Automatic Synthesis of Data Storage and Control Structures for FPGA-based Computing Engines

Pedro Diniz and Joonseok Park
Information Sciences Institute / University of Southern California
4676 Admiralty Way, Suite 1001
Marina del Rey, California, 90292-6695
{pedro,joonseok}@isi.edu

Abstract

Mapping computations written in high-level programming languages to FPGA-based computing engines requires programmers to generate the datapath responsible for the core of the computation as well as control structure to generate the appropriate control signals to orchestrate its execution. This paper addresses the issue of automatic generation of data storage and control structures for FPGA-based reconfigurable computing engines using existing compiler data dependence analysis techniques. We describe a set of parameterizable data storage and control structures used as the target of our prototype compiler. We present a compiler analysis algorithm to derive the parameters of the data storage structures to minimize the required memory bandwidth of the implementation. We also describe a complete compilation scheme for mapping loops that manipulate multi-dimensional array variables to hardware. We present preliminary simulation results for complete designs generated manually using the results of the compiler analysis. These preliminary results show that is possible to successfully integrate compiler data dependence analysis with existing commercial synthesis tools.

Keywords: FPGA-based reconfigurable computing architectures, compilation, program analysis, data queues.

1. Introduction

The extreme flexibility of Field-Programmable-Gate-Arrays (FPGAs) coupled with the widespread acceptance of hardware description languages such as VHDL or Verilog have made FPGAs the medium of choice for fast prototyping of hardware implementations and a popular vehicle for the realization of custom computing machines. Typical architectures consists of a varying number of computing FPGAs attached to their local memories. These FPGAs are connected via a predefined topology and their execution supported either by an extra controller unit or by a general purpose processor [1,2,13].

Programming these reconfigurable systems however, is a elaborate and lengthy process. The programmer must master all of the details of the hardware architecture, partition the computation and data so that it can allocate functional units to each of the computing FPGAs. In this process the programmer must partition the code between the code that orchestrates the whole execution (typically a C program executing on a general purpose processor) and the code that is to be mapped onto each FPGA. For this code, the programmer must specify the datapath that will carry out the computation using an hardware description language. In addition the programmer must tie the datapath with its controller, typically a finite-state-machine (FSM). This FSM ensures the data is stored and retrieved to and from the FPGA ports on a specific clock cycle. If a pipelined implementation is sought the programmer must also take responsibility for the correct synchronization between pipeline stages and for handling the input/output data at the correct rate.

Funded by the Defense Advanced Research Project Agency under contract number F30602-98-2-0113

For architectures with multiple FPGAs the programmer must ensure the rates of each FPGA match the rate of the respective controllers. If the data for each FPGA is fetched from memory by a external controller (typically a general purpose processor) the programmer must generate a correct control programmer for all of the FPGA executing simultaneously. This control program (occasionally implemented by a designated FPGA) in effect mimics the execution of concurrent threads, on each FPGA and must ensure that each data consumed and generated by each FPGA is done at the correct clock cycle. When the speed of the devices or the datapath implementations on the FPGA changes the programmer must revise and change the controller and/or the program executing on the host.

Of particular interest to this research are digital image processing applications. These applications tend to concentrate the computation in tightly nested loops that manipulate dense multi-dimensional array data structures expressed in languages such as MatLab or C. The rapid translation of these computations to FPGA posses several problems. First, to our knowledge no commercially available synthesis tool can translate multi-dimensional array access patterns to hardware that performs the fetching of consecutive array index accesses. The option of array linearization is not desirable in many cases as it complicates high-level compiler analysis and produces code that is not only hard to analyze but also difficult to maintain. Second, the lack of advance data dependence analysis precludes the application of analysis techniques for the automatic derivation of efficient storage models for the data manipulated in tight loop nests. Many efficient implementations of these computations done by programmers pervasively use data queues to significantly reduce the required number of memory accesses in effect caching data in internal registers. This experience suggests these data storage structures are very important in the efficient implementation of these classes of computations.

In this paper we address the issue of control and data storage structures for FPGA-based reconfigurable computing engines. We begin by describing a set of parameterizable control blocks. Like recent approaches to module generators (e.g., [4,5]) the control blocks presented in this paper can be combined in aggregate control structures. We describe address generators with auto-increment capabilities, buffered input and output queues and pipelining control structures. In the context of FPGAs where minimizing the number of reconfiguration is still an important metric, the structure proposed in this paper can be used by multiple datapath and/or reused across different FPGA configurations.

We also describe the application of data dependence analysis to the automatic generation of data queues for computations that manipulate multi-dimensional array using affine index access functions. The goal of this analysis is to explore a wide range of program transformations with the goal of reducing the number of required memory accesses, and therefore reducing the required memory bandwidth for a particular implementation. We developed a compilation and synthesis strategy using a set of high-level parameterizable building blocks that allows a compiler to automatically generate complete designs for boards consisting of multiple computing FPGAs without host processor direct intervention. We have implemented and tested the algorithms and the code generation (in synthesizable VHDL specifications) for a set of kernel digital image processing applications. We provide experimental simulation results for the automatically generated set of designs for this set of computations.

This paper makes the following contributions:

1. It presents a set of parameterizable control and data storage structures for the mapping of computations consisting of loops that manipulate multi-dimensional arrays with affine access functions onto FPGA-based systems.
2. It presents an analysis algorithm for the automatic derivation of these control structures. Of particular significance if the automatic derivation of the data queues (number, length and stride of the corresponding array accesses).
3. It presents a simple algorithm that evaluates several possible designs that exploit data queues. This algorithm uses the data access patterns in the loop nest to choose a design with the lowest number of required memory accesses or bandwidth.
4. It describes a compilation synthesis scheme for FPGA-based computing engines for computations expresses as loop nests that manipulate array variables with affine index functions.
5. It presents preliminary experimental simulation results for the automatic translation of a set of computation using our compilation/synthesis strategy and our set of defined control structures using commercial synthesis tools. These preliminary results indicate these control structures can serve as the basis of a successful compilation and synthesis flow.

We believe the control and data storage structures described here can play an important role as part of a library of synthesizable modules, but as well as used to facilitate the tasks of a compiler tools in the mapping of high-level programming constructs to hardware. The fact that programmers often use similar data and control structures in their hand crafted designs is clear evidence of their important in the effective mapping of digital image processing computation to FPGA-based implementations.

The rest of this paper is structured as follows. We next introduce a set of parameterized modules and the proposed compilation target architecture via an example. Next we describe the various control structures and their rationale. Section 4 describes our compiler analysis algorithms for the automatic derivation of data queues.[1] Section 5 presents preliminary simulation experimental results for a set of digital image processing applications. In section 6 we survey related work and conclude in section 7.

2. Example

We now illustrate the use of basic data storage and control structures for the automatic mapping of an example computation onto an FPGA-based computing engine. The computation is written in C as depicted in Figure 1. It consists of a single loop nest and computes the Sobel edge detection algorithm over an 8 bit gray-scale image. The image is stored in a 2-dimensional *img* array of characters. The output is stored in the 2-dimensional *edge* array.

```
char img[SIZE][SIZE], edge [SIZE][SIZE];
int uh1, uh2, threshold;
for (i=0; i < SIZE - 4; i++) {
  for (j=0; j < SIZE - 4; j++) {
    uh1= (((- img[i][j]) + (- (2 * img[i+1][j])) + (- img[i+2][j]))
        + ((img[i][j-2]) + (2 * img[i+1][j-2]) + (img[i+2][j-2])));
   uh2  = (((-img[i][j]) + (img[i+2][j]))
         + (-(2 * img[i][j-1]))+(2*img[i+2][j-1])
         + ((- img[i][j-2]) + (img[i][j-2])));
    if ((abs(uh1) + abs(uh2)) < threshold)
      edge[i][j]=”0xFF”;
    else
      edge[i][j]=”0x00;
    }
  }
```

Figure 1. Sobel Edge detection computation example.

The computation generates for each output image pixel either a zero value “0x00” or a 1 value “0xff”. The computation uses a vertical and an horizontal gradient operator defined by a 3-by-3 pixel window around the pixel being evaluated to decide if the corresponding output pixel value should be 1 or 0.

A naive implementation of this computation onto an FPGA (or a set of FPGA over which we have partitioned the input data and computation) could use the datapath core presented in Figure 2. This datapath core follows a direct RTL translation of the set of statements in the loop body. Missing from this design are the control structures responsible for fetching and storing the data as well as implementing pipelining execution scheme. Figure 3 below illustrates the conceptual layout of the target architecture design our compiler uses to generate complete designs.

This architecture has several auxiliary control structures to the execution of the core datapath. Because our compiler targets pipelined execution techniques the datapath architecture include a simple pipeline control unit. This unit keeps track of which

1 In reality these data storage structures are delay lines where the computation can directly access any element of the line at any time. By lack of a more commonly accepted term, such as tap delay line, we use data queue.

iterations of the loop are currently in execution and generates the appropriate control signal (mainly for data fetching and storing) corresponding to the prologue and epilogue of the pipeline. The address generation unit is a programmable array address generation unit the compiler can synthesize to automatically increment/ decrement the address of references corresponding to array references in the source program. This unit is controlled by a I/O queue controller to steer the input/output data into the appropriate core datapath port or if that is the case to the input/output data queue (see Figure 3 below).

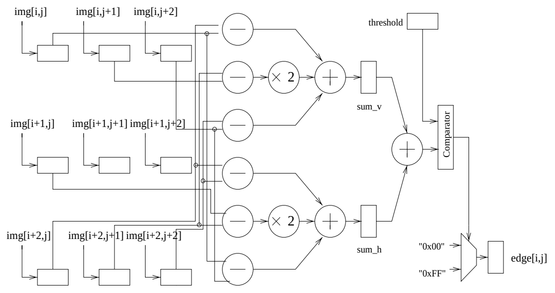

Figure 2. Naive code datapath implementation for the Sobel edge detection computation.

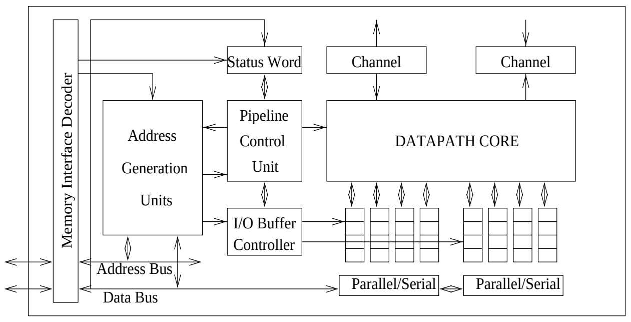

Figure 3. Target design architecture.

This example also makes apparent the advantage of exploiting the fact that consecutive iterations of the inner loop use data that previous iteration have fetched. For example 4, out of the 8 inputs values from the *img* array required for every iteration can be reuse from the values used in the previous iteration as every computation uses the pixel values in a “3-by-3” pixel window. The set of addresses generated by each array reference in the source program is very simple to define statically by the compiler - it is a simple affine index access expression. This suggests the use of data input queues to retain the values of the pixel across iteration. This strategy significantly reduces the number of memory accesses per iteration. Figure 4 presents the revised version of the datapath core for the Sobel computation example using data queues.

The auxiliary control structure required to support the execution of this revised datapath do not differ from the control structure used in the previous datapath. Less number of streams are required and fewer clock cycles per iteration. Because both our pipeline control units and address generation and I/O buffer controller are parameterizable and reprogrammable, the compiler needs to generate very little modifications to fully support the execution of both designs.

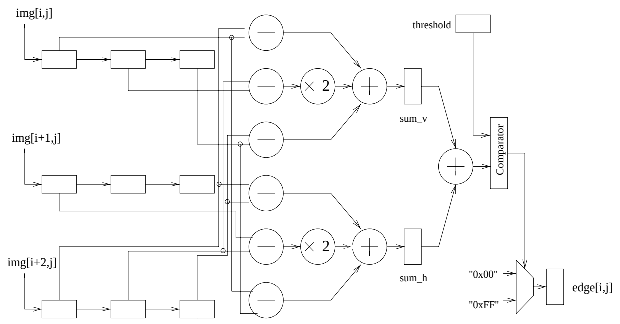

Figure 4. Sobel core datapath with input queues.

The Sobel computation example illustrates the basic hardware abstractions and compiler analysis techniques our approach captures. The computation is expressed using a tight loop nest with multi-dimensional array variables. This feature, pervasive in digital image processing kernels, allows us to use existing compiler data dependence analysis techniques to automatically extract the number of input and output data reference streams and analyze the reuse patterns of data across loop iterations. The compiler uses the memory access pattern of the array references to evaluate several possible designs by transforming the code using loop interchange and loop unrolling.

We next describe each of the hardware abstractions and then present the compiler analysis algorithm used to extract their parameters for the automatic generation of complete hardware designs using our target architecture design.

3. Basic Data Storage and Control Structures

We now describe the control and data storage structures illustrated in Section 2. In the context of our experiments we have developed code that automatically generates these structures in behavioral VHDL synthesizable using commercially available synthesis tools.

3.1 Address Generation Unit

This address generation unit (AGU) module, shown in Figure 5 is used to generate a *stream* of successive addresses corresponding to a 1D or 2D array reference using two independent index variables (e.g., a[j][i+1]), and where the index variables are incremented by a constant amount for each loop iteration.

This AGU implementation uses four kinds basic discrete components. It uses a RAM module to store the base address of the stream and the current value of the index variables corresponding to the latest memory address. For a given memory access the AGU computes the current memory address by first adding into the value of the current index a given constant value, typically 1. Next it multiplies the value of the offset by either 0, 1, or 2 (parameterizable) to account for the sizes of the data values to be fetched. Finally, the AGU controller will add the current value of the offset with the base address provided by the RAM. This AGU module is controlled by a simple FSM that stores back in the RAM the current values for the index values at the end of each cycle for utilization in subsequent memory accesses.

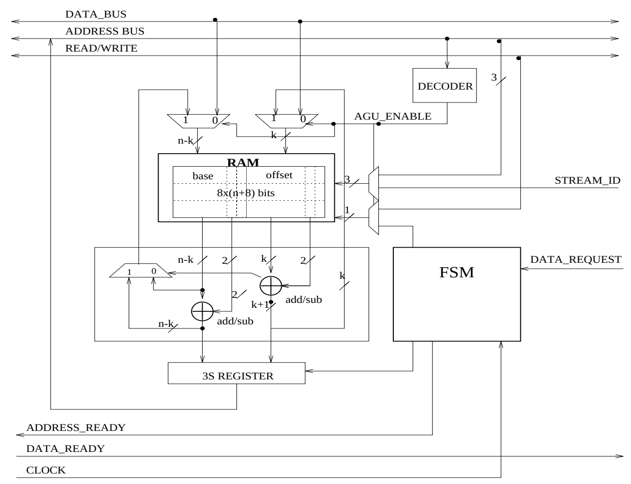

Figure 5. Address generation unit structural definition.

The AGU external interface includes a set of signals to indicate which entry is to be used in the memory access and the direction of the access (either read or write). Another set of signals defines an external entity (typically from the host processor) to write specific values to the RAM entries and therefore program the contents of the RAM entries. The concatenation option (also programmable) allows for faster memory access at the possible expense of memory in the layout of the array variables. Multiple streams for the same array might coexist in the same AGU by using distinct entries in the RAM. In the current implementations we have allocated a single AGU per memory bank in order to allow concurrent memory accesses. Clearly resources can be shared by sharing of AGU entries by multiplexing in time the utilization of the AGU at the possible expense of performance.

3.2 Queues and Window Queues

Many window-based digital image processing computations offer the possibility of data reuse by defining a “shifting” window along a given region of the arrays the computation manipulates. Different implementation variants are possible. For example, in a two dimensional domain it is possible to store the area of interest in a set of queues. Some of the queues, which we can call horizontal queues, store the values required for a given computation as the window is progressively shifted along the rows of the image, whereas another set of queues can be used to store the values of the lines of the image saving them for when the computation required the data of subsequent rows - that is the computation shifts by rows rather than by columns. Figure 6 illustrates the arrangement of “horizontal” and “vertical” queues, called here a *window queue* for the Sobel computation.

There is clearly a trade-off between the choice of a window queue and a set of either vertical or either exclusively horizontal queues. In the first option the whole design has a single data entry point and therefore a single data stream is required to fetch the successive data elements. On the other hand, multiple queues offer the possibility of parallel I/O from distinct memory banks at the expense of more entries in the corresponding AGUs.

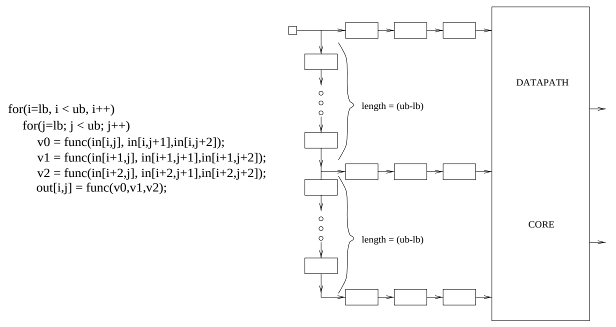
```
for(i=lb, i < ub, i++)
    for(j=lb; j < ub; j++)
        v0 = func(in[i,j], in[i,j+1],in[i,j+2]);
        v1 = func(in[i+1,j], in[i+1,j+1],in[i+1,j+2]);
        v2 = func(in[i+2,j], in[i+2,j+1],in[i+2,j+2]);
        out[i,j] = func(v0,v1,v2);
```

Figure 6. Generic window-based computation using vertical and horizontal data queues.

Yet another consideration is the ratio between the length of the horizontal and vertical queues. Longer data queues complicate the place-and-route phases of the logic synthesis tools due to the need to route longer sequences of registers further eroding the overall performance due to longer clock cycles. Our compiler currently uses Window queues when the ratio of the length of the vertical queues to the length of the horizontal queues is less than one.

3.3 Input/Output Buffer Controller

The Input/Output Buffer Controller (IBC/OBC) interfaces with the memory interface unit (MIU) unit which in turns handles the vagaries of physical memory access signals. Each I/O buffer controller receives a request for accessing a set of data streams coded in the entries of an AGU. Internally a IBC/OBC is a simple counter that signals an AGU and waits for the data to be delivered writing it to the appropriate data queue input port. Once all of the data has been fetched or stored the I/O controller signals the end of a memory cycle to a pipeline controller[2] to proceed with its computation.

If the implementation sought imposes an interface with more than one AGUs and therefore with more than one I/OBC the compiler can generate a simple AND gate combining the done signals of all of the I/OBC it needs to wait for in the current pipeline stage.

3.4 Pipelined Execution Control Unit

We use a two counter-based pipelined controller as depicted in . One counter keeps track of the iterations executed while the other of the latency of the pipeline stages. A simple FSM controls the execution of the prologue, the steady state and the epilogue of the loop execution[3]. The FSM description and the decoder combinatorial circuits are specified by the compiler when generating the control and depend on the number of stages selected

2 Clearly this is not very efficient as there could be memory access pipelined with other computation of even staggered in order to exploit specific features of the memory (e.g., ZBT - Zero Bus Turn-around capabilities of the memory). It is also conceivable that the pipeline controller interfaces with different I/O controller for accessing different memory banks. In this case the controller must wait for all I/O buffer controllers to terminate.

3 We are assuming, and the compiler will enforce it through software pipelining, that all of the stages of the pipeline have the same latency. It is not the focus of this work to deal with the specifics of software pipelining.

for the evaluation of the core of the datapath and the number of iterations of the loop. As pipelining is not the focus of this work we have used simple pipelining schemes as described in [17].

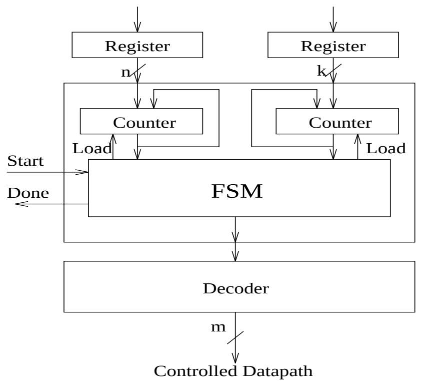

Figure 7. Pipeline Control Unit (PCU) block diagram.

3.5 Buffered Channel Units and Synchronization

These channels are routinely used for high-speed inter-board communication. Because of the difficulty to ensure appropriate timing coherency they are typically controlled using some form of signal handshaking. Figure 8 below illustrates the control for a generic unidirectional data channel.

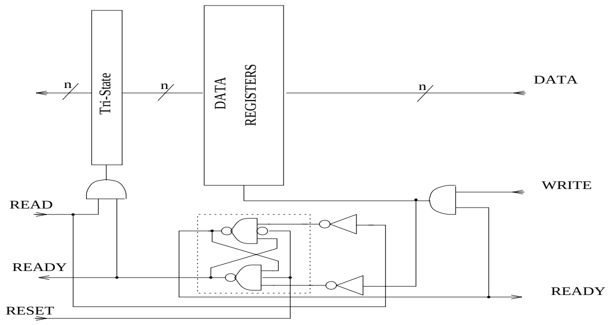

Figure 8. Unidirectional Asynchronous Buffered Channel.

4. Compiler Analysis and Algorithms

We now describe the basic analysis techniques and the algorithm our compiler uses to map computation performed in tight loop nests to hardware in VHDL. For space consideration we focus only on the automatic derivation of data queues.

Using the analysis results derived from the complier algorithm described in this section it is possible to generate code in multiple output formats depending of the capabilities of the intend target system. If for example the compiler wishes to target other behavioral synthesis compiler where for example queues are expressed in a particular paradigm the compiler can generate a representation suitable for that compiler. If, on the other hand the target systems does not provided a source level representation for a given abstraction, our compiler can generate VHDL source code that explicitly represents the intended abstraction.

4.1 Definitions and Preliminary Analysis

Our analysis is geared for perfectly nested loop with multi-dimensional array variables using affine index functions i.e., $var[f_1(i_1,...,i_n,c_1)][f_2(i_1,...,i_n,c_2)]...[f_n(i_1,...,i_n,c_n)]$, and constant loop bounds. While not all of the loops of interest are in this form, simple variations that include assignment statements between loops can be handled by moving the assignment to the inner loop and predicating its execution to the first iteration of the enclosing loops.

We also rely on previous work for the analysis and classification of loop permutable nests and data dependence analysis information as distributed in the SUIF system [12]. This preliminary analysis determines if a loop is parallelizable and which of the loops in the nest are permutable. We use other analysis included in the SUIF system release that determines which sections of a given array are read/written and which arrays are part of a reduction operations. We focus on loop with no true loop-carried dependences. Other researchers have addresses the issue of datapath core generation for loops with true dependences (e.g., [16]).

4.2 Input Data Reuse

The basic idea of data reuse steams from the fact that multiple references to the same array may access the same data items at different, or even the same, iterations of a loop. Rather than fetching the same data again from memory in most cases it would be beneficial to store the data in registers for subsequent use.

We use the definition of self-temporal reuse and group-temporal reuse from [15]. Given a reference $A[Hi+c]$ iterations i_1 and i_2 reference the same data item iff $Hi_1+c = Hi_2+c$, that is, when the vector $d = (i_1-i_2)$ is in the null-space of H. We say that there is data reuse for array A along the vector r iff $H\,d = 0$ and the vector d is in the iteration space of the loop. If, however, the loop nest has multiple references to the same array A it is possible that different iterations also reference the same data for distinct references. As before two references $A[Hi_1+c_1]$ and $A[Hi_2+c_2]$ denote the same data iff $Hi_1+c_1 = Hi_2+c_2$.

Unlike [15] our analysis requires that all of the references for a particular array A inside the loop have the same access matrix H.[4] Since our analysis generate input and output queues to store the various accesses to the array variables it has to be very precise about exactly which references are inserted at run-time into the data queues and which are not. Items already in a queue cannot be replaced (as in a cache) and so array accesses with distinct access matrices could complicate the analysis. On the other hand, and from a practical stand point, not that many computations of interest use multiple references with distinct access matrices. Even if loop unrolling transformation is applied to a single array reference in a loop by construction it is guaranteed that all of the resulting references will have the same access matrix H.

4 The case of different access matrices offers very little chance for data reuse. If the compiler applies loop permutation reuse can occur in one of the dimension along which the data is fetched in vectors. Nevertheless the points of retrieval of data inside the queue are in increasingly strides making it difficult to generate a datapath that is not a fully connected set of registers.

4.3 Quantifying Reuse and Memory Accesses

We define data reuse along a loop nest direction *d*, as the number of array references per iteration of the loop along *d* whose data values can be reuse by using the values stored in data queues. A data queue of length *n* from which the computation extracts *m* values per iteration has a data reuse of (*m/n*). High number of data reuse metrics indicate fewer accesses per iteration. Loop invariant data of size *k* are stored in registers and contribute *k* units for this metric.

Conversely we defined required memory bandwidth per iteration of the loop along the direction *d*, as the number of memory access required to fetch the data into the data queues and other input/output registers. Values that are loop invariant need only to be loaded once and are ignored in this metric.

Because digital signal processing applications typically manipulate data items of smaller bit width, the reuse direction d also exposes the opportunities for multiple data items fetched per memory access. For example, if a image pixel is encoded in 8 bits a single 32 bits memory fetch can retrieve 4 consecutive image pixels. For example, if a image pixel is encoded in 8 bits a single 32 bits memory fetch can retrieve 4 consecutive image pixels. To include this performance boost in our memory bandwidth metric we further refine the notion of memory bandwidth to include this packing factor. The memory bandwidth metric evaluation functions is shown in Figure 9.

```
double eval_bandwidth_metric(loop_nest loop_body,
    vector reuse_dir, vector layout_dir){
  double metric = 0.0;
  for all data queues q do
    b = base_ref(q);
    access = (b.access_matrix . reuse_dir);
    if(access$^T$ . layout_dir = 0)
      metric += 1.0;
    else
  if((b.data_size * b.access_stride) > mem_word_width){
        metric += 1.0;
  else
  metric += ((b.data_size*b.access_stride))/(mem_word_width);
  return metric;
}
```

Figure 9. Memory access per iteration evaluation function.

This algorithm uses the base reference for each data queue (see next Section) the array data element size and the stride of the accesses through the array references mapped to each data queue. Our compiler extracts all these basic quantities from the source program and through the data dependence analysis as explained next.

4.4 A Simple Data Reuse Algorithm

We structure the presentation of our data reuse analysis algorithm into several sections. First we describe how the algorithm uses data dependences to extract the set of possible reuse vectors and directions. Next we describe how the algorithm identifies the possible input and output data queues. At the end of this section we describe how to explore multiple designs guided by the memory access metrics described in the previous section.

4.4.1 Identifying Reuse Vectors

The basic idea of the algorithm is to use input/output data dependence analysis techniques to identify data references in the loop body that have constant distance data dependence vectors [15]. The algorithm, depicted in Figure 10, scans the data references of the loop body one array variable at a time, and computes for each pair of references for the same variable the corresponding the distance vector. It discards non-constant vectors as well as distance vectors that are not elementary (i.e., multiple of any of the axis). The algorithm arranges the data references in a matrix and uses the *DependenceTest* function in the SUIF distribution to compute the actual distance vectors between each pair of references.

The constraint of elementary reuse vectors is not a fundamental limitation, but rather an implementation and code generation convenience. While some computations such as stencil computations do exhibit “diagonal” reuse vectors code generation for exploiting the reuse is complex unless the compiler can apply loop skewing transformations to the source code.

```
{vector_list,matrix} identify_reuse_vectors(loop loop_body,
    vector DependenceTest(), array_var var){
  vector_list = new list;
  dependence_matrix = new matrix;
  for all data references dr1 of var in loop_body do
    for all data references dr2 of var in loop_body do
      vec = DependenceTest(dr1,dr2);
      if(vec is elementar_vector()) {
        vector_list += {vec};
        dependence_matrix(dr1,dr2) += {vec};
      }
    end
  end
  return {vector_list,dependence_matrix};
}
```

Figure 10. Extracting data reuse vectors algorithm.

4.4.2 Identifying Data Queues

Given the data dependence distance vectors in the dependence matrix the algorithm next computes for a given array variable and for a given direction vector the independence set of maximally connected references.

```
{queue_list} identify_data_queues(matrix dep_matrix,
      array_var var, vector reuse_direction){
  queue_list = new list;
  for all data references pair (dr1,dr2) in dep_matrix do
    if(dr1 unmarked and dr2 unmarked)
      dq =  new queue; queue_list += {dq};
      dq += {dr1, dr2}; mark dr1; mark dr2;
    if(dr1 marked and dr2 unmarked)
      dq = data_queue(dr1); dq += {dr2}; mark dr2;
    if(dr1 unmarked and dr2 marked)
      dq = data_queue(dr2); dq += {dr1}; mark dr1;
    else
      merge_data_queues(data_queue(dr1),data_queue(dr2));
  end
  return queue_list;
}
```

Figure 11. Algorithm for identifying data queues.

Two references dr_1 and dr_2 are connected along the elementary reuse direction e_i iff there exists a data dependence vector $v = k\, e_i$ relating dr_1 and dr_2. A data queue is defined as the connected set of references that share the same distance vector between references. The base for each data queue is the reference for which there is no entry *(*,dr)* in the data dependence matrix. Figure 11 presents this analysis where we use a simple marking algorithm to determine if a given reference has been assigned to a particular data queue. Once identified the base of a data queue the compiler can extract the base address, offset and stride for that data queue.

4.4.3 Exploring Possible Implementations

The compiler uses the algorithms described previously to generate a set of possible implementations one for each of the identified reuse vectors. The compiler first extracts the set of possible reuse vectors by inspection of the entries in the data dependence matrices for all of the array variables. Next the compiler creates the set of possible data queues for each of the direction vectors and evaluates their performance using the data reuse and memory access metrics. Our compiler currently chooses the version with the lowest aggregate memory access metric.

4.5 Implicit Loop Unrolling Data Reuse Analysis

Unfortunately not all of the loops have the data references exposed as in the Sobel example in Section 2. In some cases the computation is expressed in a compact form using a deeply loop nest with a single reference per array variable rendering innefective the analysis described above.

To address this shortcoming we have developed an analysis that examines the effects of loop unrolling on the set of generated array references and extracts the possible set of data queues for the unrolled data references. Because this analysis does not actual unrolled every loop it checks in an explicit fashion, but rather examines its implications on the set of generated references, we call this an implicit loop unrolling data reuse analysis. The main advantage of this implicit unrolling strategy is that our compiler does not have to incur in the space, and therefore time, costs of doing explicit loop unrolling, in particular when the loop bounds are large. In addition this implicit unrolling approach provides a handle when the loop bounds are unknown and/or when the compiler wishes to do partial unrolling.

We now describe the analysis algorithm the compiler uses to determine which of the loops should be unrolled and which should be exchanged to expose the maximal data reuse. We make the assumption (and this is often the case in practice) that the array data references inside the loop have the property that every loop index variable can be present in at most mode one array dimension. We call this property index orthogonality and greatly simplifies the analysis and determination of reuse directions. The outline of the algorithm is illustrated in Figure 12 below.

```
for each legal unrolling l in L do
  for every data reference dr in DR do
    if(dr.am.$\overline{unroll}$ = 0) then
      $list_{loopinv}$ += {dr};
  for every data reference dr in {DR - $list_{loopinv}$} do
    for every non-zero entry i of l do
      $v_{proj}$ = dr.am.$e_i$;
       if($v_{proj}$ != 0 and $l_i$ . $e_i$ != 0) then
       // vproj by definition is either zero or unit basis vector
        $v_{coupled}$ =  v in loop such that dr . am . v = $v_{proj}$
      if($v_{coupled}$ != 0) then
        reuse along $v_{coupled}$;
      queue layout = mask(dr.am.l);
  }
}
```

Figure 12. Data reuse algorithm with implicit loop unrolling.

The algorithm explores all possible legal combinations of loop unrolling. For each loop unrolling it classifies the array references as loop invariant after unrolling or as potentially exposing reuse along a subset of the remaining rolled loop dimensions. If a given reference is loop invariant there will be reuse carried along any of the rolled loops. The algorithm next concentrates on the remaining loop variant references. First the algorithm determines the set of projected dimensions as the set of array dimensions whose indices vary with the set of unrolled loops. Because of the orthogonality only a single dimension varies with a single unrolled loop. Next the algorithm determines the rolled loop dimension that is couple with the same unrolled dimension that projects onto the same array dimension. This couple dimension will be the loop along which reuse will be carried as successive unrolled data references have replicas of the reuse loop index variable when unrolling is applied. For each reference the algorithm collects the set of reuse vectors for each of the loop unrollings. In the last step the algorithm determines for each of the reuse directions which of the data references exhibit reuse along a given reuse direction. The algorithm also computes the dimensions and shape of the queue to include the unrolled data references. Figure 13 illustrates the various algorithm results for a window-based correlation algorithm and for a particular loop unrolling vector *(m,n,i,j) = (0,0,1,1)*.

```
for(m=0; m < M; m++){                     for(n=0; n < N; n++){       // Loops m and n
  for(n=0; n < N; n++){                     for(m=0; m < M; m++){ // Interchanged
    sum = 0;                                  sum = 0;
    for(i=0; i < I; i++)                      .
      for(j=0; j < J; j++)                    .   loop body unrolled (I x J) times
        if(mask[i][j] != 0)                   .
          sum += array[m+i][n+j];             .
    res[m][n] = sum;                          res[m][n] = sum;
  }                                         }
}                                         }
```

a. Original source code. **b. Transformed source code**

```
DR = {dr1 = mask[i][j], dr2 = array[m+i][n+j]}
unroll vector l = [0011];
dr1.am = [0010|0],[0001|0];
dr2.am = [1010|0],[0101|0];

$list_{loopinv}$ = {dr1} = {mask[i][j])}
dr = dr2 = array[m+i][n+j]
case   ei = [0001]                             case  ei = [0010]
       vproj = [01]                                  vproj = [10]
       vcoupled = [0100]                             vcoupled = [1000]
       reuse = [0100] // along loop n                reuse = [1000] // along loop m
       layout = dr2.am.[0011] = [* *]                layout = dr2.am.[0011] = [* *]
```

Figure 13. Implicit loop unrolling analysis example.

Because the number of possible loop unrollings grows exponentially with the depth of the loop nest we have limited the number of explored loop unrolling by considering only legal unrollings that lead to one or more of the array variables to become loop invariant after unrolling. Clearly future compiler analysis implementations should also consider the impact on the resources by loop unrolling. Given the results of the analysis the compiler next evaluates each of the candidate solutions based on the memory access metric described in Section 4.3.

4.5.1 VHDL Code Generation

Given a loop nest a selected reuse direction and corresponding data queue implementation, the compiler uses a set of predefined control structures to generate and program the control structure described in Section 3. To generate the datapath corresponding to the statements in the loop body our compiler traverses the abstract syntax trees in SUIF generating an internal datapath representation where identical scalar references and data values are mapped to registers. The current implementation handles limited set of constructs in C and uses a simple pipelining implementation guided by the set of *assign* statements in the loop body. In the current implementation we have ignored the issues related to bit widths and optimizations related to using small than the predefined bit width for convenience purposes. Exploring these optimizations is orthogonal to the work described here and can be easily composed with the current implementation.

The current implementation of our compiler generates VHDL specifications for the control structures, and data queues. these structures are instantiated using predefined templates. Each structure is described using a behavioral or in some cases by structurally composing behavioral components using the "port map" VHDL language construct.

While the current implementation works for a specific vendor logic synthesis, we have neither relied nor exploited any vendor specific VHDL style or library implementations. Clearly the data gathered by the compiler analysis described here would allow the compiler to exploit FPGA library component features by adequately choosing among a myriad of possible implementations customized for each of the targets architectures.

5. Experimental Results

We now present experimental results for the compiler analysis and semi-automated compiler generated designs. We are currently unable to fully automatically generate complete designs using a set of library functions we have developed. Instead we use the information provided by the compiler to a file to feed a code generation program manually and merging several VHDL files for the complete implementation of a design.

5.1 Methodology

We used a set of complete applications written in C for the evaluation of the compiler analysis and compiler generated designs. The applications are compiled using SUIF v1.1.2 *scc* distribution tool. We then used the compiler algorithms described in Section 4 (approx. 8,000 lines of C++) to analyze and select a particular implementation for the loop nests of interest in each application. We then used the results of the compiler analysis to generate VHDL files using a set of predefined template generation functions (approx. 10,000 lines of C code). We then merged the VHDL files by hand and used the Xilinx Foundation Series V2.1i logic synthesis tools to generate logic design data and bitstream files.

We report results for the logic synthesis running on a Pentium II PC running at 450 MHz and with 128 Mbytes of memory for the generation and evaluation of the compiler generated designs. In our synthesis experiments we used the Virtex 1000BG560 FPGA series and used a low effort and optimized for speed place-and-route (P&R) settings. The timing results we report are extracted from the generated log files with timing analysis with complete (100%) path coverage.

5.2 Applications

5.2.1 Sobel Edge Detection - Sobel

This application implements the Sobel edge detection algorithm over a 64-by-64 gray scale image. The application uses two 2D array, one to store the image and another to store the results of the computation. The core of the computation is performed in a doubly nested loop. At each loop iteration the computation uses a 3-by-3 window of image pixels to compute vertical and horizontal gradient values. Using these two metrics the algorithm decides to assign either a '1' value or a '0' value to the result array pixel.

5.2.2 String Pattern Matching - Pattern

This application performs a simple character by character matching of patterns against substrings in a given string of characters. Th original code is as depicted in Figure 14. It scans the *pattern* array variable repeatedly and compares it against shifted versions of the array *str* variable. If at least one of the characters differs the corresponding result of the matching is set to the value 0.[5]

```
for(i=0; i < STRING_SIZE-16; i++){
  res[i] = '1';
  for(j = 0; j < 16; j++){
    if(pattern[j] != str[i+j]){
      res[i] = '0';
      break;
    }
  }
}
```

Figure 14. String pattern matching computation.

5.2.3 Automatic Target Recognition - ATR

This application performs matchings between a given template and windows of an gray scale image. The basic computation consists of binary image correlations between the template matrix and shifted windows over the input image as illustrated in Figure 15.

```
for(m=0; m < IMAGE_SIZE-MASK_SIZE; m++)
  for(n = 0; n < IMAGE_SIZE-MASK_SIZE; n++){
    sum = 0;
    for(i=0; i < MASK_SIZE; i++)
      for(j=0; j < MASK_SIZE; j++)
        if(mask[i][j] != 0)
          sum += image[m+i][n+j];
    res[m][n] = sum;
    }
```

Figure 15. Shifted window binary correlation computation.

5.3 Results

We now describe the performance results of our code generation strategy and quantify the simulated performance of the resulting generated designs. We begin this discussion by presenting the compilation and synthesis metrics.

Table 1 presents the compilation and synthesis results. For each of the tested applications we report the number of source code lines for both C and the VHDL generated codes (excluded comments and

5 We have also implemented an analysis capable of recognizing the presence of "break" and "continue" statements in typical search and matching computations. This analysis coupled with the data reuse analysis allows a compiler to eliminate the break statement in this loop without changing the semantics of the computation.

blank lines). We report the number of loop nests in each application and the number of loops the compiler selected for hardware execution. For the generated VHDL source code we report on its size, the number of distinct components and instances used. Finally we report on the compilation analysis and synthesis speed.

	Source Code Metrics			VHDL Code Metrics			Analysis & Synthesis Time		
App	Code Lines	Loop Nests	Loop Hard	Code Lines	Num Comps	Num Inst	Analyzes Time	Emit Time	Synthesis Time
Sobel	80	3	1	2,340	39	134	< 1 sec	< 1 sec	10 min
Pattern	98	4	1	2,445	32	111	< 1 sec	< 1 sec	8 min.
ATR	300	5	3	4,400	38	386	< 1 sec	< 1 sec	780 min.

Table 1: Compilation and synthesis results.

Table 2 presents the results of the compiler analysis. For each application we report on the number an length of the data queues the algorithm has identified and the unrolling vector that has the lowest memory access metric.

Application	Unrolled Loops	Reuse Vector	Data Queues	Length Data Queue	Data Reuse	Mem Band
Sobel	-	(x,y) = (0,1)	3	3	6	1.0
Pattern	{j}	(i) = (1)	2	16	31	0.5
ATR	{i,j}	(m, n) = (0,1)	2	32-by-32	2016	8.25

Table 2: Data reuse analysis results.

For the *Sobel* application the compiler recognizes the opportunities of two reuse directions. Because the current compiler implementation chooses the implementation with the lowest memory bandwidth we therefore report only on the performance results of the selected version of Sobel. As for *Pattern* the analysis recognizes that unrolling loop *j* is high profitable as the *pat* variable becomes loop invariant. The resulting implementation should have a single queue of length 16 for the *str* variable and another queue of the same length for the *pat* variable. Finally the *ATR* application has 3 loops in which there is a substantial amount of reuse. We present only the results for the loop illustrated in Figure 15. For this loop the analysis selects the unrolling of loop *i* and *j*, which reveals a maximum reuse for a 32-by-32 queue for the *mask* variable which is loop invariant and a 32-by-32 queue for the *img* variable. Unfortunately the design corresponding to the full unrolling of the two inner loops is too large to fit on a single FPGA. Instead we partially unroll each of the two inner loop by a factor of 16, therefore creating 16-by-16 data queues. This consumes less FPGA resources but significantly increases the control complexity and therefore the simulated execution time.

Table 3 shows the simulated performance results for the generated designs. It includes the overall simulated clock speed, the number of flip-flops and latches used as well as the number of LUTs and equivalent gates counts. For raw performance comparison we included the number of execution cycles required to complete the entire loop nest computation of each application. Finally we report on the area of the Virtex1000 reported used by the P&R tool.

The table shows the three designs to attain respectable clock rates for automatically derived designs. Recall these designs were generated automatically by manually using the results of the analysis with the library of code generation functions we have implemented using generic, and simple, parameterized modules (e.g, adders, sub, comparators, multiplexors, etc.). These results reveal the compiler is able to identify the opportunities for data reuse and generate the data required to automatically generate a complete VHDL design. Because of their relative small size, the generated designs for *Sobel* and *Pattern* are synthesized and routed fairly quickly. The design corresponding to the ATR application uses 55% of a single FPGA resource and takes much longer to synthesize (even with hierarchical P&R). We attribute this discrepancy to the PC memory trashing effects.

Application	Core Datapath Clock (MHz)	Global Design Simulated Clock (MHz)	Number FF & Latch	Number LUTs	Equiv. Gates	Simulated Execution Cycles	Virtex1000 Area
Sobel	56.5	26.5	840	727	11,375	2,196,480	5%
Pattern	56.5	26.0	782	771	11,239	287	5%
ATR	52.5	25.7	9,163	9,649	145,759	182,272	55%

Table 3: Simulated target designs performance metrics.

The performance results (simulated clock rates) also reveal that the limiting factor is the control datapath as the core datapath are capable of much higher clock rates. Several factor contribute to this. First the generality of the modules used. As an example our memory access subunit is fairly generic as it can handle both SRAM as DRAM modules. This clearly introduces latency in terms of clock cycles. Our interfaces allow for the presence of multiple memory interface module for distinct memory banks with a common pipelining control unit. Several other design aspects have not been explored in this paper as our focus was the design of a compiler analysis algorithm to allow the automatic generation of hardware implementations. We have not explored the trade-offs in the design of the pipelining control unit and the possible refinements of pipelined memory access unit. As such our simulation performance results can be improved by using control units with more advanced features.

6. Related Work

Several research efforts have concentrated on the development of new reconfigurable architectures. The RaPiD [6] architecture is a coarse-grained field-programmable architecture composed by multiple functional units such as ALUs, multipliers, registers and RAM blocks. These units are organized linearly over a bus and communicate through registers in a pipeline style. As with other efforts the authors have develop a specific extension to the C programming language that exposes some of the architecture to the programming model hence simplifying the compilation process.

At CMU researchers developed the PipeRench [10] architecture geared solely to pipelined computations with virtually unlimited number of stages. Programmers rely on compilers to map their applications, written in C, to partition their computation to the computational capabilities of each strip. The compiler then generates a schedule of the virtual strips and relies on hardware to swap in and out the configuration for each of the strips on demand.

The MIT RAW [3] is a coarse-grained mesh architecture where each of the computing core has a simple controller, a set of registers, a local RAM block and programmable communication channels. The compiler partitions the computation and data among the cores and programs the communication channels to best suite the communication pattern for each application.

At Berkeley researchers integrated a MIPS core with reconfigurable hardware to be used as an accelerator in a co-processor integration model [19]. The reconfigurable hardware consists of a set of 2D arrays of CLBs interconnected by programmable wiring. Each of the array has a section dedicated to memory interfacing and control and uses a fixed clock. The reconfigurable array has direct access to memory for fetching either data or configuration data hence avoiding both data and reconfiguration bottlenecks.

These efforts differ from our research in two main aspects. First and as new architectures, these efforts have chosen which components of the systems are reconfigurable and what are the macro instructions the non-reconfigurable portion can execute. In our case we are given a set of FPGAs and an external interface and have to synthesize from the ground up all of the control structures in the FPGA to allow them to operate autonomously. Because these approaches do not use commercial synthesis tools they avoid the performance and interface issues with place-and-route.

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 [17] 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 [7,8] explores the problem of automatic mapping array variables to memory banks. This work in 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.

7.0 Conclusions

We described a compilation and synthesis scheme for the automatic mapping of loop nests that access multi-dimensional arrays to hardware implementations. We have implemented a compiler analysis algorithm capable of automatically generating efficient implementations for these loop nests. This algorithm exploits the data reuse and array memory layout to minimize the number of required memory accesses. The simulation experiments, although limited, suggest the code generation approach combined with our compiler analysis can be used as a basis of a successful compilation and synthesis approach for digital image processing computation onto FPGA-based computing engines.

Bibliography

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

[2] J. Arnold, “The Splash 2 Software Environment”, In Proc. IEEE Symp. on FPGAs for Custom Computing Machines (FCCM ‘93), IEEE Computer Society Press, Los Alamitos, 1993, pp.88-93.

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

[4] P. Bellows and B. Hutchings,“JHDL - An HDL for Reconfigurable Systems”, In Proc. IEEE Symp. on FPGAs for Custom Computing Machines (FCCM’98), IEEE Computer Society Press, Los Alamitos, 1998, pp. 175-185.

[5] M. Chu, N. Weaver and K. Sulimma, “Object-Oriented Circuit Generation in JAVA”, In Proc. IEEE Symp. on FPGAs for Custom Computing Machines (FCCM’98), IEEE Computer Society Press, Los Alamitos, 1998, pp. 158-165.

[6] 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 (FCCM’98), IEEE Computer Society Press, Los Alamitos, 1998, pp. 116-125.

[7] 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 (FCCM’99), IEEE Computer Society Press, Los Alamitos, 1999, pp. 63-69.

[8] M. Gokhale and J. Stone, “Napa C: Compiling for a Hybrid RISC/FPGA Architecture”,In Proc. IEEE Symp. on FPGAs for Custom Computing Machines (FCCM’98), IEEE Computer Society Press, Los Alamitos, 1998, pp. 126-135.

[9] M. Gokhale and B. Schott, “Data Parallel C on a Reconfigurable Logic Array”, Journal of Supercomputing, 9(3):291-313, 1994.

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

[11] J. Hauser and J. Wawrzynek, “Garp: A MIPS Processor with a Reconfigurable Coprocessor”, In Proc. IEEE Symp.on FPGAs for Custom Computing Machines (FCCM’97), IEEE Computer Society Press, Los Alamitos, 1997, pp.12-21

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

[13] J. Villasenor, B. Shoner, K. Chia and C. Zapata, “Configurable Computing Solution for Automatic Target Recongition”, In Proc. IEEE Symp. on FPGAs for Custom Computing Machines (FCCM ‘96), IEEE Computer Society Press, Los Alamitos, 1996, pp. 70-79.

[14] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. “Baring it all to Software: RAW Machines”, IEEE Computer, Sept. 1997, pp. 86-93.

[15] M. Wolf and M. Lam, “A Loop Transformation Theory and an Algorithm for Maximizing Parallelism”, IEEE Trans. on Parallel and Distributed Systems, Oct., 1991.

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

[17] M. Weinhardt and W. Luk., “Pipelined Vectorization for Reconfigurable Systems”, In Proc. IEEE Symp. of FPGAs for Custom Computing Machines (FCCM’99), IEEE Computer Society Press, Los Alamitos, 1999, pp. 52-62.

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