FULLY PARALLEL HARDWARE/SOFTWARE CODESIGN FOR MULTI-DIMENSIONAL DSP APPLICATIONS

Michael Sheliga, Nelson Luiz Passos and Edwin Hsing-Mean Sha

Dept. of Computer Science & Engineering
University of Notre Dame, Notre Dame, IN 46556
{msheliga,npassos,esha}@bach.helios.nd.edu

ABSTRACT

The design of multi-dimensional systems using hardware/software codesign allows a significant improvement in the development cycle. This paper presents a technique that enables a design to have arbitrarily high throughput by using multi-dimensional retiming techniques while adjusting the composition of hardware and multiple software elements in order to satisfy the area requirements. A multi-dimensional graph representing the problem is transformed and scheduled such that all nodes are executed in a fully parallel way. The techniques presented are applicable to any problem which can be represented as a multi-dimensional data flow graph. Results are shown which illustrate the efficiency of the system as well as the savings achieved.

1. INTRODUCTION

Applications such as image processing, fluid mechanics, and weather forecasting, require high computer performance. Researchers and designers in those areas are looking for solutions to multi-dimensional problems through the use of parallel computers and/or specialized hardware. It is known that most of the current commercial computers are not able to achieve the computational speeds required by those applications. For example, the Advance Vision Systems proposed by ARPA, involving image formation of synthetic aperture radar (SAR) from collected phase histories, requires greater computational power than is normally available. At the same time, the development cycle of new systems needs to be reduced so as to speed up the process of getting access to system prototypes, in order to accelerate their evaluation and utilization. Hardware/software (hw/sw) codesign is assumed to be the technique that will provide such an improvement[5]. However, most of the existing results in this area do not consider the advantages that may be obtained by considering the multi-dimensionality of the applications. This paper presents algorithms that perform hw/sw codesign for the special area of *multi-dimensional systems*. Such algorithms allow the design cycle time to be significantly reduced.

In the hw/sw codesign area most research has focused on particular aspects of the design process, however, some automated tools have been developed to perform an entire hw/sw codesign. For example, COSYMA [9] uses a simulated annealing partitioning algorithm, and VULCAN II [7, 8] performs traditional hw/sw codesign for reactive systems which have inputs whose arrival times are unknown. However, these tools do not present any specific solution for the multi-dimensional applications. The technique proposed in this paper utilizes the multi-dimensional characteristics of the problems, in order to achieve full parallelism of all operations and optimize the scheduling process associated with the hardware/software distribution.

Hardware/software codesign requires not only an excellent planning strategy for the distribution of computational tasks among the software elements and specialized hardware, but also an optimized utilization of those resources. Such an optimization, usually, can be obtained by a good scheduling technique. Most of the previous results on scheduling focus on one-dimensional problems [2, 6, 12, 16]. Multi-dimensional hw/sw codesign requires the improvement of the parallelism inherent to multi-dimensional applications. Some studies focusing on uniform nested loop scheduling are similar to the solution of the multi-dimensional problem. For example, unimodular transformations [17], loop skewing [18] and loop quantization [1]. These techniques do not change the structure of the iterations, and therefore may not achieve a fully parallel solution.

More recent research has studied the scheduling of multi-dimensional applications. For example, the affine-by-statement technique [4] and the index shift method [13] are able to achieve a fully parallel execution of

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

0-8186-7243-9/96 $05.00 © 1996 IEEE

multi-dimensional tasks, utilizing algorithms based on linear programming techniques. However, these methods do not consider possible memory changes and consequently they may introduce new queues dependent on the problem size. The chained multi-dimensional technique, proposed in [15], can achieve the fully parallel solution in polynomial time. Therefore, we use this technique as the basis for our new algorithm.

Minimizing the system execution time and amount of hardware includes several important steps. Among these are deciding what operations to execute in software and what operations to execute in hardware, deciding on the number of software units (processors) and the amount and type of hardware to use, choosing the number of iterations to be executed in parallel (Should the hardware and software only focus on one iteration at a time. Or should a group of iterations be done in parallel?), choosing a good schedule vector, fully parallelizing and transforming the computational graph by retiming and unfolding, choosing the *shape* of the block of iterations to be executed in parallel, and minimizing the number of registers required. In addition, minimizing the queue size is critical since the queue size is proportional to the size of the array, and since it is affected by the scheduling vector.

In this paper we present a method for answering the above questions. Our method lays out guidelines for implementing a multi-dimensional hardware/software system. Once the above guidelines are used further details such as minimizing the number of registers during parallelization can be considered. Since the final design consists of one hardware partition and several software partitions it is referred to as the *multi-dimensional hardware/multi-software*, or *MD HMS*, system.

Our system designs any integrated circuit which can be represented as a multi-dimensional data flow graph. We make several important contributions in the paper. We prove that the schedule vector that minimizes the queue size for existing data dependencies can be found in constant time. We also show how to choose the schedule vector and the retiming vector so that the queue size does not change during retiming. We also prove that the shape of the group of iterations that are executed in parallel will not affect the queue size by more than a constant. It is further demonstrated how unfolding and retiming can be combined to make each block fully parallel.

The design time is reduced by following the general strategy of beginning with an all software implementation and then adding hardware until a solution is found. Another strategy that is used to decrease design complexity is to have each software unit execute one iteration, resulting in minimal communication and control overhead. When it is necessary to add hardware, it is added in proportion to the number of operations of each type. This not only allows all hardware units to be fully utilized, but it permits the complexity of the design process to be further reduced by grouping the hardware units so that they may be replicated in different parts of the design.

The MD HMS methodology is described in this paper as follows: Section 2 introduces definitions and terminology while Section 3 covers the assumptions of our system and explains the MD HMS codesign algorithms in detail. Section 4 demonstrates the effectiveness of the algorithm for several input systems. Finally, section 5 draws conclusions from the results obtained and discusses ideas for future research.

2. DEFINITIONS AND TERMINOLOGY

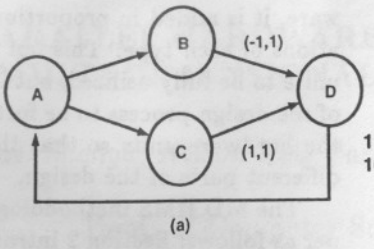
2.1. Basic Concepts

In this section we present some concepts related to the modeling of multi-dimensional problems, such as multi-dimensional data flow graphs, retiming, dependence vectors, iterations, and vector operations. A *multi-dimensional data flow graph* (MDDFG) G = (V, E, d, t) is a node-weighted and edge-weighted directed graph, where

- V is the set of computation nodes.
- $E \subset V \times V$ represents the set of dependence edges between nodes.
- d is a function from E to Z^n, representing the multi-dimensional delays between two nodes, with n being the number of dimensions. A three dimensional delay would, for example, be $(3, -2, 2)$.
- t is a function from V to the positive integers representing the computation time of each node. In order to simplify the presentation of the MD HMS system, we assume $t_v = 1 \; \forall \; v \in V$ for the remainder of the paper.

A two dimensional data flow graph (2DDFG) $G2 = (V, E, d, t)$ is a MDDFG with d being a function from E to Z^2. We adopt $d(e) = (d.x, d.y)$ as a general representation of any delay in a two-dimensional DFG. An example of a two-dimensional DFG and its equivalent Fortran code is shown in figure 1. For this example, $V = \{A, B, C, D\}$ and $E = \{e1 : (A, B), e2 : (A, C), e3 : (D, A), e4 : (B, D), e5 : (C, D)\}$ where, $d(e1) = d(e2) = d(e3) = (0, 0)$, $d(e4) = (-1, 1)$, $d(e5) = (1, 1)$.

An *iteration* is the execution of the loop body exactly once, i.e., the execution of each node in V exactly




```
DO 10 j = 0,n
   DO 11 k = 0, m
      D: d(k,j) = b(k+1,j-1) * c(k-1,j-1)
      A: a(k,j) = d(k,j) * .5
      B: b(k,j) = a(k,j) + 1.
      C: c(k,j) = a(k,j) + 2.
11    CONTINUE
10 CONTINUE
```

(b)

Figure 1: (a) MDDFG extracted from a Wave Digital Filter (b) equivalent Fortran code

once. Iterations are identified by a vector i, equivalent to a multi-dimensional index, starting from $(0, 0, \ldots, 0)$. Inter-iteration dependencies are represented by vector-weighted edges. For any iteration $\hat{j}$, an edge e from u to v with delay vector $d(e)$ means that the computation of node v at iteration $\hat{j}$ depends on the execution of node u at iteration $\hat{j} - d(e)$. An edge with delay $(0, 0, \ldots, 0)$ represents a data dependence within the same iteration. A legal MDDFG must have no zero-delay cycle, i.e., the summation of the delay vectors along any cycle can not be $(0, 0, \ldots, 0)$. Several techniques are available to verify that an MDDFG does not have a cycle [3, 11].

An equivalent *cell dependence graph (DG)* of an MDDFG G is the directed acyclic graph showing the dependencies between copies of nodes representing the MDDFG. Figure 2(a) shows the replication of figure 1(a), and figure 2(b) shows the cell DG with each node representing a copy of the MDDFG. A *computational cell* is the cell dependence graph node that represents a copy of an MDDFG, excluding the edges with delay vectors different from $(0, 0, \ldots, 0)$, i.e., a complete iteration. A cell is considered an atomic execution unit.

The notation $u \xrightarrow{e} v$ means that e is an edge from node u to node v. The notation $u \overset{p}{\rightsquigarrow} v$ means that p is a path from u to v. The delay vector of a path $p = v_0 \xrightarrow{e_0} v_1 \xrightarrow{e_1} v_2 \ldots \xrightarrow{e_{k-1}} v_k$ is $d(p) = \sum_{i=0}^{k-1} d(e_i)$ and the total computation time of a path p is $\sum_{i=0}^{k} t(v_i)$.

To manipulate MDDFG characteristics represented by vector notation, such as the delay vectors, we make use of component-wise vector operations. Considering two two-dimensional vectors P and Q, represented by their coordinates $(P.x, P.y)$ and $(Q.x, Q.y)$, an example of arithmetic operation is $P + Q = (P.x + Q.x, P.y + Q.y)$. The notation $P \cdot Q$ indicates the inner product between P and Q, i.e., $P \cdot Q = P.x * Q.x + P.y * Q.y$.

A *schedule vector* S is the normal vector for a set of parallel equitemporal (equally spaced in time) hyperplanes that define a sequence of execution of a cell

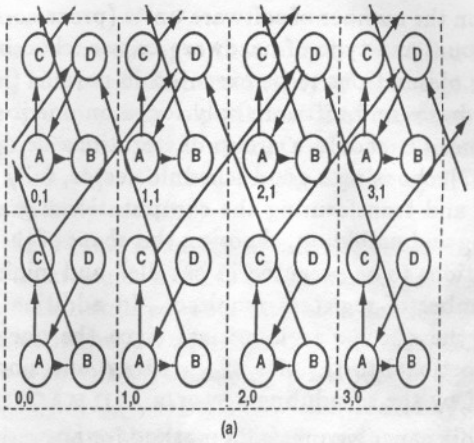


Figure 2: (a) DG based on the replication of an MDDFG, showing iterations starting at (0,0).

dependence graph. The existence of a schedule vector prevents the existence of a cycle. We say that an MDDFG $G = (V, E, d, t)$ is *realizable* if there exists a schedule vector S for the cell dependence graph with respect to G, i.e., $s \cdot d > 0$ for any $d \in G$ [14].

2.2. Retiming a Multi-Dimensional Data Flow Graph

A *multi-dimensional retiming* r is a function from V to Z^n that redistributes the nodes in the original dependence graph created by the replication of an MDDFG G. A new MDDFG G_r is created, such that each iteration still has one execution of each node in G. The retiming vector $r(u)$ of a node $u \in G$ represents the offset between the original iteration containing u, and the one after retiming. The delay vectors change accordingly to preserve dependencies, i.e., $r(u)$ 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 $u \xrightarrow{e} v$

and $d_r(l) = d(l)$ for every cycle $l \in G$. After retiming, the execution of node u in iteration i is moved to the iteration $i - r(u)$. A *two-dimensional retiming* r is a function from V to Z^2 that redistributes the nodes in the original dependence graph created by the replication of a 2DDFG G, resulting in a new 2DDFG G_r, such that each iteration still has one execution of each node in G.

For example, figure 3(a) shows the MDDFG from figure 1(a) retimed by the function $r = (1, 0)$. Figure 3(b) shows the modified Fortran code. The critical paths of this graph are the edges $A \rightarrow B$ and $A \rightarrow C$ with an execution time of 2 time units.

The retimed cell DG, for the example in figure 1(a), is shown in figures 4(a) and (b), where the nodes originally belonging to iteration $(0, 0)$ are marked. A possible schedule vector for the retimed graph is $s = (1, 3)$. Figure 5(a) shows an illegal retiming function applied to the same example. By simple inspection of the cell dependence graph in figure 5(b) we notice the existence of a cycle created by the dependencies $(1, 0)$ and $(-1, 0)$.

A *prologue* is the set of instructions that are moved in directions x and y, in a two-dimensional retiming, and that must be executed to provide the necessary data for the iterative process. In our example shown in figure 4(a), the instruction D becomes the prologue for that problem. The *epilogue* is the other extreme of the DG, where a complementary set of instructions will be executed to complete the process. In the example, the reduction of the critical path, as seen in figure 3, results in a saving of approximately one third of the total execution time.

2.3. Terminology

In order to simplify the reader's understanding of the paper some important terminology and variable names, as well as background assumptions, are summarized here.

- This paper is concerned with implementing large numbers of iterations (100 or more). It is assumed that the iterations to be executed may be represented by a square array of size N. The results obtained for a square array may easily be expanded to include rectangular arrays.
- The schedule vector for the processor array is denoted by S.
- The delays for a MDDFG are denoted by the set D. $DS = \sum_{d \in D} d$.
- The number of nodes per scheduling hyperplane is denoted by NH. This is equal to $\frac{N}{max(S_x, S_y)}$ for a square array.
- The retiming vector for the MDDFG is denoted by r. For reasons that are explained below, r will always be perpendicular to S. We use $r \perp S$ to indicate this.
- P is used to denote the number of software units, or processors, available. All of the P processors are identical.
- Each processor contains the same number of functional units. For example, the processors may contain four adder units, two comparator units, and one multiplier unit.
- The *block shape* is the shape of the group of computational cells that are to be executed in parallel using the existing hardware and software. For example, if we had eight software units, each executing one computational cell, with a schedule direction of $(0, 1)$, some possible block shapes would be 1x8, 2x4, and 8x1. Of course, intra-block dependencies of the form $(0, 0, \ldots, 0)$ are not allowed.

3. ALGORITHM

In order to minimize the design time and the amount of hardware in the final design, the MD HMS system follows the general strategy of beginning with all operations in software. Initially, the maximum number of possible software units are used for the given area constraint. Hardware is then added until the maximum area is exceeded. When this happens software is eliminated and replaced by more hardware. This continues until the design constraints are met, or it is determined that they can not be satisfied.

The MD HMS algorithm is begun by calculating S, the schedule direction. This is important since it determines the queue size. Since the system may be made fully parallel, the area depends in large part on the queue size. The retiming vector r is then chosen to be perpendicular to S. Next, the maximum amount of software that may be used is inserted, and the iterative process of calculating system parameters, eliminating software, and inserting hardware is begun. For each iteration four steps are performed. First the number of iterations to be executed in parallel are calculated. Second, the shape of these iterations is chosen to be aligned to the scheduling hyperplane. Third, the iterations within the block are unfolded. Fourth, retiming is used to make all intra-block operations parallel.

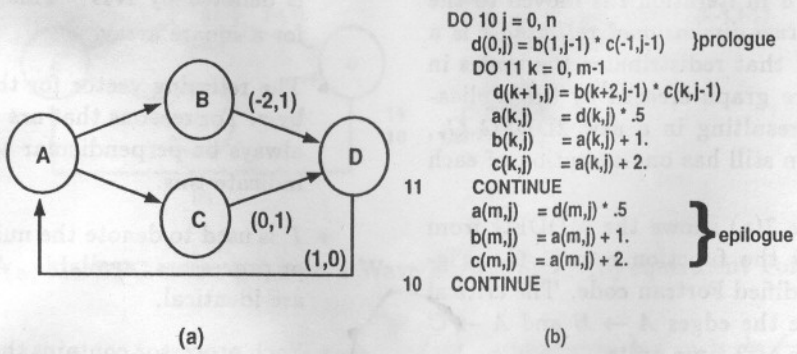


Figure 3: (a) MDDFG after retiming by r(D)=(1,0) (b) equivalent Fortran code

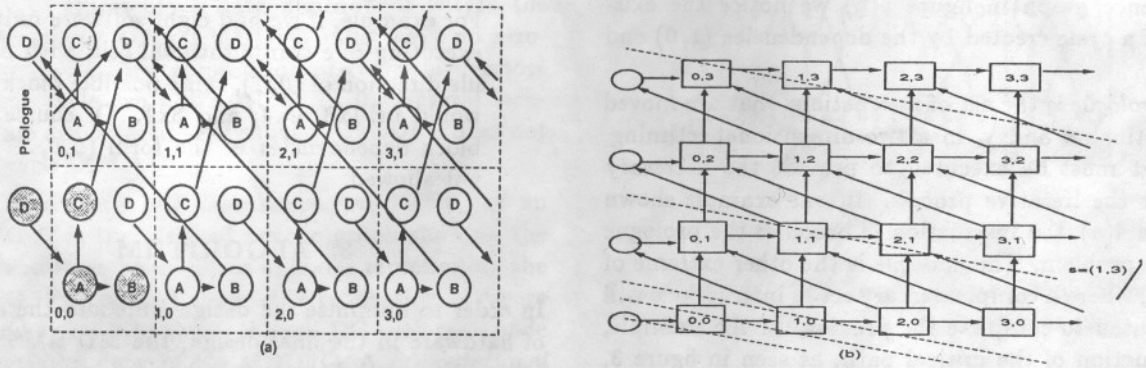


Figure 4: (a) DG based on the replication of an MDDFG, after retiming. (b) same DG represented by computational cells.

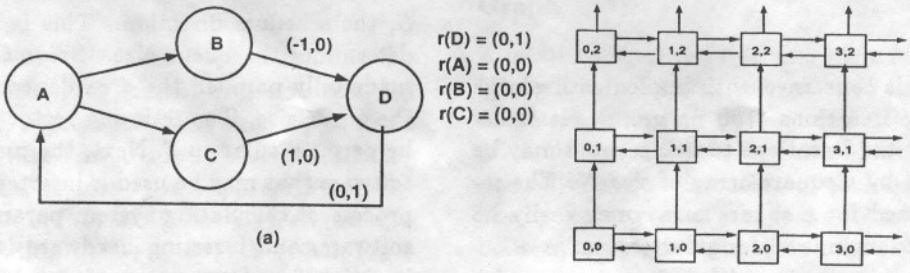


Figure 5: (a) Example of illegal retiming. (b) DG showing cycles in the x-direction.

3.1. Selection of the retiming vector r and the scheduling vector S

In most multi-dimensional systems the vast majority of edges do not have delays. In order to make the system fully parallel, we must add delays to these edges. However, each delay that is added will result in either a register or a queue entry during execution. Since the queue size is directly related to the problem size ($OrderN$), it is beneficial to have as many registers and as few as queues as possible. It is also known that if S is perpendicular to some delay d, then d will result in a fixed number of registers [15]. Hence, our system always chooses $r \perp S$, with r being a constant for all retimings. In this manner all edges that are originally $(0, 0, 0 \ldots 0)$ will result in registers, not queues.

In order to choose $r \perp S$, we must first choose S. Hence, the MD HMS system begins by choosing an appropriate value for the schedule vector S. The scheduling vector must be chosen so that $S \cdot d > 0 \; \forall \; d \in D$, (otherwise we would be attempting to execute iterations that depend on other iterations that have not yet been executed!). We refer to the region defined by the above inequalities as the *legal scheduling vector region.* By definition the legal scheduling vector region must be continuous and, intuitively, within ninety degrees of all delays. The legal scheduling vector region in figure 6 is represented by the shaded region. Note the extremes of this region, $(1, 1)$ and $(0, 1)$, are ninety degrees from the extreme delays of $(-1, 1)$ and $(1, 0)$, respectively. We define S_{cw} as the boundary of the legal schedule vector region in the clockwise direction, and S_{ccw} as the boundary of the legal schedule vector region in the counter clockwise direction. In figure 6, $S_{cw} = (1, 1)$ and $S_{ccw} = (0, 1)$. We also note that the magnitude of the scheduling vector is unimportant. For example, scheduling in the $(1, 1)$ direction is equivalent to scheduling in the $(3, 3)$ direction.

The choice of the scheduling vector will have a crucial influence on the rest of the system. In particular, it is known that $NH \times \sum_{d \in D} \; S \cdot d$ will be equal to the queue size at the end of the execution of each hyperplane. NH is the number of nodes per scheduling hyperplane, and is equal to $\frac{N}{\max(|S_x|, |S_y|)}$ for a square array. It may be noted that since we chose $r \perp S$, the queue size will remain constant, even after retiming.

Hence, we wish to minimize $NH \times \sum_{d \in D} \; S \cdot d$ under the constraint $S \cdot d > 0 \; \forall \; d \in D$. It may be shown that the S that results in the minimum queue size is one of the extremes of the legal scheduling vector region for a square iteration space.

Best Schedule Vector Problem: Consider an $N \times N$ multi-dimensional data flow graph, γ, with a set of delay vectors, D, that define a (continuous) legal schedule vector region.

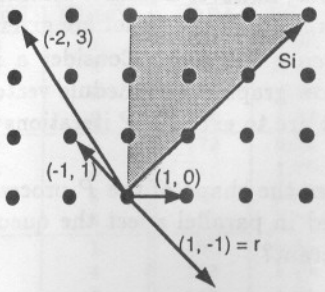


Figure 6: The legal scheduling vector region for $r_1 = (-1, 1), r_2 = (-2, 3)$, and $r_3 = (1, 0)$, and an illegal schedule vector $S_i = (1, -1)$.

Question: What scheduling vector S will result in the minimal queue size, Q?

Theorem 3.1 *The best scheduling vector will be one of the extremes of the legal scheduling vector region.*

The proof of this theorem is not shown here due to space considerations. Intuitively the result means that S should be as far away from the delay vectors as possible, but within ninety degrees of all delay vectors. In figure 6, of the many legal values for S, the one that will result in the minimum number of queues must be in the $(1, 1+)$ or $(0+, 1)$ directions. These two vectors are ninety degrees from the extreme vectors of $(-1, 1)$ and $(1, 0)$. Hence the schedule vector could be chosen to be $(9, 10)$. This vector results in an actual queue size that is very near the minimal queue size as well as a relatively small prologue. It is important to note that we use $S \cdot d > 0$ not $S \cdot d \geq 0$. If S were exactly ninety degrees away from an original delay vector d, then r would be 180 degrees away from some d. In this case, a cycle would be produced upon retiming.

3.2. Block Selection and Parallelization

Once S has been selected (and as a direct result r) the iterations to be scheduled in parallel, or *scheduling block*, are chosen. For example, if there are four processors, no hardware, one delay vector $d = (1, 5)$ and $S = (1, 0)$, we could choose to schedule four horizontally aligned iterations, four vertically aligned iterations, or a square two by two group of iterations. It may be proven that the shape of the block of iterations chosen will not have an effect on the queue size by more than a constant. The proof is not shown due to

space considerations, however a formal definition of the problem and the results of the proof are given below.

Block Equivalence Problem: Consider a multi dimensional data flow graph, γ, a schedule vector S, and P processors that are to execute P iterations in parallel.

Question: Does the shape of the P processors that are to be executed in parallel affect the queue size by more than a constant?

Theorem 3.2 *The block shape does not affect the queue size by more than a constant.*

Since the block shape selection will not affect the queue size by more than a constant we may look to some other criteria to determine it. We do not consider this problem further in this paper, but leave it as an open problem. Instead, the block shape is chosen to be perpendicular to S in all cases (aligned with the scheduling hyperplane). By choosing the block shape in this manner will assure that there will not be zero delay dependencies between iterations in the block. This is guaranteed since S is perpendicular to r. For example, if $d = (1,1)$, $S = (0,1)$, $r = (1,0)$ and $P = 4$, then four horizontally aligned iterations would be executed in parallel. Note that it is impossible to have a delay in the $(1,0)$ direction between the iterations in this block.

Once we have decided on the block shape of the iterations that are to be executed in parallel, we unfold the graph and then apply internal multi-dimensional retiming inside each block. By retiming using this method, we assure ourselves that all nodes in the unfolded graph (block of iterations) are fully parallel. Hence, all hardware units may be fully utilized.

3.3. Software and Hardware Allocation

Given a fixed number of processors, we assign each processor to one iteration per block. In this manner communication between processors may be ignored since all such communication will take place between iterations in different blocks. Furthermore, since each processor is identical, only one control unit will need to be designed for all processors. In a similar manner only one global control unit will need to be implemented on the chip. When software is eliminated in favor of hardware, the number of functional units to be added for each operation type must be decided. Hardware is added in proportion to the number of operations of each type. In this manner all hardware may be fully utilized. and the hardware design may be reused. For every four adders and three multipliers that are added, the same layout and control unit may be replicated.

4. EXPERIMENTAL RESULTS

This section presents experimental results for the three two-dimensional problems, as shown in figure 7. A few comments will be made to help explain the results, with the pulse-code modulation device[10] in mind. The modulation device contains five adders and three multipliers, while the software contains one adder and one multiplier. It is assumed that each software unit is able to execute one addition and one multiplication in one time unit. The execution rate of hardware units is twice that of the corresponding software units. The hardware and software units are assumed to take up the areas given at the bottom of the figure. The first four columns of the figure represent the number of cpus, the total number of hardware units (adders and multipliers), the number of adders, and the number of multipliers.

It is useful to introduce five addition units for every three multiplication units, when replacing software with hardware. Once this is done these units may be grouped together. The "Hardware Groups" (HWG) column shows the number of such groups. The "Hardware Extra" (HWE) column is the number of hardware units that can not be placed in such groups, while the next two columns show the number of adders and multipliers, respectively, that comprise the extra hardware.

The system begins using the maximum number of software units possible. If the number of iterations per time unit is less than the desired value of two, one software unit is eliminated, and additional hardware is inserted. This process would normally stop once the desired value of iterations per time unit is reached, as indicated by the asteriks, however, results were generated in this case until an all hardware solution was reached.

5. CONCLUSION

Multi-dimensional systems are important in areas such as image processing, fluid mechanics, and weather forecasting. In this paper we considered the MD HMS system which automates the hw/sw codesign process for multi-dimensional systems. Since design time is often the most important factor, our system attempts to implement as much of the system as possible in software provided that chip area and timing constraints are met. Results that illustrate the savings which are possible with the MD HMS algorithm are presented for several two-dimensional graphs.

6. REFERENCES

[1] A. Aiken and A. Nicolau, " Loop Quantization: An Analy-

	Units				Hardware Grouping				Results		
Filter	CPU	HW	ADDS	MULTS	HWG	HWE	ADDS	MULTS	AREA	IPTU	IWR
PCM	3	3	2	1	0	3	2	1	172	1.26	0.71
	2	9	6	3	1	1	1	0	166	2.40*	1.14
	1	16	10	6	2	0	0	0	172	4.20	1.86
	0	22	14	8	2	6	4	2	166	5.33	2.29
WDF	3	3	2	1	0	3	2	1	172	0.88	0.64
	2	9	6	3	0	9	6	3	166	1.75	1.11
	1	16	10	6	1	4	2	2	172	2.63*	1.55
	0	23	15	8	1	11	7	4	171	3.75	2.12
IIR	3	2	1	1	0	2	1	1	167	0.63	0.50
	2	8	4	4	0	8	4	4	168	1.25	1.00
	1	14	7	7	0	14	7	7	169	1.88	1.50
	0	19	9	10	1	4	2	2	165	2.50*	1.80

PCM: Pulse Code Modulation (5 Adder, 3 Mult)
WDF: Wave Digital Filter (8 Adder, 4 Mult)
IIR: Infinite Impulse Response (7 Adder, 8 Mult)

Area Desired = 175	Desired IPTU = 2.0
Adders Per CPU = 1	Mults Per CPU = 1
CPU Area = 50	
Adder Area = 5	Multiplier Area = 12
IPTU = Iterations Per Time Unit	IWR = Iterations Without Retiming
HWE = Hardware Extra	HWG = Hardware Groups

Figure 7: Results for several two-dimensional problems

sis and Algorithm," Technical Report 87-821, Department of Computer Science, Cornell University, March 1987.

[2] L.-F. Chao, A. LaPaugh, and E. H.-M. Sha, " Rotation Scheduling: A Loop Pipelining Algorithm," *Proc. 30th ACM/IEEE Design Automation Conference* , Dallas, TX, pp. 566-572, June, 1993.

[3] E. Cohen and Nimrod Megiddo, " Strongly Polynomial-Time and NC Algorithms for Detecting Cycles in Dynamic Graphs," *Proc. 21th ACM Annual Symposium on Theory of Computing*, 1989, pp. 523-534.

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

[5] D. Gaski, F. Vahid, S. Narayan, and J. Gong, "Specification and Design of Embedded Systems," Prentice-Hall, Inc, Englewood Cliffs, NJ, 1994.

[6] G. Goosens, J. Wandewalle, and H. de Man, " Loop Optimization in Register Transfer Scheduling for DSP Systems," *Proc. ACM/IEEE Design Automation Conference*, 1989, pp. 826-831.

[7] R. Gupta and G.DeMicheli, "Hardware-Software Cosynthesis for Digital Systems," *IEEE Design and Test of Computers*, October 1993, pp. 29-41.

[8] R. Gupta and G.DeMicheli, "System Level Synthesis Using Re-programmable Components," *The European Conference on Design Automation*, March, 1992 pp. 2-7.

[9] J. Henkel, T. Benner, and R. Ernst, "Hardware Generation and Partitioning Effects in the COSYMA System," *2nd International Workshop on Hardware-Software Co-Design*, Workshop Handout, 1993.

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

[11] S. R. Kosaraju and G. F. Sullivan, " Detecting Cycles in Dynamic Graphs in Polynomial Time," *Proc. 20th ACM Annual Symposium on Theory of Computing*, 1988, pp. 398-406.

[12] T.-F. Lee, A. C.-H. Wu, D. D. Gajski, and Y.-L. Lin, " An Effective Methodology for Functional Pipelining", in *Proc. of the International Conference on Computer Aided Design*, December, 1992, pp. 230-233.

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

[14] S. Y. Kung, *VLSI Array Processors*, Englewood Cliffs, NJ: Prentice Hall, 1988.

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

[16] R. Potasman, J. Lis, A. Nicolau, and D. Gajski, " Percolation Based Scheduling". *Proc. ACM/IEEE Design Automation Conference* , pp. 444-449, 1990.

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

[18] M. Wolfe, " Loop Skewing: the Wavefront Method Revisited,". *International Journal of Parallel Programming*, Vol. 15, No. 4, pp. 284-294, August 1986.