A VHDL Design Optimization for Two-Dimensional Filters

Nelson Luiz Passos  Jian Song  Robert Light  Ranette Halverson  Richard Simpson
Department of Computer Science
Midwestern State University
Wichita Falls, TX 76308

ABSTRACT

Multi-dimensional applications, such as image processing and seismic analysis, usually require the high computer performance obtained from the implementation of Application Specific Integrated Circuits (ASICs). The critical sections of such applications consist of nested loops with the possibility of embedded conditional branch instructions. Current commercial systems use branch predication techniques, which can also be applied in the design of ASIC systems. Those techniques utilize predicate registers to control the validity of computed results. The optimized design and allocation of such registers becomes then a significant factor in the performance of the system. By using branch predication to transform control dependencies in data dependencies, the application of a multi-dimensional retiming to an MDFG permit the iterations of the original loop body to be naturally overlapped, making the existent parallelism explicit. Based on the retiming information, predicate registers are designed as shift registers that allow the correct execution of the filter function.

Keywords: multidimensional retiming, branch predication, instruction level parallelism, filters, edge detection

1. INTRODUCTION

Image enhancement and edge detection are well-known digital image signal processing applications that may require two-dimensional (2D) filter-like computational solutions. These applications usually depend on computation intensive sections, consisting of the repetition of sequences of operations, which may include conditional branches [12]. An effective technique in improving the computing performance of such applications has been the design and use of ASICs [4]. Multi-dimensional retiming techniques can be used to improve the instruction level parallelism of uniform loops. Non-uniform loops that depend on conditional instructions may degrade the performance gain achievable through the retiming technique. Branch predication techniques utilize predicate registers to control the validity of speculatively computed results, hiding the existence of conditional statements. This paper presents a VHDL design of a 2-D filter system, including predicate registers, while optimizing the computational time of the loop, and reducing the overall execution time of the filter function.

Most of the previous results on scheduling loops are solely based on one-dimensional uniform problems, which do not consider the effects of conditional branches [1,3,5,13,16,20]. Other scheduling methods were developed for multi-processor architectures and are not applicable to the problems covered in this paper [6,7,8,11,14,15]. During the circuit design phase, nested loop structures can be coded using hardware description languages, such as VHDL constructs, in order to reduce the design time. However, in VHDL, the loop control indices will represent the number of times a section of the circuit will be replicated in the final synthesis [10]. This study focuses on the parallelism inherent to multi-dimensional applications involving branch decision, ignored by the one-dimensional methods [2,9], emphasizing the parallel execution of nested loops iterations found in the VHDL implementation of 2-D filters.

In previous studies, the authors introduced the concept of multi-dimensional retiming, which allowed the restructure of the loop body represented by a general form of multi-dimensional data flow graph (MDFG), while preserving data dependencies [16,17,18]. In an MDFG, the nodes represent the loop body operations and the edges, the data dependencies between operations. By associating each execution instance of the loop with an integral index in a Cartesian space, it is possible to compute a multi-dimensional distance between the production and consumption of each data value. By using an MDFG to represent the loops required to process the two-dimensional filter function, we allow the application of a multi-dimensional retiming to that MDFG making the iterations of the original loop body to be naturally overlapped and the existent parallelism explicit. However, this technique may require the use of a new schedule vector, i.e., a new execution order of the loop iterations that directly affects the loop bounds and indices. The proposed implementation approach allows the designer
to specify the scheduling vector in such a way to improve the computational time of the loop. Memory is assumed to be local, such that it can be accessed in a speed compatible with the optimized processor design. This new execution sequence is then implemented on an address generator mechanism, computed in pre-compile time, in order to provide the correct index values, according to the schedule. This paper presents the new VHDL construct, how the decomposition is done, and the theoretical aspects of the address generator.

Some 2-D filters, such as edge detectors, may have conditional statements embedded in the target loops. Branch predication techniques use if-conversion methods to transform the conditional code in straight-line code [22]. The new code uses Boolean guards to implement the conditional execution of individual operations. Such a transformation reduces existing control dependencies to more common data dependencies. However, it requires the use of special registers, commonly named predicate registers to store the Boolean values. This paper reports a new method for scheduling cyclic MDFGs with an infinite number of non-pipelined processing resources and predicate registers available and their VHDL experimental implementation. The resulting schedule length is associated with the number of control steps, i.e., the clock cycles of the circuit design, required to execute all operations in the loop body. Considering the assumption that each single operation in the loop can execute in one time unit, such schedule can be reduced to one time unit. Such a technique is implemented through a polynomial time scheduling algorithm, resulting from a modification to the chained retiming algorithm presented in [17].

Next section presents an overview of the addressing mechanism required by the multi-dimensional retiming technique. It also discusses the branch predication technique, followed by a brief presentation of the basic fundamentals of the multi-dimensional scheduling method. The VHDL design is developed next. A simple example of an edge detector demonstrates the method’s efficiency. A summary finalizes the paper.

2. BACKGROUND

In this section, we present some concepts used to model multi-dimensional problems. We also discuss the problems of using VHDL to describe the filter solution. We begin by focusing our attention in the execution sequence of the problem.

The schedule vector

The execution of each operation in the body of a nested loop, exactly once, represents an iteration. Iterations are identified by a vector \( i \), equivalent to a multi-dimensional index, starting from \( (0,0,...,0) \). The Cartesian space, whose integral points represent the iteration indices, is called iteration space. The iteration space is bounded by the dimensions of the problem that it represents. In the case of multiple nested loops, such bounds are the loop indices bounds. This paper focuses on two-dimensional rectangular iteration spaces. However, the concepts presented can be applied to 3 or more dimensions, which was not currently implemented due to limitations on the synthesis tools.

A common loop statement in any programming language has a pre-established execution order. Looking at the iteration space representing such a loop, the standard execution sequence for a two-dimensional problem runs left-to-right in a row-wise direction and progress from one row to the next row, as seen in figure 1(a). A schedule vector \( s \) is the normal vector for a set of parallel hyperplanes that define a sequence of execution of the iteration space. We call a local schedule the direction indicating the execution sequence of iterations along a hyperplane. A global schedule is the direction of the schedule vector. In order to keep an equivalence with regular patterns of loop execution, we assume that the local execution sequence, i.e., the execution of the loops along the scheduling hyperplane, will follow a left to right order with respect to the global schedule. We say that the first iteration to be executed in any hyperplane is the leftmost iteration, represented by \( L_{\text{left}} \), for the leftmost iteration of hyperplane \( p \). For the row-wise execution, the local schedule follows the direction \((1,0)\), and the global schedule is equivalent to the direction \((0,1)\) in the Cartesian space, as seen in figure 1(a). The rows are then considered the scheduling hyperplanes, and \((0,1)\) is the schedule vector. Figure 1(b) shows the execution sequence for a global schedule \((1,1)\) and a local schedule \((1,-1)\).

VHDL loop semantics
When using loop constructs in a VHDL behavioral code, the designer must have in mind that the VHDL process statement has an infinite loop semantics, and must be understood as an infinite outmost time loop [10]. However, in multi-dimensional problems, not all loops are related to the time dimension. Most of the image processing applications require transformations done in the space directions and the process statement is not the most appropriate to represent such requirements. At this point, we assume that along the time dimension, new images will be input to the circuit, and stored for processing. Therefore, there is no mandatory sequence in the image data scanning, which could be done row-wise, column-wise, or in any other sequence.

![Figure 1. Iteration space (a) under row-wise execution (b) under a global schedule vector (1,1)](image)

An alternative to the use of a process statement, representing the repetitive execution of the computation, is the loop command. In general, this is combined with a for constraint, in order to describe the boundaries of the problem. Unfortunately, as mentioned before, this construction is synthesized as a replication of the operations included in the loop body. One may say that the synthesized solution is the smartest one. However, for large number of instructions in the loop body, that may not be true.

In order to avoid the unnecessary replication (or to optimize the final design), the designer must ignore the original description of the problem, i.e., a natural nested loop situation, and re-think the solution in a totally different scope. In general, it will be necessary to imagine the loop body as an individual component and start designing control sections and other widgets to execute the multi-dimensional loop. In this paper, we propose to reduce the design time by providing to the designer a specific construct for multi-dimensional loops description.

**Multi-dimensional retiming**

A multi-dimensional data flow graph G consists of a tuple \((V,E,d,t)\), where \(V\) is the set of computation nodes, \(E\) represents the set of dependence edges, \(d\) is a function representing the multi-dimensional delays between two nodes, and \(t\) is a function representing the computation time of each node. The earliest starting time for the execution of node \(u\), \(ES(u)\), is the first scheduling control step following the end of the execution of all nodes predecessors of \(u\) by a zero-delay edge. This can be represented as: \(ES(u) = \max \{ 1, ES(v) + t(v)\} \) for all \(v\) preceding \(u\) by an edge \(e\), such that \(d(e) = (0,0,...,0)\). A schedule vector \(s\) determines the sequence of the execution of the iterations. An MD retiming \(r\) redistributes MD delays in an MDFG \(G\), creating a new MDFG \(G_r=(V,E,d_r,t)\). A retiming vector \(r(u)\) applied to a node \(u \in V\) represents the offset between the original iteration containing \(u\), and the one after retiming. Such a vector represents MD delays pushed into the edges \(u \rightarrow v\), and subtracted from the edges \(w \rightarrow u\), where \(u, v, w \in V\). The chained MD retiming technique [13] is one of the possible methods able to compute a legal MD retiming of an MDFG, and to produce a fully parallel graph. This technique is characterized by three important properties:

1. A legal MD retiming \(r\) of a node in an MDFG \(G\), with all its incoming edges having non-zero delays, is any vector orthogonal to a schedule vector \(s\) that realizes \(G\).
2. If \(r\) is a MD retiming function orthogonal to a schedule vector \(s\) that realizes \(G= (V,E,d,t)\), and \(u \in V\), then \((k\times r)(u)\) is also a legal MD retiming.
3. The chained MD retiming algorithm transforms \(G\) to \(G_r\), such that \(G_r\) is realizable and fully parallel.

**Branch Predication**
Predicated execution, also called guarded execution, provides the means to prevent the performance loss generated by the speculative execution of instructions in processors with multiple functional units, following a conditional branch command. Under this technique, an instruction is executed based upon a Boolean value called predicate. If the predicate is false, the instruction results are nullified, not changing the processor state. The predicated execution combined with if-conversion allows the removal of branch instructions from the instruction stream and consequent elimination of control dependencies, which are automatically transformed in data dependencies. As a simple example, let us consider the code below:

```plaintext
if A > 5 then B = 2
else B = 3
endif
```

The use of the if conversion technique would need a Boolean guard $p$, which receives the result of the test of the condition $A > 5$. In the predicated execution, $p$ is associated to each of the two instructions assigning new values to $B$. After $p$ is calculated true or false, then one of the assignment instructions is discarded while the other is completed. The code below shows the predicated format for this example:

```plaintext
p = A > 5
B = 2 | p /* B=2 if p == true */
B = 3 | ¬p /* B=3 if p == false */
```

In such a situation, the two instructions assigning new values to $B$ could be fetched and partially executed in parallel with the decision process. However, when such a condition is found within a loop, the overlap of iterations may not be possible due to the need to reuse the predicate register $p$.

### 3. CONTROLLING THE EXECUTION SEQUENCE

In order to satisfy the designer need of an user-friendly tool, this study describes the behavioral description of a multidimensional application by a new loop command, proposed in [19], that scans the iteration space according to an user defined global schedule. One of the most important points on this implementation regards the correct index generation of each iteration, therefore we begin by reviewing the theoretical aspects of such topic.

**An infinite iteration space**

As we have seen before, the execution of the iteration space, according to a global schedule coincident with one of the Cartesian axis, or even with the direction $(1,1)$ seems to be of trivial implementation. However, since this study contemplates a more general solution, the execution order must satisfy any possible schedule vector. Let us assume, for now, an iteration space that has no bounds in the x- or y-direction. We begin by examining some straightforward properties [19] of the new execution sequence for a given two-dimensional iteration space and a global schedule $s=(s_x,s_y)$:

1. The local schedule, represented by $s^j$, follows the direction $s^j=(s_x,s_y)$.
2. An iteration $t=(i,j)$ belongs to the hyperplane $p$, where $p=t \cdot s$.
3. After an iteration $t=(i,j)$ has been executed in a hyperplane $p$, the next iteration must be either $t+s^j$, or the leftmost iteration on hyperplane $p+1$.
4. If the leftmost iteration on hyperplane $p$ is iteration $L_{h=p}$, $(0,j)$ such that $p=j \times s_y$ and $s_y \neq 0$, then iteration $(j+1)$ will be the leftmost iteration on hyperplane $p+s_y$, i.e., $L_{h=p+s_y}$.
5. If $(i,j)$ is the leftmost iteration in a hyperplane $p$ and $(i',j')$ is the leftmost iteration in a hyperplane $p+I$ then $\gcd(abs(i'-i),abs(j'-j))=1$.

By examining such properties, if iteration $(0,j)$ belongs to hyperplane $p$, it is noticeable that $L_{h=p+s_y}$, where $p<k<p+s_y$ may not be in the y-axis. Usually, this new iteration is computed through the use of integer conversions, mod functions, and so on.
Let us examine a simple example consisting of the iteration space shown in Figure 2, being executed according to a global schedule (2,3), and local schedule $s^\perp=(3,-2)$. Iteration (0,2) is the leftmost iteration of hyperplane 6. Iteration (0,3) is the leftmost iteration of hyperplane 9. After executing (0,2), the next iteration will be (3,0). After (3,0) is done, the next iteration is in hyperplane 7 and it is (2,1). Using the properties mentioned above, a modified version of the Euclid’s algorithm for GCD computation, shown below, is used to compute the $L_{h=ap}$, given $L_{h=ap}$. The algorithm assumes $L_{h=ap}=(0,0)$, and a schedule vector $s=(s_x,s_y)$.

**Algorithm LIRA**

```plaintext
/* compute the leftmost iteration of next hyperplane */
a ← max(s_x,s_y) ; b ← min(s_x,s_y)
(x,y) ← FindIt(a,b)
if (x ≥ s_y) /* if out of range adjust the values */
  (x,y) ← (x,y) - s^⊥
endif
if (s_x<s_y) /* invert the results if s_x<s_y */
  (x,y) ← (y,x) + s^⊥
endif
if (x ≥ s_y) /* if out of range adjust again */
  (x,y) ← (x,y) - s^⊥
endif
end LIRA
```

**Function FindIt**(a,b)

```plaintext
k ← floor (a/b) ; a ← mod (a,b)
if (a=0)
  (x,y) ← (k,0)
  return x,y
endif
(x',y') ← FindIt(b,a)
(x,y) ← (k*x' + y',k*x')
return x,y
```

To demonstrate the application of this algorithm, let us compute the leftmost iteration of hyperplane 7 in Figure 2. In the first stage of the algorithm, for $s_x=2$, and $s_y=3$, we have $a=3$ and $b=2$. The function FindIt begins with $k=1$ and $a=1$. Proceeding with the first recursion, $k=1$ and $a=0$. At this point we begin the backtrack processing, producing $x=1$ and $y=1$. At the main algorithm, we get the values $x=1$ and $y=-1$. Since $s_x<s_y$, the final result is corrected to $(x,y)=(y,x)+S^\perp=(1,1)+(3,-2)=(2,-1)$. Adding this value to the initial iteration (0,2) of hyperplane 6, we get (0,2)+(2,-1)=(2,1) as expected.

![Figure 2. Iteration space and execution sequence (2,3)](image)

**The address generator**

From property 4 and the algorithm LIRA, we know that there will be a repetitive pattern among the $Li$s at every $s_y$ hyperplanes. Therefore, we can design an address generator to control the advancement of the execution along different hyperplanes. In order to understand the function of such address generator, let us assume the execution begins at iteration (0,0). After computing the displacement vector with the algorithm LIRA, we can determine $L_{h=1}$. Without loss of generality, let us consider $s_y$ very large. The lemma below shows how to obtain $L_{h=ap}$ for $2s_y \leq s_y$, through a recursion function.

**Lemma 1:** Given $L_{h=1}=(i,j)$, $L_{h=ap}=(i',j')$ and the schedule vector $s$, then $L_{h=ap}$ is recursively computed by:

1. $L_{h=ap} = L_{h=ap-1} + L_{h=1}$, when $i+i' < s_x$, or
2. $L_{h=ap} = L_{h=ap-1} + L_{h=1}-s^\perp$, when $i+i' \geq s_y$.
Therefore, a new displacement vector can be computed for each hyperplane in the interval \( [0, s] \). The set of vectors can be made equivalent to the output of a look-up table, in order to produce a fast correct sequence of execution. The complexity of the address generator design can be reduced by using an if statement to control the advancement along the hyperplanes, generating a displacement vector between hyperplanes instead of the final iteration indices.

The constrained iteration space

The final computation of the indices requires the use of the LI{s} and a perfect control over the boundaries of the iteration space. Let us assume, from now on, that there are \( M \) iterations in the x-direction and \( N \) iterations in the y-direction. We already know, that along any hyperplane \( p \), the sequence of execution is controlled by the addition of \( s^\perp \) to the current iteration, beginning with \( L_{h=0} \). Across the hyperplanes, the address generator will provide the way to compute the \( L_{h=1} \) from \( L_{h=0} \). The problem becomes to keep the execution inside the iteration space bounds, what is solved by the properties expressed in the theorem below:

**Theorem 1:** Given an iteration space, size \( M \times N \), being executed according to a schedule vector \( s = (s_x, s_y) \), and a scheduling hyperplane \( p \), then:

1. If \( \implies (i, j) \in \mathbb{E} \) and \( j \cdot s_x < 0 \) or \( i + s_x \geq M \), then the next iteration after \( i \) is \( L_{h=1} \)
2. If \( L_{h=0} = (i, j) \) and \( j \geq N \), then \( L_{h=1} = (i, j) + s^\perp \).

Such theorem can be proven by geometric constructions. However, when considering the boundaries of the iteration space, such properties will not be sufficient to allow the correct computation of the entire iteration space, since we will notice that some hyperplanes, at the lower left and upper right corners, may not contain any iteration point. This can be seen in the example of figure 2, where the second hyperplane fits in this condition. Such discontinuities are restricted to an initial and final set of hyperplanes that occur in specific intervals. The algorithm below shows the mathematical formulation of the index generator, applicable to the region of the iteration space where there are no discontinuities in the sequence of hyperplanes, that will be translated in the main section of the VHDL component. Note that \( M, N, \) and \( s \) are known at compile time and are substituted by their exact value, reducing the code complexity.

**Algorithm Address_Generation**

\[
\begin{align*}
(x,y) & \leftarrow (x,y) + s^\perp & \text{/* computes the next iteration */} \\
\text{if (reset)} & \text{ /* resets the system */} \\
& \text{w} \leftarrow \min(s_x, s_y) \\
& (x,y) \leftarrow L_{h=0} + (w-1) \\
& \text{w} \leftarrow (M-1) \cdot s_x + (N-1) \cdot s_y - \text{w} \cdot (w-1) \\
\text{endif} \\
\text{if (y < 0 or x > M)} & \text{ /* if out of range move to next hyperplane */} \\
& (x,y) \leftarrow (x_{old}, y_{old}) + L_{h=1} \\
\text{if (x \geq s_x and y \leq N - s_y)} & \text{ /* special case, does not satisfy properties */} \\
& (x,y) \leftarrow (x_{old}, y_{old}) + L_{h=0} + s^\perp \\
\text{endif} \\
& (x_{old}, y_{old}) \leftarrow (x,y) \\
\text{endif} \\
\text{if } (x,y) = L_{h=0} \text{ then stop} \\
\text{else outputs } (x,y) \\
\text{end Address_Generation}
\end{align*}
\]

**Implementation**

As seen in the previous sections, the address generator becomes the most important component of the final design. The amount of combinatorial circuit associated with the address generator requires a careful design. However, the address generation algorithm provides the complete basis for a large portion of the circuit implementation. Therefore, we choose to use two processes: the first responsible for updating the indices, and the second for storing the results between two clock cycles. A VHDL behavioral description of the storage (feedback) process can be seen in [19]. In order to handle the
discontinuities introduced by the boundaries of the iteration space, two possible design alternatives exist. The implementation option adopted on our experiments consists of including in the address generator process an initial and final sequence of pre-compiled indices in such a way that the empty hyperplanes will be skipped. A loop modified to accommodate the information about the schedule vector would look like the example below:

```c
for (i,j) = (1,1) to (8,8) sch (2,3)
    /* LOOP BODY */
    y(i,j) = c1*y(i-1,j) + c2*y(i,j-1) + x(i,j)
```

The equivalent VHDL behavioral description code for the process involved in the design of the address generator A is presented in [19]. A second solution for the design of the address generator assumes that there are no such discontinuities and every time an out-of-range iteration is computed, it will send a disable signal to the loop body process, preventing it of producing any new result on that cycle. Such a solution is also described in [19].

4. THE OPTIMIZED FILTER FUNCTION

This section describes how to combine the chained multi-dimensional retiming and the concept of predicated execution in order to allow full fine-grain parallelism of the instructions comprising the body of a nested loop. An example of a two-dimensional problem consists of an edge detection algorithm based in the computation of the Laplace edge enhancement [12]. The nested loop representing this problem could be coded as:

```c
for (i,j) = (1,1) to (64,64) sch (0,1)
    /* LOOP BODY */
    Ly(i,j) = y(i+1,j) + y(i-1,j) + y(i,j+1) + y(i,j-1) - 4*y(i,j)
    If Ly(i,j) < threshold
        Image = dark
    Else
        Image = edge
endif
```

An example of an image treated with the Laplacian edge detector is presented in figure 3.

![Figure 3. Example of an image submitted to the Laplacian edge detection filter](image)

An acyclic graph equivalent to the code above is shown in figure 4, where the node F is equivalent to an if statement. In a processor with one arithmetic logic unit, one can easily conclude that the minimum schedule length, required to execute the edge detection process, would be equivalent to 7 control steps. After applying the if-conversion technique to the loop, if we assume that an infinite number of computational units, able to execute one operation in one control step, can be added to the VHDL design then the schedule length can be reduced to 5 control steps by the application of a traditional list scheduling method [4], as shown in figure 5(a). Notice that the execution of instructions G and H are now dependent on the predicate register f. Figure 5(b) shows a more compact schedule in which node D has been rescheduled to the first control step. This new schedule can be associated with the retimed graph presented in figure 6(a), where the two-dimensional delay (0,1) was pushed through nodes A and B of the original MDFG (Note that (0,1) is arbitrarily chosen here. The vector (1,0) is also possible, as are many others). Intuitively, the retimed node no longer precedes the operation D within the same iteration. Also, in this particular example, there is no need for a change on the schedule vector.
The overlap of different iterations of the loop requires the reutilization of the predicate registers. The *chained multi-dimensional retiming algorithm* modifies the loop in such a way that all instructions can be run in parallel. The use of overlapped iterations is the basis for this parallelism. In order to avoid the reutilization of the predicate registers, these registers are then implemented as shift registers and dimensioned in such a way to store their Boolean values until they are required.

The MDRPRED algorithm, introduced in [21] and revisited below, allows the designer to obtain the dimensions of the predicate registers in such a way that overlapped iterations do not interfere with each other.

**Algorithm MDRPRED** (\( G = (V, E, d, t) \))

Choose \( s = (s_1, s_2, \ldots, s_n) \) such that \( s \bullet d(e) > 0 \) for any \( e \in E \)

Choose \( r \) such that \( r \perp s \)

\( \forall u \in V, LV(u) \leftarrow 0 \)

\( LV_{\text{max}} \leftarrow 0; \ \text{QueueV} \leftarrow \emptyset \)

/* remove original edges with non-zero delays */

\( E \leftarrow E - \{e_i, \text{ s.t. } d(e_i) \neq (0, \ldots, 0)\} \)

/* queue schedulable nodes */

\( \text{QueueV} \leftarrow \text{QueueV} \cup \{u \in V, \text{ s.t. indegree } (u) = 0\} \)

while \( \text{QueueV} \neq \emptyset \)

get(u, QueueV); schedule(u)

/* propagate the level values to successor nodes of u */

\( \forall v \text{ such that } u \leftarrow v \)

\( \text{indegree} (v) \leftarrow \text{indegree}(v) - 1 \)

\( LV(v) \leftarrow \max\{LV(v), LV(u)+1\} \)

\( LV_{\text{max}} \leftarrow \max\{LV(v), LV_{\text{max}}\} \)

/* check for new schedulable nodes */

if \( \text{indegree}(v) = 0 \)

Figure 4. MDFG representing an edge detection filter

Figure 5. (a) Initial schedule (b) schedule after retiming A and B (c) final schedule
QueueV ← QueueV ∪ \{v\}
end if
end while
/* compute the multi-dimensional retiming */
\forall u \in V, r(u) = (LV_{max} - LV(u))
/* compute the size of the predicate shift registers required between decision \(u_p\) and nodes \(v_i\) */
\forall p \in P, s(p) = \max(LV(v_i|p) - LV(u_p)) + 1
End MDRPRED

Let us examine how the algorithm works. In the first step, the nested loop is translated in a MDFG, which is the input to the algorithm. Since there is no restrictions on the schedule vector \(s\), it can be easily selected under the condition that its product by any graph dependency be positive. After choosing \(s\), a basic retiming vector perpendicular to \(s\) is computed. Non-zero delay edges are removed making the graph an acyclic one. A simple graph traversal procedure is used to do a topological sort of the graph. The level of each node, computed by the topological sort is used to evaluate the required retiming. Those nodes initially labeled at level 1 can be concurrently executed, while those in upper levels are clearly data dependent on their predecessors. In order to have these nodes in parallel with those at level 1, the dependencies found within the iteration need to be transformed in inter-iteration dependencies.

Figure 6. Graph after retiming

The retiming process will indicate these overlapping requirements by looking up to the levels of each node and the maximum level in the graph. This same information becomes the source for the evaluation of the life span of the predicate registers. Therefore, in the last lines of the algorithm the dimension of the shift registers is determined. Considering again the example, in figure 4(c), the remaining nodes were also rescheduled to new positions, resulting in an optimal schedule with length 1. Figure 5 shows the final graph, where the two-dimensional delay \((0,1)\) was repetitively applied to the nodes of the graph. The VHDL implementation is derived directly from the resulting schedule and address generation process. The MDFG delays are translated in memory queues and provide the required data dependency between operations. Figure 7 shows a block diagram of this circuit while a sketch of the VHDL program is shown in figure 8. Notice that this is not the entire program and several details of the implementation have been omitted.

Figure 7. Diagram of the filter implementation
The code was implemented using Altera VHDL software and Altera library components and is being tested in a FPGA prototype board. As it can be seen, registers were placed in between the functional units to reduce the cycle time of the circuit and allow the parallel execution of the inner loop operations.

5. SUMMARY

The design of ASICs can provide the necessary performance to critical nested loops of multi-dimensional applications including two-dimensional filters. The performance degradation experienced through the execution of conditional branches can be eliminated by using branch predication techniques. Hardware description languages are required to optimize the design cycle time and documentation of a system. VHDL being one of the most important of those languages is supposed to be used for any application. However, existing synthesis tools present a set of constraints that force the designer to write VHDL code with the final hardware solution in mind. This limitation prevents the designer of using automatic optimization tools. This paper showed the implementation of a VHDL description of an edge detector filter that efficiently combines multidimensional retiming and branch predication in the scheduling of the optimized loops and in the design of the predicate registers required by the branch predication techniques.

```vhdl
entity filter is port ( x, y, threshold: in std_logic_vector (8 downto 1);
                          load, shiftin, clk, enable, clk1, ena, reset, write: in std_logic;
                          result: out std_logic);
end filter;
architecture arch1 of filter is
begin
    xp:=x + "00000001";
    xm:=x - "00000001";
    yp:=y + "00000001";
    ym:=y - "00000001";
    ADDRGEN1: ADDRGEN PORT MAP (x=x,y=yp.reset=>reset,clock=>clk,address=>ad1);
    ADDRGEN2: ADDRGEN PORT MAP (x=xm,y=ym.reset=>reset,clock=>clk,address=>ad2);
    ADDRGEN3: ADDRGEN PORT MAP (x=xm,y=yp.reset=>reset,clock=>clk,address=>ad3);
    ADDRGEN4: ADDRGEN PORT MAP (x=x,y=ym.reset=>reset,clock=>clk,address=>ad4);
    ADDRGEN5: ADDRGEN PORT MAP (x=x,y=yp.reset=>reset,clock=>clk,address=>ad5);
    add1: lpm_add_sub PORT MAP (dataa=>fp, datab=>fm, result=>R1in, clock=>clk);
    reg1: lpm_ff PORT MAP (data => R1in, clock => clk, q => R1out);
    reg2: lpm_ff PORT MAP (data => R2in, clock => clk, q => R2out);
    reg3: lpm_ff PORT MAP (data => R3in, clock => clk, q => R3out);
    reg4a: lpm_ff PORT MAP (data => R4ain, clock => clk, q => R4in);
    reg4: lpm_ff PORT MAP (data => R4in, clock => clk, q => R5in);
    reg5: lpm_ff PORT MAP (data => R5in, clock => clk, q => R6in);
    reg6: lpm_ff PORT MAP (data => R6in, clock => clk, q => R6out);
```
reg7: lpm_ff     PORT MAP (data => sum, clock => clk, q => R7out);
    a_s<=0;
--this component subtracts the result of "shifta" from that of "add3"
sub1: lpm_add_sub     PORT MAP (dataa=>R3out, datab=>R6out, result=>sum,
add_sub>a_s,clock=>clk);
--this component compares the result from "sub1" with an input threshold
    comp: lpm_compare     PORT MAP (data=>R7out, datab=>threshold, ageb=>latchin,clock=>clk);
--store the final result
    latch1: latch PORT MAP (d=>latchin,ENA=>ena, q=>final);
end arch1;

Figure 8. VHDL code

ACKNOWLEDGEMENTS

This work was partially supported by the National Science Foundation under Grant No. MIP 97-04276.

REFERENCES


