration the code determines whether or not a pixel of the input image should be accumulated with a running sum depending on the value of a *mask* array. An array reference *mask*[*i*][*j*] is read in every iteration of loops *m* and *n*. The reuse distance (*,*,0,0) from *mask*[*i*][*j*] to itself means the same array element is referenced repeatedly in every iteration of loop *m* and *n*. The array *image* also carries reuse, but the reuse distance is not consistent to avoid the accesses to the same memory location using a fixed number of registers. Therefore, we give up the reuse opportunities and the array *image* is read from memory.

4.2 Scalar Replacement

When the code shown in Figure 3(a) is mapped to an FPGA-based system, it will result in repeated accesses to the same external memory location. The compiler must transform the code to guide behavioral synthesis to exploit the reuse that the previously described analysis has identified.

Scalar replacement replaces certain array references with temporary scalar variables that will be mapped to on-chip registers by behavioral synthesis. Scalar replacement makes the data reuse explicit, resulting in a reduction in the memory accesses. It also avoids multiple write references by writing the redundant memory writes to a register and only writing the data to memory for the last write. Our scalar replacement has more freedom than previously proposed work[6] in that it can utilize the flexibility of FPGAs, *i.e.*, the number of registers to keep the frequently used data does not have a rigid limit, and is only constrained by the space limitation of the FPGA device and the complexity of designs involving very large numbers of registers.

The reuse distance between the generator and the tail reference in a reuse chain decides the required number of registers to exploit a possible data reuse. Intuitively, all the array references in the reuse chain can be replaced by accesses to the same register, and the generators are assigned data from memory.

Figure 3(b) shows the output of our scalar replacement. From the reuse information given by the reuse analysis, the reuse carrying array reference *mask*[*i*][*j*] is replaced with a scalar variable *mask*_0. Each array element *mask*[*i*][*j*] is kept in 16 different registers, *mask*_0 through *mask*_15. Data in these registers are loaded when *m* and *n* are 0 and

```
int th[60][60];
char mask[4][4];
char image[63][63];
for(m=0; m<60; m++){
  for(n=0; n<60; n++){
    sum = 0;
    for(i=0; i<4; i++){
      for(j=0; j<4; j++){
        if(mask[i][j] != 0)
          sum += image[m+i][n+j];
      }
    }
    th[m][n] = sum;
  }
}
```

$$\begin{tikzcd} \text{mask[i][j]} \arrow[loop left, "(*,*,0,0)"] \end{tikzcd}$$

(a) Source Code & Reuse Graph

```
for(m = 0; m < 60; m++){
  for(n = 0; n < 60; n++){
    sum = 0;
    for(i = 0; i < 4; i++){
      for(j = 0; j < 4; j++){
        if (m == 0 && n == 0)
          mask_0 = mask[i][j];
        if (mask_0 != 0)
          sum += image[m+i][n+j];
        rotate_register(mask_0,
         mask_1,mask_2,mask_3,
         mask_4,mask_5,mask_6,
         mask_7,mask_8,mask_9,
         mask_10,mask_11,mask_12,
         mask_13,mask_14,mask_15);
      }
    }
    th[m][n] = sum;
  }
}
```

(b) Scalar Replacement

```
for(m=0; m<60; m++)     {
   for(n=0; n<60; n++)     {
      sum = 0;
      for(i=0; i<4; i++)     {
         for(j=0; j<4; j += 2)     {
            if(mask[i][j] != 0)sum += image[m+i][n+j];
            if (mask[i][j+1] != 0) sum += image[m+i][n+j+1];
         }
      }
      th[m][n] = sum;
   }
}
```

(c) Loop Unrolling

```
for (m = 0; m < 60; m++)    {
 for (n = 0; n < 60; n++)    {
  sum = 0;
  for (i_tile = 0; i_tile < 2; i_tile++)    {
   if (m == 0 && n == 0)  {
    mask_0_0 = mask[2*i_tile][0]; mask_1_0 = mask[2*i_tile][1];
    mask_2_0 = mask[2*i_tile][2]; mask_3_0 = mask[2*i_tile][3];
    mask_4_0 = mask[2*i_tile+1][0]; mask_5_0 = mask[2*i_tile+1][1]
     mask_6_0 = mask[2*i_tile+1][2]; mask_7_0 = mask[2*i_tile+1][3
   }
   if (mask_0_0 != 0) sum += image[m+2*i_tile][n];
   if (mask_1_0 != 0) sum += image[m+2*i_tile][n+1];
   if (mask_2_0 != 0) sum += image[m+2*i_tile][n+2];
   if (mask_3_0 != 0) sum += image[m+2*i_tile][n+3];
   if (mask_4_0 != 0) sum += image[m+2*i_tile+1][n];
   if (mask_5_0 != 0) sum += image[m+2*i_tile+1][n+1];
   if (mask_6_0 != 0) sum += image[m+2*i_tile+1][n+2];
   if (mask_7_0 != 0) sum += image[m+2*i_tile+1][n+3];
   swap_reg(mask_0_0, mask_0_1); swap_reg(mask_1_0, mask_1_1);
   swap_reg(mask_2_0, mask_2_1); swap_reg(mask_3_0, mask_3_1);
   swap_reg(mask_4_0, mask_4_1); swap_reg(mask_5_0, mask_5_1);
   swap_reg(mask_6_0, mask_6_1); swap_reg(mask_7_0, mask_7_1);
  }
  th[m][n] = sum;
 }
}
```

(d) Tiling

Figure 3. ATR Kernel

rotated in each iteration of the j loop so that *mask_*0 always contains the correct data, avoiding checking loop variable *i* and *j* repeatedly. The rotate operation can be done in parallel in hardware.

Our compiler also performs loop peeling to isolate the condition (m==0 && n==0) to eliminate the conditional read to memory. For clarity of presentation, the results of this transformation are not shown here.

4.3 Tapped Delay Lines

An alternative to scalar replacement can be used if reuse is further restricted to input dependences with a consistent dependence carried by the innermost loop in a nest (or other loops in the nest if inner loops are fully unrolled). For example, if a computation accesses consecutive and overlapping elements of a given array (such as scanning an array corresponding to rows or columns of an image) over consecutive iterations of a loop, it is possible to store the values across iterations in a linearly-connected set of registers - known as *tapped delay lines*. The data accessed in one iteration differs from the data in the previous iteration by an ajacent memory location. All but one element of the delay line can be reused, substantially reducing the number of memory accesses. This important improvement opportunity has long been recognized by experienced designers who map computations by hand to hardware, by mapping elements in memory to tapped delay lines. As part of our previous work, we developed several compiler analyses that recognize and exploit opportunities in the source program for data reuse and can be subsequently mapped to tapped delay lines [8,16].

The application reuses the data in different registers by moving in tandem, across loop iterations, the data stored in all of the registers in the tapped delay line. By doing this shifting operation, the computation effectively reuses the data in all but one of the registers in the tapped delay line, therefore reducing the number of memory accesses. Although the same sort of reuse could be exploited by scalar replacement, the tapped delay line structure results in a more efficient hardware mapping, as we can incorporate a library with hand-placed components.

Our approach for identifying this reuse and exploiting tapped delay lines in FPGA designs is described elsewhere[10]. While the current analysis for tapped delay lines is limited to input data dependences on innermost loops, we feel this covers a substantial set of image processing applications that scan subsections of images stored in arrays.

5. Transformation Strategy for Automating Design Space Exploration

Current practice when developing designs for FPGA-based systems is for the application programmer to develop a series of designs and go through numerous iterations of synthesizing the design, examining the results, and modifying the design to trade off performance and space. To automate this process, called *design space exploration*, we must define a set of transformations to be applied and metrics to evaluate specific optimized designs among candidates in the design space. Simply stated, the optimization criteria for mapping a single loop nest to FPGA-based systems are as follows: (1) the design must not exceed the capacity constraints of the system; (2) the execution time should be minimized; and, (3) for a given performance, FPGA space usage should be minimized. The motivation for the first two criteria is obvious, but the third criterion is also needed for several reasons. First, if two designs have equivalent performance, the smaller design is more desirable, in that it frees up space for other uses of the FPGA logic, such as to map other loop nests. In addition, a smaller design usually has less routing complexity, and may achieve a faster target clock rate. Moreover, the third criterion suggests a strategy for selecting among a set of candidate designs that meet the first two criteria.

The DEFACTO compiler uses two metrics to guide the selection of a design. First, results of estimation provide space usage of the design, related to criterion 1 above. Another important metric used to guide the selection of a design, related to criteria 2 and 3, is *balance*, defined by the following equation.

$$balance = \frac{F}{C},$$

where F refers to the data fetch rate, the total data bits that memory can provide per cycle, and C refers to the data consumption rate, total data bits that computation can consume during the computational delay. If *balance* is close to one, both memories and FPGAs are busy. If *balance* is less than one, the design is memory bound; if greater than one, it is compute bound. When a design is not balanced, this metric suggests whether more resources should be devoted to improving computation time or memory time.

The DEFACTO compiler adjusts *balance* by increasing/decreasing operator and memory parallelism, and increasing/decreasing on-chip storage. For example, consider loop unrolling[16]. Figure 3(c) shows the result of unrolling the inner loop *j* once on the original code in Figure 3(a). The number of memory accesses in one iteration in the unrolled code is doubled, and, since behavioral synthesis will only schedule independent memory accesses in parallel if they appear in the same inner loop body, the unrolled code fetches data at twice the rate of the original loop body. In addition, if the clock period is sufficiently long, even the dependent computations, two accumulations, can be scheduled in the same cycle. Therefore, unrolling can increase both the consumption rate and the fetch rate.

We can model the relationship between the unrolling factor and the fetch and consumption rates under the following assumptions: (1) computation and memory accesses are completely overlapped, requiring that addresses for memory accesses can be calculated at the rate at which the memory can provide data; (2) the target architecture, target clock rate, and memory latency are fixed; (3) data is laid out in independent memories to maximize memory parallelism, as described in the next section; and, (4) estimates derived from a combination of behavioral synthesis and compiler analysis are

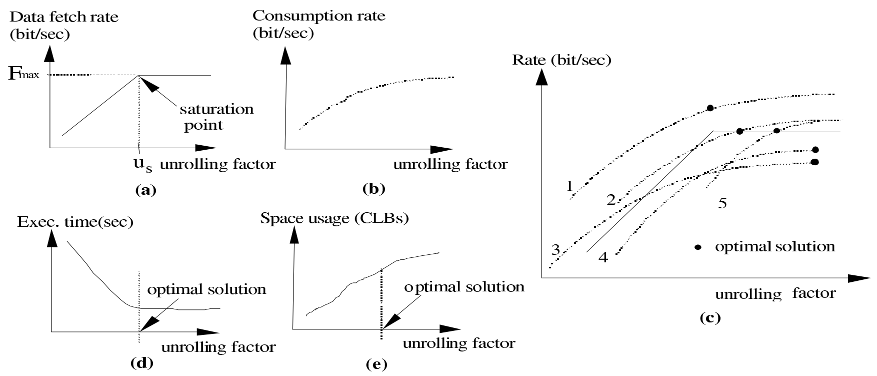

Figure 4. Data Fetch and Consumption Rates.

accurate. Then, the fetch rate increases linearly in the unroll amount until it saturates at an upper limit based on memory parallelism in the application and the bandwidth of the architecture, as shown in Figure 4(a). The consumption rate, which combines the estimate of computation delay (ignoring memory latency) taken directly from behavioral synthesis with the compiler's model of the data fetch rate, increases with the unroll amount, but not linearly, and does not saturate, as shown in Figure 4(b). The relationship is sublinear because executing operations in parallel depends on the availability of operands (the data fetch rate) and data dependences across operations in the unrolled loop body.

5.1 Optimal Unrolling Factor Search

Given the above model, the compiler uses *balance* to guide determining the optimal unroll amount. Given the properties of the data consumption rate and the data fetch rate described in the previous section, there could be five possible scenarios that can happen between the data fetch rate and the consumption rate, as shown in Figure 4(c). In any scenario, the slower rate dominates the overall execution time, and the faster rate cannot get its full performance. In scenario one and four, the data fetch rate and the consumption rate do not cross each other. In scenario three and five, there is one crossing point, where the *balance* is one. In scenario two, there are two crossing points. The solution with higher rates between the two crossing points, of course, results in better performance, provided the solution meets the space constraint.

In all cases, the optimal solution can be found beyond the saturation point, if the solution meets the space constraint. In scenario two and three, for example, even though there is a balanced solution before the saturation point, a greater unrolling factor performs better than the balanced solution because its dominant rate is greater. If a balanced solution exists beyond the knee (saturation point) of the fetch rate, increasing the unrolling factor further will not improve the overall performance, since the slower memory operations dominate the overall performance. Figure 4(d) illustrates the relationship between the unrolling factor and overall performance. The knee of the graph is the optimal solution. It is a balanced solution in scenarios two and five if it meets the space constraint. In scenario one, it is the solution at the saturation point. In scenario three and four, the optimal solution is the maximum unrolling factor. Figure 4(e) shows the relationship between the unrolling factor and the space usage. The space usage of a design grows as the unrolling factor increases, even beyond the optimal solution.

5.2 Beyond Unrolling: Tiling

A similar optimization strategy can be used for tiling, which is the multi-loop analog of unrolling. Tiling divides computation and data that are allocated to an FPGA into blocks.
Traditionally, tiling is used to manage the space limitation of cache memories, as it increases the frequency of data reuse, so that data is still available in cache by the time it is reused. In DEFACTO, we use tiling to manage the space for on-chip data storage. The compiler unrolls the entire tiled loops completely not including the tile-controlling loops. Therefore, the unrolling factor is controlled by the tile size on each loop. By changing the tile size, we control the parallelism, locality, and space usage for on-chip storage.

Figure 3(d) shows an example of tiling. Two inner loops of the original program in Figure 3(a) are tiled with tile size two on loop *i* and tile size four on loop *j*, and the tiled loops are completely unrolled.

6. Parallelization and Customization of Memory Accesses

In DEFACTO we explore several techniques to increase aggregate external memory bandwidth and to decrease the amount of latency per memory access. These include 1) introducing a new architecture to create an optimized external-memory-to-FPGA interface and designing optimized circuitry to implement the architecture; 2) customizing data partitioning across a set of memories to gain parallel accesses; and 3) performing data reorganization and packing to create a tailored data layout.

6.1 Memory Interface Architecture

FPGAs offer a unique opportunity to exploit application specific characteristics in terms of the design and implementation of the datapath-to-external-memory interface. In order to build a modular, yet efficient interface to external memory, we defined two interfaces and a set of parameterizable abstractions that can be integrated with existing synthesis tools. These interfaces decouple target-architecture dependent characteristics between the datapath and external memories. One interface generates all the external memory control signals matching the vendor-specific interface signals - physical address, *enable*, *write_enable*, *data_in/out*, etc. In addition, this interface allows the designer to exploit application-specific data access patterns by providing support for pipelined memory accesses. The second interface defines the scheduling of the various memory operations allowing the designer to define application specific scheduling strategies. We have implemented these two interfaces over a simple, yet modular, architecture a described in detail in [17]. This architecture consists of conversion FIFO queues, and channels, an address generation unit (AGU), and a memory controller for each of the external memories in the design.

In this context, we introduce two abstractions, data ports and data channels. A data port defines several hardware attributes for the physical connection of wires from the memory controller (the entity responsible for moving data bits to and from the external memory) to the datapath ports. A data channel characterizes (for a given loop execution) the way the data is to be accessed from/to memory to/from a datapath port. Examples include sequential or strided accesses. A channel also includes a conversion FIFO. The FIFO queue allows for the prefetching of data from a memory, and also allows for deferred writebacks to memory such that incoming data transfers, necessary to keep the computation executing, take priority. The conversion FIFO queue also allows for the implementation of data packing and unpacking operations implicitly, if any, so that no modifications to the datapath either producing or consuming the data are necessary. Over the lifetime of the execution of a given computation it is possible to map multiple data channels to the same datapath port by redefining the parameters of the data stream associated with it.

6.2 Memory Interface Controller and Scheduling Optimizations

Part of the target hardware design consists of auxiliary circuitry to deal with the external memories. These less glamorous and often neglected entities provide the means to generate addresses, carry out memory signaling and temporarily store data to and from external memories dealing with the pins and timing vagaries of the vendor-specific memory and FPGA interfaces. Because of its key role in exploiting application-specific features to reduce memory latency, we focus our attention on the memory channel controller.

The role of a memory channel controller is to serialize multiple memory requests for a set of data channels associated with a single memory. A naive implementation of the memory controller would call for the controller to determine for each data channel whether or not a memory access is required. If required the controller must issue an address and engage in the memory signaling protocol. All these phases are implemented via finite state machines that the compiler synthesizes from parameterized VHDL code generation templates and which correspond to a substantial amount of complexity of the final design.

The current implementation allows for application-specific scheduling by defining a distinct ordering of the memory accesses for the different channels. While predicting exact memory access order is impossible, our compiler can reduce the number of clock cycles required to access the data corresponding to many memory channels by bypassing memory controller states. These "*test*" states can be eliminated when the controller is assured that if a datum is required by a given channels, it is also true that it is required by a set of other channels. In this situation the controller need not explicitly check whether or not a memory access is required. The benefits of this optimization when compounded with pipelined memory access mode can lead to a substantial performance improvement of up to 50% for a small set of image processing kernels[17].

6.3 Data Partitioning

To take advantage of the multiple external memories associated with an FPGA, in such a way as to obtain the maximum number of parallel reads and writes, we want to divide program data among them in a useful way. For very simple programs, it may be easy for the programmer to simply divide each array into equal parts, assigning each to an associated memory to gain some parallel memory accesses. Determining exactly how to divide the data and place it in a memory to gain the maximum amount of parallelism requires further analysis. Therefore, in contrast to having the programmer specify a data partition by hand, we automatically partition an array across available external memories by building a set of constraints in a linear algebra framework and then solving for a partitioning solution.

We borrow from multiprocessor data partitioning and layout[1]. The multiprocessor analyses considered cache effects, sequential code execution on a per processor basis, and were limited by the processors' fixed architecture and instruction set. We add constraints to the system based on how the computation is converted into a datapath, how commercial synthesis tools perform scheduling, and using information about our own communication architecture between off-chip memories and the FPGA (*i.e.* the memory access protocols described earlier in this paper). While the data access patterns may suggest different partitions for the array at different execution points, the communication costs associated with repartitioning the data are high. Therefore, we choose one global partition for the entire execution based on loop parameters. SUIF annotations[18] capture the partitioning scheme for each array. This information includes to which memory or memories data is partitioned and whether the array is partitioned on a per element, row, column or block basis in contrast to the modulo unrolling used for bank disambiguation in the MIT Raw project[5]. Host code, initially transferring data to the FPGA memories before program execution and also returning data to the host memory upon program completion, implements the partitioning scheme.

6.4 Data Reorganization and Packing

Similar in concept to increasing spatial locality in a cache line described by Anderson *et. al.*[2], we use data reorganization and packing techniques to increase spatial locality and decrease access latency inherent in the system. By reorganizing data assigned to a particular memory, we can create a tailored layout that will allow the channel access pattern to be sequential or strided, thus taking advantage of the optimized circuitry. By identifying and packing array elements whose sizes are less than the width of one data transfer, we are able to match the computation rate. We may even be able to uncover further information that could be used to direct additional loop transformations to optimize the system. An example would be that by packing data, we gain increased bandwidth and realize that we could further unroll the loop.

6.5 Example

We now illustrate the application of the two compiler techniques described in the previous sections to the running example described earlier in this paper. First we partition the arrays *mask* and *image* across the two external memories. From the program, we build a set of constraints and solve for the partitions. Both the *mask* and *image* arrays have similar access patterns; therefore, the resulting partitions are similar. Within the loop body, the 16 elements of the *mask* array are accessed in a row wise manner. The even rows of the *mask* array are partitioned to one memory and the odd to the other. The array *image* is also accessed in a row wise manner, in blocks of four by four elements, within the loop body. Since square blocks are accessed on each loop iteration, we can partition either row or column wise. Since no code transformations are necessary for a row wise partition, we choose that scheme. The even rows of array *image* are assigned to one memory and the odds to the other. Similarly shaded data in each memory are accessed by the FPGA in parallel. .

We reorganize the partitioned data so that the even rows of the array *mask* are contiguous in the assigned external memory. We reorganize the odd rows in a similar manner for both arrays. Image processing applications, such as the example, that operate on pixel-based images using eight bit values can have a substantial reduction in the number of memory accesses as a single 32-bit memory transfer can retrieve/store four consecutive pixel values. We pack data elements in memory to take advantage of the full channel width.

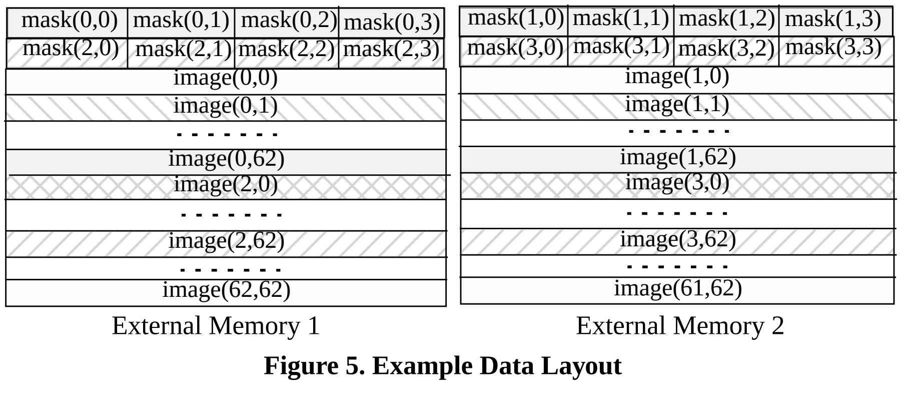

Figure 5. Example Data Layout

7. Related Work

7.1 Configurable Architectures

Several research efforts have concentrated on the development of new reconfigurable architectures (*e.g.*, [8] [13] [14]). These efforts differ from our research in two main aspects. First, as new architectures, they chose which components of the systems are reconfigurable and what are the macro instructions the non-reconfigurable portion can execute. As such they have develop target architecture-specific compiler and synthesis tools and have not taken advantage of the wealth of techniques available in behavioral synthesis tools.

In DEFACTO we use commercially available FPGAs and corresponding tools and synthesize from the ground up all of the control structures in the FPGA to allow them to operate autonomously. Because other approaches do not use commercial synthesis tools they avoid the performance and interface issues with place-and-route.

7.2 Compilation Systems for Configurable Architectures

Like our effort other researchers have focused on using automatic data dependence and compiler analysis to aid the mapping of computations to FPGA-based machines. Weinhardt[19] describes a set of program transformations for the pipelined execution of loops with loop-carried dependences onto custom machines using a pipeline control unit and an approach similar to ours. He also recognizes the benefit of data reuse but does not present a compiler algorithm. No references in the literature mention multi-dimensional arrays as well as the implementation of a decision procedure to analyze the various trade-off choices for the implementation of data queues. We further use loop unrolling to expose more array references in the program and therefore infer data reuse for the unrolled loops.

The Napa-C compiler effort[11][12] explores the problem of automatic mapping array variables to memory banks. This work is orthogonal to ours. We are interested in implementing an efficient and autonomous computing engine on each FPGA of a multi-FPGA board. Their RISC-based interface as expected is very similar to our target design architecture as a way to control the complexity of the interface between the FPGAs and the host processor. A major difference is the fact that we target commercially available components and not an embedded custom architecture.

Multiprocessor systems are highly effected by the computation and data partitioning across processors and memory. Anderson’s work[1] presents an algorithm that automates this mapping to free the programmer from performing the process by hand. The process uses a linear algebra framework to set up constraints contained in the user program and then solves for a data and computation distribution. Data reorganization[2] may need to be calculated as the algorithm looks within procedures as well as across procedure boundaries. This work takes into account effects from the cache(s) and sequential code execution on a processor. We replace these processor specifics with hardware-synthesis-specific information.

In the context of the MIT Raw project - a tiled and configurable architecture[4] the compiler partitions the computation and data among the cores and programs the communication channels to best suite the communication pattern for each application. Barua *et. al.*[5] present a code transformation technique using compile-time information to manage a distributed address space memory, with exposed memory banks. A memory reference instruction is said to be bank-disambiguated when the compiler guarantees that every dynamic instance of that instruction references the same compile-time-known bank. Using the bank information, array accesses are uniformly laid out across tiles, via a low order interleaving scheme. The low order bits of the address specify the tile. To achieve program correctness, the loop body must be unrolled. While arrays are partitioned across memories similar to our work, the only scheme is low order interleaving and we propose several others.

7.3 Data Reuse Analysis

The data reuse analysis developed for the DEFACTO system differs from the data reuse analysis described by Carr and Kennedy[6] in several points. First, our analysis includes output dependence in addition to the true and input dependences in [6] to decrease unnecessary store operations. Secondly, we exploit reuse along all loops of a loop nest and unlike[6] we are not restricted to the innermost loop only. Since loop permutation moves the loop that exploits the most reuse innermost and it takes more register to exploit reuse in outer loops, this approach is sufficient to get the most benefit if the number of registers are limited or if cache is the target memory hierarchy, where it is hard to exploit outer loop's locality due to cache size limitation. In addition, considering localized iteration space where register contents are reused throughout all the iterations without shifting between registers, the number of registers can be significantly reduced. Thirdly, Carr/ Kennedy’s analysis use Callahan's balance model to evaluate how the loop matches the capabilities of a specific architecture. Our notion of balance can explain whether either memory side or computation side is the bottleneck. It also can guide whether the limited space on an FPGA must be devoted more to memory operations or computations. Fourth, their complexity of optimization problem is greater than ours. Our analysis employs a commercial synthesis tool to get a feedback on the space use and execution rate. By doing so, our reuse analysis reduces the size of the candidate tile size search space and tunes the unrolling factors.

8. Project Status and Conclusion

We have implemented the bulk of what has been presented in this paper in the DEFACTO system. We have demonstrated a fully automatic design flow for DEFACTO that consists of a subset of the passes described in Section 3 on three image processing kernels, mapping from a C specification to working designs on the Annapolis WildStar Board described in Section 3.2.

A major focus of our current work is integrating the compiler and synthesis tool via estimation for design space exploration. We have developed an interface to estimation from the Synopsys Behavioral Compiler as well as Monet[9], and are using it to automatically calculate *balance* for a particular optimized loop nest design.

With the growing size of FPGA devices and the increased complexity of the place-and-route and mapping related passes in commercially available synthesis tools, we envision estimation as a fundamental technique for quickly exploring a wide range of implementation options while keeping the compilation/synthesis time at a reasonable cost. The DEFACTO system shows that it is possible to successfully map applications written in imperative programming languages directly to FPGA-based computing boards combining the application of traditional data dependence analysis techniques with commercially available behavioral synthesis tool.

References

[1] J. M. Anderson, Ph.D Thesis, Stanford University, March 1997. Published as Stanford CSL-TR-97-719.

[2] J. Anderson, S. Amarasinghe, and M. Lam., “Data and Computation Transformations for Multiprocessors,” in *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP’95)*, Santa Barbara, CA, July 19-21, 1995.

[3] Annapolis Micro Systems Inc., “WildStar^{TM} Reconfigurable Computing Engines. User’s Manual R3.3”, 1999.

[4] J. Babb, M. Rinard, A. Moritz, W. Lee, M. Frank, R. Barua and S. Amarasinghe.”Parallelizing Applications into Silicon”. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM‘99)*, IEEE Computer Society Press, Los Alamitos, 1999, pp. 70-81.

[5] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal, “Memory Bank Disambiguation using Modulo Unrolling for Raw Machines“. In *Proceedings of the Fifth International Conference on High Performance Computing*, Chennai, India, 1998, December 17-20.

[6] S. Carr and K. Kennedy, "Improving the ratio of memory operations to floating-point operations in loops". In *ACM Transactions on Programming Languages and Systems*, 15(3):400-462, July 1994.

[7] K. Bondalapati, P. Diniz, P. Duncan, J. Granacki, M. Hall, R. Jain, H. Ziegler, “DEFACTO: Design Environment for Adaptive Computing TechnOlogy”. In *Proceedings of the Reconfigurable Architecture Workshop*, held in conjunction with the International Parallel Processing Symposium, San Juan, Puerto Rico, April, 1999.

[8] D. Cronquist, P. Franklin, S. Berg and C. Ebeling, “Specifying and Compiling Applications for RaPiD”. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM‘98)*, IEEE Computer Society Press, Los Alamitos, 1998, pp. 116-125.

[9] P. Diniz and A. Venkatachar, “A Behavioral Synthesis Estimation Interface for Configurable Computing”, In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM‘01)*, IEEE Computer Society Press, Los Alamitos, Calif., Oct. 2001.

[10] P. Diniz and J. Park, “Automatic Synthesis of Data Storage and Contol Structures for FPGA-based Computing Machines“. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM‘00*), IEEE Computer Society Press, Los Alamitos, Calif., Oct. 2000, pp. 70-79.

[11] M. Gokhale and J. Stone, “Automatic Allocation of Arrays to Memories in FPGA Processors with Multiple Memory Banks”. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM‘99)*, IEEE Computer Society Press, Los Alamitos, 1999, pp. 63-69.

[12] M. Gokhale and J. Stone, “Napa C: Compiling for a Hybrid RISC/FPGA Architecture”. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM‘98*), IEEE Computer Society Press, Los Alamitos, Calif., 1998, pp. 126-135.

[13] S. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. Taylor and R. Laufer, “PipeRench: A Coprocessor for Streaming Multimedia Acceleration”. In *Proceedings of 26th Intl. Symp. on Computer Architecture (ISCA’99)*, ACM Press, New York, 1999.

[14] J. Hauser and J. Wawrzynek, “Garp: A MIPS Processor with a Reconfigurable Coprocessor”. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM‘97)*, IEEE Computer Society Press, Los Alamitos, 1997, pp.12-21

[15] Monet^{TM} User’s Manual, Mentor Graphics Inc., 2000.

[16] S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Fransisco, Calif.. 1997.

[17] J. Park and P. Diniz, "Synthesis of Memory Access Controller for Streamed Data Applications for FPGA-based Computing Engines". To appear in *Proceedings of the 14th*

International Symposium on System Synthesis (ISSS'2001), IEEE Computer Society Press, Los Alamitos, Calif., Oct. 2001.

[18] "The Stanford SUIF Compilation System". Public Domain Software and Documentation available at http://suif.stanford.edu

[19] M. Weinhardt and W. Luk., "Pipelined Vectorization for Reconfigurable Systems". In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'99)*, IEEE Computer Society Press, Los Alamitos, Calif.,1999, pp. 52-62.

[20] M. Weinhardt and W. Luk, "Memory Access Optimization and RAM Inference for Pipeline Vectorization". In *Proceedings of the 9th International Workshop on Field Programmable Logic and Applications (FPL'99)*, Springer Verlag LNCS Vol. 1673, 1999, pp. 61-70.

[21] M. Wolf and M. Lam, "A Loop Transformation Theory and an Algorithm for Maximizing Parallelism". In *IEEE Transactions on Parallel and Distributed Systems*, Oct. 1991.

[22] M. Wolfe, *High-Performance Compilers for Parallel Computing*, Addison-Wesley, 1996.