A Parameterized Index-Generator for the Multi-Dimensional Interleaving Optimization*

Nelson Luiz Passos Edwin Hsing-Mean Sha
Dept. of Computer Science & Engineering
University of Notre Dame
Notre Dame, IN 46556

Abstract

The novel optimization technique for the design of application specific integrated circuits of multi-dimensional problems, called multi-dimensional interleaving consists of an expansion and compression of the iteration space. It guarantees that all functional elements of a circuitry can be executed simultaneously, and no additional memory queues proportional to the problem size are required. Such technique, that considers the parallelism inherent to multi-dimensional problems, depend on loop transformations that require a new execution sequence of the loop. This study presents a new approach on synthesizing multi-dimensional (nested) loops, where pre-processor tools can rewrite the instructions in such a way to accomodate the required changes in the optimized design. This new approach is expected to improve the design cycle by including multi-dimensional signal processing and other common applications in the scope of the synthesis tools.

1. Introduction

Multi-dimensional systems, including image processing, geophysical signal processing, and fluid dynamics, are becoming one of the most important targets of computational improvement studies. Most of the optimized solutions to those problems point to the use of Application Specific Integrated Circuits (ASICs). It is well known that a parallel implementation of such operations would improve the performance of the ASIC design. A linear time algorithm, applicable to multi-dimensional (MD) problems, able to achieve a fully parallel system design, i.e., simultaneous execution of all the operations in the loop body, while maintaining the original number of required memory queues has been developed [11]. This new mechanism, based on fine-grain parallelism is called *multi-dimensional interleaving*. It requires a new execution order of the loop, which directly affects the loop bounds and indices. In this study we present a technique on how to implement the correct indices generation for that method.

Most of the previous research on loop parallelization has focused on one-dimensional problems. Recent studies have considered the optimization of nested loops, a software point of view of the MD systems [1, 7, 12]. In the area of high-level synthesis, researchers also have focused on the optimization of MD problems [2, 6]. In a previous study, it has been shown that full parallelism can be obtained by the application of MD retiming techniques [10]. In general, these methods transform the loops in such a way to obtain a new sequence of execution characterized by a higher parallelism. This sequence of execution is commonly associated with a schedule vector. The new schedule vector usually differs from the one used in the original design, introducing new memory requirements that may end up in complex storage control and substantial increment on the memory size. In [9] a fully parallel solution is achieved by applying a new execution order to the iterations in a given block size. In the *multi-dimensional interleaving* method, the loop body, also known as one iteration, is modeled as a multi-dimensional data flow graph (MDFG) [11]. The method improves the parallelism by restructuring the iteration space and the loop body without changing the original schedule vector, and consequently, not requiring additional queues. An expansion of the iteration space is applied to improve the potential of parallelism, while a compression avoids any loss of performance. A single processor architecture is the target system, and a maximum throughput of one result per cycle time[1] is achievable through the additional application of a *multi-dimensional retiming*.

In this paper, we present a new coding approach that allows the designer to specify the expansion and migration vectors in such a way to obtain the exact ordering of the indices, according to the multi-dimensional interleaving algorithm. A simple example of the application of our method consists of a filter, represented by the transfer function:

$$H(z_1, z_2) = \frac{1}{\left(1 - c_1 * z_1^{-1} - c_2 * z_2^{-1}\right)}$$

An MDFG representing this problem is shown in figure 1. Nodes represent operations and edges represent data dependencies. The labels on the edges indicate the MD distance between iterations. A designer could represent the simulation of such a problem in a computer programming language such as C or Fortran, by:

```
for i = 0 to 1000
  for j = 0 to 1000
    /* LOOP BODY */
    y(i,j) = c1 * y(i-1,j) + c2 * y(i,j-1) + x(i,j)
```

*This work was supported in part by the NSF CAREER grant MIP 95-01006, and by the William D. Mensch, Jr. Fellowship.

[1] The cycle time is assumed to be the longest execution time among all operations in the loop body

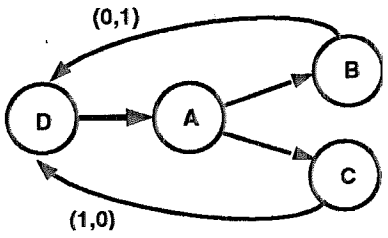

Figure 1. MDFG representing a simple filter

In our approach, the same loop would be coded in such a way to describe the execution sequence through two specific parameters, f and c for the expansion and compression vectors, computed by the multi-dimensional interleaving algorithm. For example, the previous loop, without any optimization, would be coded as:

$$for\ (i,j) = (0,0)\ to\ (1000,1000)\ f\ = (1,1)\ c\ = (0,0)$$
/* LOOP BODY */
$$y(i,j) = c1 * y(i-1,j) + c2 * y(i,j-1) + x(i,j)$$

In this new construct, the loop indices are computed from the initial index to the last possible one, according to those parameters that define the expansion and compression operations in the multi-dimensional interleaving. The final synthesis consists of two components: one representing the loop body and the second being an address generator imposing the order of execution. Re-examining the previous example, let us assume that the new sequence of execution, after applying the multi-dimensional interleaving would be $(0,0),(0,1),(1,0),(0,2),(1,1),(2,0),(0,3),\ldots$ for $f = (1,3)$ and $c = (-1,4)$, instead of the regular row-wise sequence.

As we can see there is a high complexity in the management of the execution sequence. In our new construct, a clear, neat, and efficient format is used to reflect the change. Only the parameters f and c in the code are affected, as shown below:

$$for\ (i,j) = (0,0)\ to\ (1000,1000)\ \text{f= (1,3) c= (-1,4)}$$
/* LOOP BODY */
$$y(i,j) = c1 * y(i-1,j) + c2 * y(i,j-1) + x(i,j)$$

This new execution sequence is then implemented on the address generator mechanism, which emulates a modified finite state machine, computed in pre-compile time, in order to provide the correct index values.

This paper presents the new construct, how the decomposition is done, and the theoretical aspects of the address generator. It begins by showing some basic concepts in Section 2. In Section 3, we discuss the theoretical aspects on how to compute the correct sequence of execution. Section 4 shows the implementation of our technique to practical situations, and some observations on its utilization.

2. Basic concepts

In this section, we briefly review the multi-dimensional interleaving technique that could be applied to produce a parallel design solution. We begin by focusing our attention in the execution sequence of the problem.

2.1. The execution sequence

The execution of each operation in the loop body, exactly once, represents an *iteration*. Iterations are identified by a vector i, equivalent to a multi-dimensional index, starting from $(0,0,\ldots,0)$. 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 which 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.

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. Vectors are ordered in lexicographic order. A *schedule vector* s is the normal vector for a set of parallel hyperplanes that define a sequence of execution of the iteration space.

Most of the applications requiring an ASIC design have a time constraint that can not be achieved by the straightforward implementation of the loop. In this case, optimization techniques are used to improve the parallelism among the operations in order to satisfy the time constraint.

Some of the existing optimization methods point directly to MD problems [2, 6, 10, 12]. Most of these methods derive from the work by Lamport [5], and require a new scheduling direction when executing the loop iterations. Such a change implies in complex formulations of loop bounds and indices when writing the transformed, optimized, code. A possible method of obtaining fully-parallel solution to our problem is to use *multi-dimensional retiming* techniques. A *multi-dimensional retiming* $r(u)$, of a node u in an MDFG G, represents delay components pushed into the edges $u \rightarrow v$, and subtracted from the edges $w \rightarrow u$, where $u, v, w \in G$. Therefore, we have $d_r(e) = d(e) + r(u) - r(v)$ for every edge $e = u \rightarrow v$ and $d_r(l) = d(l)$ for every cycle $l \in G$.

2.2. The multi-dimensional interleaving technique

Let us examine again the example in figure 1. We see that the data produced by node D is immediately consumed by node A. The simultaneous execution of those two functions requires a register or latch device between them. In order to solve this problem, we begin by applying an *expander function* that will provide the potential for the desired retiming.

The *expander function* transforms an MDFG $G = (V, E, d, t)$ into a new MDFG $G_e = (V, E, d_e, t)$ such that $d_e = f_e \times d$, where the vector f_e has the form $f_e = (1,\ldots,1,f)$, and is called the *expander coefficient*. The expansion of the iteration space results in a new set of iterations larger than the existing number of points in the original problem. Therefore, some of the new iterations will compute non-valid outputs. These new iterations are said to be *empty* or *inactive*. An *expanded row cycle* (ERC) is the set of f consecutive iterations in the row direction, such that the first iteration of the set is equivalent to one of the original iterations in G and the next $f-1$ are empty iterations. This first iteration is called *head* of the expanded row cycle.

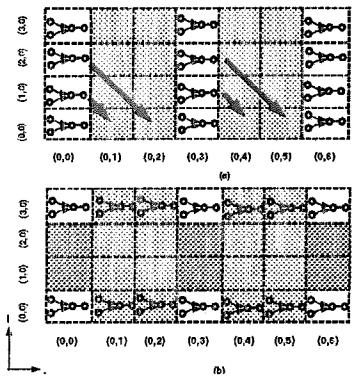

Figure 2. (a) expanded iteration space and iterations being moved (b) iteration space after compression

After the expansion of the iteration space, the memory elements have an increased size. An immediate consequence of the increased size of the memory elements is that the required retiming is no more constrained by the number of delays in a cycle since such number can be increased as much as necessary to obtain a legal retiming.

The expanded iteration space presents a loss of performance due to the inactive iterations. In order to recover such a loss, several rows are compressed in one, called the *target row*. A *compression operation* on row h, moves valid iterations from rows $h+1$ to $h+f-1$ to the inactive iterations found in row h. After the compression, the $f-1$ rows moved down are deleted from the iteration space. Figure 2 presents a compression operation applied to the expanded iteration space representing the example on figure 1. However, the result is not useful for our further optimization because some of the original delay vectors $(1,0)$ became $(0,1)$ in the transformed space and will restrict the application of the MD retiming.

To avoid the problem of introducing retiming restrictions, the compression mechanism puts together iterations that can run in parallel. For example, examining again figure 2(a), we will notice that iteration $(1,0)$ is the appropriate candidate to be moved to the position after iteration $(0,3)$, i.e, moved to the position $(0,4)$. In order to identify such ideal candidates, a *compression vector* c indicates the direction in which valid iterations should be moved to. In our example, since we want to bring down iteration $(1,0)$ to position $(0,4)$, the compression vector is $c=(-1,4)$. Combining the expansion and compression, this mapping can be generalized as: iteration $I=(i,j)$ is moved to position $I*f+mod(i,f)*c$.

A new set of dependencies in the row-direction is created by the transformation of vertical dependencies, and they may affect the final MD retiming by introducing new restrictions on the number of delays in a cycle. Therefore, for a given MDFG $G=(V,E,d,t)$, submitted to a compression vector c and a expander function f_e, transforming G in $G_m=(V,E,d_m,t)$, it has been proven that there exists a multi-dimensional retiming function $r=(0,\ldots,0,r_n)$, $r_n>0$, such that $d_m(e)\neq(0,0,\ldots,0), \forall e\in E$, if for any cycle $l=v_1\rightarrow v_2\rightarrow\ldots\rightarrow v_k\rightarrow v_1$, $v_i\in V, 1\leq i\leq k$, $d_m(l)\geq(0,0,\ldots,0,k)$.

The correct choice of f_e and c is the key for obtaining the MD retiming in the row direction such that all operations can be executed in parallel. The problem of finding the expander coefficient and the retiming function is solved by the algorithm called *MuDIn* for *Multi-Dimensional Interleaving*, described in [11] and reproduced below:

MuDIn(G)
remove non-zero delay vectors of G
perform a topological sort of the graph, assigning decreasing negative levels to each node
compute the multi-dimensional interleaving parameters
$\forall u \xrightarrow{e} v \in E \; s.t. \; d(e)=(0,0,\ldots,0,d_1)\neq(0,0,\ldots,0)$
$f \leftarrow max\left\{\left\lceil\frac{LEVEL(v)-LEVEL(u)+1}{d_1}\right\rceil\right\}$
$\forall d(e)=(0,0,\ldots,0,(d_2,d_1)),\ 0<d_2<f,\ e\in E$
$p \leftarrow min\left\{0,\left\lfloor\frac{d_1}{d_2}\right\rfloor\right\}$
$\forall u \xrightarrow{e} v \in E \; s.t. \; d(e)=(0,0,\ldots,0,(d_2,d_1)),\ 0<d_2<f$
$\gamma \leftarrow max\left\{0,\left\lceil\frac{LEVEL(v)-LEVEL(u)+1-f*d_1-d_2+d_2*f*p}{d_2*f}\right\rceil\right\}$
$g \leftarrow \gamma*f-f*p+1$
compute G_m
compute the retiming function for each node
$\forall v\in V$, compute $r(v)=LEVEL(v)\times(1,0,\ldots,0)$

---

3. Controlling the execution sequence

In order to satisfy the designer need of a function able to compute the correct sequence of indices, this study implements the behavioral description of an MD application by a new loop command that scans the iteration space according to the compression and expansion vectors computed by the multi-dimensional interleaving.

3.1. An infinite iteration space

As we have seen before, the execution of the iteration space, according to a row-wise sequence seems to be of trivial implementation. However, since this study contemplates a more general solution, the execution order must satisfy the constraints imposed by the multi-dimensional interleaving technique. Let us assume, for now, an iteration space that has no bounds in the x- or y-direction. We begin by examining some properties of the new execution sequence for a given two-dimensional iteration space and the expansion and compression vectors. The theorem below shows how to compute the correct execution sequence.

Theorem 3.1 *Given an expander coefficient $f_e=(1,f)$ and a compression vector $c=(-1,g)$, after an iteration $t=(i,j)$ has been executed, the next iteration must be either $\frac{t*f_e-c+(0,1)}{f_e}$, or $\frac{t*f_e+(f-1)*c+(0,1)}{f_e}$.*

Let us examine the example consisting of the iteration space shown in figure 3. After executing $(0,1)$, the next iteration will be $\frac{(0,1)(1,3)-(-1,4)+(0,1)}{1,3}=\frac{(1,0)}{1,3}=(1,0)$ as stated in theorem 3.1. From $(0,2)$ the next iteration is $\frac{(0,2)(1,3)-(-1,4)+(0,1)}{1,3}=\frac{(1,3)}{1,3}=(1,1)$. In order to develop

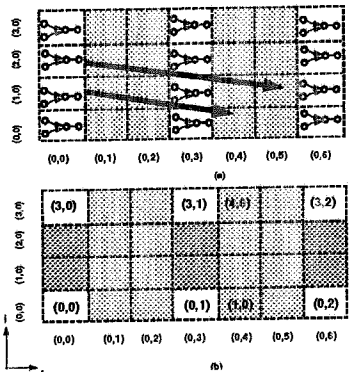

Figure 3. (a) expanded iteration space and iterations being moved (b) iteration space after compression

a method on how to compute the sequence of iterations, we focus now on properties of those iterations:

Property 3.1 *If (i, j) was mapped to the head of an ERC, obtained through the application of an expander coefficient $f_e = (1, f)$, then $(i, j+1)$ will be executed f iterations after (i, j).*

Property 3.2 *If (i, j) was mapped to the head of an ERC, obtained through the application of an expander coefficient $f_e = (1, f)$, and (i', j') is the next executable iteration then $(i', j') = (i, j) + \frac{(0,1)-c}{f_e}$.*

By using properties 3.1 and 3.2, we compute the correct sequence of execution, without the usage of any *mod* or *divide* operations, since f_e and c are known at compile time. We call the vector $\delta = \frac{(0,1)-c}{f_e}$ the *displacement vector*.

3.2. The finite state machine

From theorem 3.1 and the MD interleaving algorithm, we know that there will be a repetitive pattern at every f iterations. Therefore, we can design a finite state machine (FSM) with f states to control the advancement of the execution. Furthermore, this machine can be re-designed and reduced to two states. In order to understand the function of such FSM, let us assume the execution begins at iteration $(0, 0)$. After computing the displacement vector, we can determine the execution sequence. Without loss of generality, let us consider row zero as the initial target row. Next, we summarize how to obtain the execution sequence through a recursion function, for a given $t_k = (i, j)$, $0 \leq i \leq f-1$, and a displacement vector δ computed with a compression vector c and an expander coefficient $f_e = (1, f)$. t_{k+1} is recursively computed by:

1. $t_{k+1} = t_k + \delta$, when $i < f-1$, or
2. $t_{k+1} = t_k - (f-1) * \delta$, when $i = f-1$.

current state	next	output
0	1	$t_0 + (0, 1)$
1	2	$t_0 + \delta$
2	3	$t_0 + 2 * \delta$
...	...	...
$f-1$	0	$t_0 + (f-1) * \delta$

Table 1. Example of the FSM output values

Therefore, a new displacement vector with respect to the target row, can be computed for each iteration in the rows in the interval $[0, f[$. The set of vectors could be made equivalent to the output of each state in the FSM in order to produce the correct sequence of execution. Table 1 shows a possible example of the parameters used to control the FSM.

The FSM can be reduced to a two state hybrid machine, consisting of combinatorial and sequential logic, by using an *if statement* to control the state switch, and generating the displacement vector between rows instead of the final iteration indices. This reduced FSM is then used to implement the address generator described next.

3.3. The constrained iteration space

The final computation of the indices requires 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. The problem becomes to keep the execution inside the iteration space bounds, what is solved by the properties expressed in the lemma below:

Lemma 3.2 *Given an iteration space, size $M \times N$, being executed according to an expander coefficient $f_e = (1, f)$ and a compression vector c that produced a displacement vector $\delta = (\delta_x, \delta_y)$, then:*

1. *if $t = (i, j)$ has been executed in a sequence where t_0 is a head of an ERC, and $j + \delta_y < 0$ or $i + \delta_x \geq M$ or $i - i_0 + \delta_x \geq f$, then the next iteration after t is $t_0 + (0, 1)$.*
2. *if $j_0 + 1 \geq N$, then the next iteration after t is $t_0 + (0, 1) + (j_0 + 1 - N) * \delta$ or $(i + 1, 0)$.*

Such theorem can be proven by geometric constructions. The algorithm below shows the mathematical formulation of the index generator, that will be translated in a function or subroutine in a traditional programming language, or in the main section of a VHDL component description. Note that M, N, f_e and c are known at compile time and are substituted by their exact value, reducing the code complexity.

Algorithm Address_Generation
/* computes the next iteration */
$(x, y) \leftarrow (x, y) + \delta$
/* resets the system */
if (*reset*)

$x_{ref} \leftarrow f - 1$
$(x, y) \leftarrow (0, 0)$
$(x_{old}, y_{old}) \leftarrow (x, y)$
endif
/* if out of range move to ERC */
if $(y < 0$ *or* $x > M - 1$ *or* $x > x_{ref})$
$(x, y) \leftarrow (x_{old}, y_{old}) + (0, 1)$
$\{(x_{old}, y_{old}) \leftarrow (x, y)$
/* does not satisfy right boundaries */
endif
if $y > N - 1$
/* special case, time to move up */
if $(x = x_{ref}$ *and* $y_{old} = N - 1)$
$(x, y) \leftarrow (x_{ref} + f, 0)$
$x_{ref} \leftarrow x + f - 1$
else
$\{(x, y) \leftarrow (x_{old}, y_{old}) + delta$
$\{(x_{old}, y_{old}) \leftarrow (x, y)$
endif
endif
outputs (x, y)
end Address_Generation

4. 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 two-states FSM requires a careful design. However, the address generation algorithm provides the complete basis for a large portion of the circuit implementation. Therefore, in a VHDL implementation, we choose to emulate the FSM through the use of two processes: the first responsible for updating the indices, and the second for storing the results between two clock cycles. Below, we show the behavioral description of the storage (feedback) process.

```
--  Latch the data between clock cycles
save: PROCESS (clk)
BEGIN
IF clk = '0' THEN
   IF ch = '1' THEN
      x_0s <= x_s1;
      y_0s <= y_s1;
   END IF ;
   st_s <= st;
   x_s <= x_s1;
   y_s <= y_s1;
END IF ;
END PROCESS ;
```

Figure 4 shows the representation of such a solution. Examining again the problem of figure 1, let us assume that it could be described by the MD loop below:

for $(i, j) = (0, 0)$ *to* $(7, 7)$ $f = (1, 3)$ $c = (-1, 4)$
/* 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 could be:

```
-- Address generation process
gaddr: PROCESS(x_0s,y_0s,x_s,y_s,x_r_s,rst,clk)
   VARIABLE x_0,y_0: qsim_state_VECTOR (0 TO 7);
```

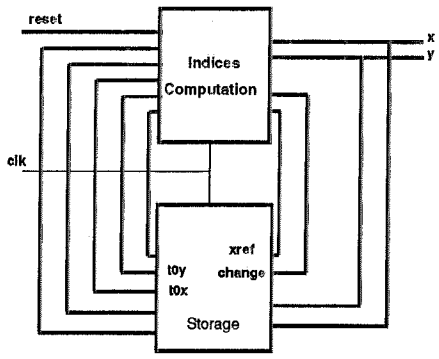

Figure 4. Address generator implementation

```
   VARIABLE x_1,y_1: qsim_state_VECTOR (0 TO 7);
   VARIABLE x_v,y_v: qsim_state_VECTOR (0 TO 7);
   VARIABLE x_ref  : qsim_state_VECTOR (0 TO 7);
   VARIABLE change : qsim_state;
BEGIN
IF clk= '1'  THEN
    x_ref := x_r_s;
    x_0 := x_0s;
    y_0 := y_0s;
    x_v := x_s;
    y_v := y_s;
    x_1 := x_v + "00000001";
    y_1 := y_v - "00000001";
    change := '0';
--  Re-initialize (asynchronous signal)
    IF rst = '1' THEN
         change :='1';
         x_1   := "00000000";
         y_1   := "00000000";
         x_ref := "00000010";
    END IF;
--  If next is out-of-range,
--          compute next head of ERC
    IF y_1(0) = '1' OR x_1 > "00000111"
       OR x_1 > x_ref THEN
         change :='1';
         x_1 := x_0;
         y_1 := y_0 + "00000001";
    END IF;
--  Out-of-bound in the y-direction
    IF (y_1 > "00000111") THEN
         IF (x_1 = x_ref
            AND y_0 = "00000111") THEN
--  Move to the next target row
            x_1   := x_ref + "00000001";
            x_ref := x_1 + "00000010";
            y_1   := "00000000";
         ELSE
--  Move to the next non-executed point
            change :='1';
            x_1 := x_1 + "00000001";
            y_1 := y_1 - "00000001";
         END IF ;
    END IF ;
x_ref1 <= x_ref;
ch     <= change;
x      <= x_1;
```

```
y       <= y_1;
x_s1    <= x_1;
y_s1    <= y_1;
END IF ;
END PROCESS ;
```

The process *gaddr* performs all necessary computations in order to issue the indices in the right order, while the process *save* stores the signals required for the next cycle. Due to some synthesis constraints we were forced to use bit vectors for integer representation.

The loop body is easily implemented, as also the memory elements, using available library components and the required combinatorial logic. The address generator is compiled and synthesized as an independent component, becoming available to the pre-compiler as replacement commands for the MD loop. For the example problem, we assumed a three-read port memory in order to keep the loop body synchronized with the address generator, using only one global clock signal. A possible high-level VHDL structural description code for this implementation would be the one presented below:

```
PROCESS (rst,clk,ydata)
BEGIN
 GADDR(x0,y0,xs,ys,xs1,ys1,x,y,st,st_s,rst,clk);
 SAVE(x0,y0,xs,ys,xs1,ys1,ch,st,st_s,clk)
 LOOPBODY(x,y,ydata,clk,output);
END IF;
END PROCESS ;
END MDloop;
```

In this example, we know that we can get a fully parallel design of the loop body by using the MD retiming optimization technique. In our example, when using available CMOS component libraries, the multipliers used in the loop body have a execution time equivalent to 40 ns and the adders need 20 ns. This implies that the original loop body would require a clock cycle of 80 ns due to the sequential execution of two additions followed by two parallel multiplications. After applying the optimization techniques, the execution time would be reduced to approximately 40 ns (a 50 % improvement), justifying the use of the new address generator component. Combining this addressing mechanism with the MD retiming optimization results in a efficient and automatic technique of improving the execution time of MD loops. Four examples of MD problems are presented in table 2 to show the achievable performance gain by applying the multi-dimensional interleaving technique as done in our new command. The first one, *2D IIR*, consists of the code required to emulate a two-dimensional Infinite Response Filter [10]. The second case is a loop used to simulate a differential pulse-code modulation (*DPCM*), commonly used in image data compression [4]. The third, *HEAT*, is the optimization of a heat transfer problem [8], and finally, the fourth is a wave digital filter, *WDF*, used to simulate a transmission line problem [3]. These examples demonstrate the importance of having automatic optimization tools in the design process, justifying the adaptation of languages such as VHDL to allow the application of such techniques.

Problem	execution time		% improvement
	initial	optimized	
2D IIR	120 ns	40 ns	67
DPCM	140 ns	40 ns	71
HEAT	80 ns	40 ns	50
WDF	160 ns	40 ns	75

Table 2. Performance improvement in different problems, using the multi-dimensional interleaving/retiming technique

REFERENCES

[1] L.-F. Chao and E. H.-M. Sha, " Static Scheduling of Uniform Nested Loops," *Proc. of 7th International Parallel Processing Symposium* , 1993, pp. 1421-1424.

[2] A. Darte and Y. Robert, " Constructive Methods for Scheduling Uniform Loop Nests," *IEEE Trans. on Parallel and Distributed Systems*, 1994, Vol. 5, no. 8, pp. 814-822.

[3] A. Fettweis and G. Nitsche, " Numerical Integration of Partial Differential Equations Using Principles of Multidimensional Wave Digital Filters."*J. VLSI Signal Processing* , 3, pp. 7-24, 1991.

[4] A. K. Jain, " Image Data Compression: a Review," in *Proc. of the IEEE*, vol. 69, no. 3, pp. 349-389, Mar. 1981.

[5] L. Lamport, " The Parallel Execution of DO Loops," *Commun. ACM SIGPLAN*, 17(2) Feb. 1974, pp. 82-93.

[6] L.-S. Liu, C.-W. Ho and J.-P. Sheu, " On the Parallelism of Nested For-Loops Using Index Shift Method," *Proc. of the 1990 International Conference on Parallel Processing*, 1990, Vol. II, pp. 119-123.

[7] A. Nicolau, " Loop Quantization or Unwinding Done Right," *Proc. of the 1987 ACM International Conference on Supercomputing*, Springer Verlag Lecture Notes on Computer Science 289, May 1987, pp. 294-308.

[8] M. J. Quinn, *Parallel Computing Theory and Practice*, New York, NY: McGraw-Hill, Inc., 1994.

[9] K. K. Parhi and D. G. Messerschmitt, " Concurrent Architectures for Two-Dimensional Recursive Digital Filtering". *IEEE Trans. on Circuits and Systems*, Vol. 36, No. 6, pp. 813-829, 1989.

[10] N. L. Passos and E. H.-M. Sha " Full Parallelism in Uniform Nested Loops using Multi-Dimensional Retiming," *Proc. of 23rd International Conference on Parallel Processing*, Aug., 1994, vol. II, pp. 130-133.

[11] N. L. Passos, E. H.-M. Sha, and L.-F. Chao, " Multi-Dimensional Interleaving for Time-and-Memory Design Optimization," in the *Proc. of the International Conference on Computer Design*, Oct., 1995, pp. 440-445.

[12] M. Wolf and M. Lam, " A Loop Transformation Theory and an Algorithm to Maximize Parallelism," *IEEE Trans. on Parallel and Distributed Systems*, vol. 2, n. 4, pp. 452-471, 1991.