

A Study of Software Pipelining for Multi-dimensional Problems

Reynold Bailey, Delvin Defoe, Ranette Halverson, Richard Simpson, Nelson Passos

Department of Computer Science

Midwestern State University

Wichita Falls, TX 76308

Abstract

Computational performance of multi-dimensional applications, such as image processing and fluid dynamics, is highly dependent on the parallelism embedded in nested loops found in the software implementation. Software pipeline techniques for multidimensional applications can be roughly divided into two groups. One focused on optimizing the innermost loop body, while another group attempts to exploit the parallelism across the loop dimensions. This paper presents a comparative study of these two groups of techniques through two methods representative of those classes, *Iterative Modulo Scheduling* and *Push-up Scheduling*.

1. Introduction

Multi-dimensional (MD) computation, such as image processing and fluid dynamics, can be modeled and coded as nested loops, which contain groups of repetitive operations. The parallelism within the loop body can usually be exploited by a software pipeline technique. A software pipeline is a class of compiler parallelization techniques, which changes the execution order of some or all of the operations in the loop body to allow them to be executed in parallel. The optimization is usually processed in a resource-constrained environment, i.e., the functional units available for the operations are limited.

Software pipeline techniques for multidimensional applications can be roughly divided into two groups. One group is focused on optimizing the innermost loop body [1, 12, 13], while another attempts to exploit the parallelism across the loop dimensions [2, 8, 9, 10]. Various software pipeline techniques have been developed and published, and the body of work related to software pipeline has primarily focused on establishing the formalism and algorithms for the techniques. However, little work has been done to compare these two groups of techniques in terms of schedule quality as well as computational complexity. This paper presents a comparative study of these two groups of techniques through two methods representative of those classes.

The algorithms chosen for this study are *Iterative Modulo Scheduling* [12] and *Push-up Scheduling* [9]. *Iterative Modulo Scheduling* is a framework within which

a software pipeline algorithm of innermost loops is defined. *Iterative Modulo Scheduling* is an important software technique subsequently used as the basis for numerous other algorithms. *Push-up Scheduling* is a scheduling technique that was originally designed for solving multi-dimensional problems. The heart of *Push-up Scheduling* is the chained MD retiming technique [9, 11].

2. Background

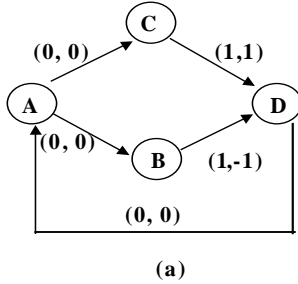
Push up Scheduling

A MD computation is modeled as a cyclic data flow graph, which is called an MD data flow graph (MDFG). A valid MDFG is a directed graph represented by the tuple (V, E, d, t) , where V is the set of operation nodes in the loop body, E is the set of directed edges representing the dependence between two nodes, a function d represents the MD-delay between two nodes, and t is the time required for computing a certain node [9]. An example of a valid MDFG and its corresponding loop body is presented in Figure 1.

Push-up Scheduling was introduced by Passos and Sha [9] for scheduling MD problems. The objective of this technique is to legally change the delays of the edges in an MDFG such that non-zero delays on all edges may be obtained in order to achieve full parallelism.

There are three basic functions used in determining an optimal schedule for this technique. One is the earliest starting time (control step) for computing node u , $ES(u)$, which can be obtained by $ES(u) = \max\{1, ES(v_i) + t(v_i)\}$, where v_i is a member of the set of nodes preceding u by an edge e_i and $d(e_i) = (0, 0)$. In order to simplify the examples presented in this paper, we assume $t(v_i) = 1$. The second required function is $AVAIL(fu)$, which returns the earliest control step in which the functional unit fu is available. Given an edge $e: u \rightarrow v$, such that v can be scheduled to $ES(v)$ and $d(e) = (0, 0)$, a MD retiming of u is required if $ES(v) > AVAIL(fu_v)$. This retiming allows the schedule of v at time $AVAIL(fu_v)$. This means that a node does not need to be rescheduled several times during the process. The third function $MC(u)$ gives the number of extra non-zero delays required by u along any zero-delay path to u ; this value is then used to calculate the

actual retiming function for node u . *Push-up Scheduling* generates a retiming vector $r(u)$ for each node in the MDFG such that the retimed delay of each edge is given by $d_r(e_j) = d(e_j) + r(u)$, where e_j represents all the outgoing edges from u , and $d_r(e_i) = d(e_i) - r(u)$, where e_i represents all the incoming edges of u . A new scheduling table is generated by reassigning the $ES(u)$ for each node in the MDFG.



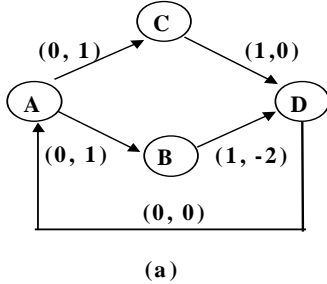
```

for i = 0 to ...
  for j = 0 to ...
    D(i, j) = B(i-1, j+1) + C(i-1, j-1)
    A(i, j) = 5 + D(i, j)
    B(i, j) = 3 * A(i, j)
    C(i, j) = 6 * A(i, j)
  end j
end i

```

(a)

Figure 1. (a) A valid MDFG. (b) Corresponding loop body.



```

for i = 0 to ...
  for j = 0 to ...
    D(i, j+1) = B(i-1, j+2) + C(i-1, j)
    A(i, j+1) = 5 + D(i, j+1)
    B(i, j) = 3 * A(i, j)
    C(i, j) = 6 * A(i, j)
  end j
end i

```

(a)

Figure 2. (a) A retimed MDFG. (b) Its corresponding loop body.

The essence of *Push-up Scheduling* is the chained MD-retiming [11], which pushes the MD-delay from incoming edges of u to its outgoing edges. If an incoming edge has zero delay, it is necessary to apply the same retiming function to that incoming node and to keep the sum of

delays between each pair of nodes unchanged. A retimed MDFG for the example in Figure 1 is presented in Figure 2. *Push-up Scheduling* is implemented in the OPTIMUS algorithm presented in [9]

2.2 Iterative Modulo Scheduling (IMS)

IMS was introduced by Rau and Glasser [13]. Modulo scheduling was designed for scheduling the innermost loop body, however, it can be applied to any *MD* problem. The basic idea is to find a valid *Initiation Interval (II)* cycle. When a statement within the loop body is scheduled at time c , the instance of c in iteration i is scheduled as time $c+i*II$. The goal of *IMS* is to find the smallest II while satisfying the resource and recurrence constraints.

In order to use *IMS*, two pseudo operations must be added to the dependence graph: *START* and *STOP*. *START* is assumed to be the predecessor of all the other operations, and *STOP* is assumed to be the successor of all other operations. *IMS* begins by calculating a minimum initiation interval (*MII*). *MII* is determined by $\max(ResMII, RecMII)$, where *ResMII* is a resource constraint *MII*, and *RecMII* is the recurrence constraint *MII*. The *ResMII* is constrained by the most heavily utilized resource along any execution path of the loop. If an execution path p uses a resource r for c_{pr} clock cycles and there are n_r copies of this resource, the *ResII* is

$$ResII = \max_{p \in P} \left(\max_{r \in R} \left\lfloor \frac{C_{pr}}{n_r} \right\rfloor \right)$$

where P is the set of all execution paths and R is the set of all resources.

The recurrence constraint occurs when an operation in one iteration of the loop has a direct or indirect dependence upon the same operation from a previous iteration. This implies a circuit in the dependence graph. If a dependence e in a cycle of the dependence graph has latency l_e and the operations connected to it are d_e iterations apart, then *RecII* is

$$RecII = \max_{c \in C} \left\lfloor \frac{\sum_{e \in E_c} l_e}{\sum_{e \in E_c} d_e} \right\rfloor$$

where C is the set of all dependence cycles and E_c is the set of edges in dependence cycle c . The function *ComputeMinDist* published in [12] is used to compute *ResII*.

For a given II , the $[i, j]$ entry of *MinDist* in *ComputeMinDist* specifies the minimum permissible interval between the time at which operation i is scheduled and the time at which j , from the same iteration, is scheduled. If the value of the entry is $-\infty$, it means there is no path from i to j . If *MinDist* $[i, i]$ is positive for any i , it implies that i must be scheduled later than itself in the same iteration, which is clearly impossible and a larger value of II must be tried. Once

the MII is determined, the scheduling process can start with the initial value of MII , which consecutively calls the IterativeSchedule. If it fails to obtain a legal schedule at any II , it tries the next larger II until a legal schedule is found.

The IterativeSchedule works as follows: it schedules all the operations within the loop according their priorities and dependence constraints such that all higher priority operations are scheduled first. The scheduling priorities are assigned in such a way that the most constrained have the highest priority. Every operation is then scheduled at the earliest start time (MinTime) unless there is a resource conflict. If there is a resource conflict, the operation is scheduled at the next available time slot. If an operation cannot be scheduled by its maximum allowable time, which is determined by $MaxTime = MinTime + II - 1$, this trial fails and a larger value of II must be used.

Once the schedule is determined, the motions of the code, i.e. the movements of operations from one iteration to another, are also determined from their scheduled time and II . The retiming vector for each operation necessary for our comparison can be obtained by

$$r_i = 1 - \left\lfloor \frac{ES_i}{II} \right\rfloor$$

where ES_i is the earliest start time of operation i . The relationship between the retiming vector and the schedule time is illustrated in Figure 3.

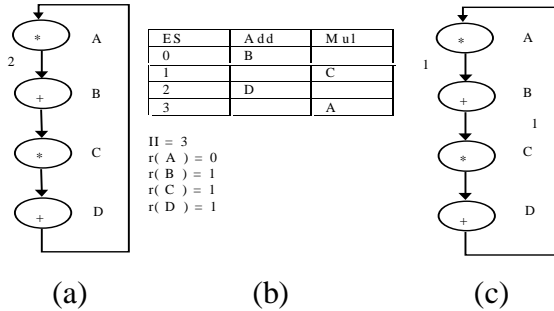


Figure 3. (a) Original dependence graph. (b) Schedule table and retiming vector. (c) Retimed dependence graph.

3. Experiments

A software system, named Precise-MD, was built to carry out the *Push-up Scheduling* process. A valid MDFG is the input to the PRECISE-MD system. After retiming the nodes, each operation is reassigned to the earliest control step in which the corresponding functional unit is available. Any delays are updated as necessary. Finally, the resulting MDFG is returned to the user. All these operations are performed in a user-friendly way, through a graphical user interface [7].

In *IMS*, the user is responsible for entering the dependence graph and the maximum number of steps

attempted. A MII is then calculated by a private member function. By using this MII as the initial trial value of II , a public member function can be invoked to start the scheduling process. Finally, a legal schedule table and the final II are returned to the user.

In the following sections, various experiments are discussed relating to these two methods. The experiments are not intended to show the applicability of these two methods, which has been proven by the original papers [9, 12]. The experiments are directed to determine the types of the problems they can solve, what schedule length they can obtain for a given problem and how parameters will impact on the schedule processes. To ensure a fair comparison, the following assumptions were made: every scheduling process will be run on the same machine, two kinds of units are available for each test - adders and multipliers, and each unit will take one time unit to finish its job. These assumptions have no bias against either method in terms of applicability and schedule quality.

Another issue to be considered when using *IMS* on *MD* problems is the linearization of the problem. *IMS* was developed for one-dimensional problems, thus in order to use it for *MD* cases, it was necessary to change them into one-dimensional problems. The method adopted in this study was to feed the iteration number for the innermost loop body of the original *MD* problems. Thereafter, a new dependence graph with one-dimensional delays can be obtained by $d(i) = dx*N + dy$ where, $d(i)$ is the new delay for each node, dx is the original delay in the x direction (outer loop), and dy is the original delay in y direction (inner loop). The direct result of doing this is to translate the outer loop dependencies to inner loop dependencies. An example is shown in Figure 4.

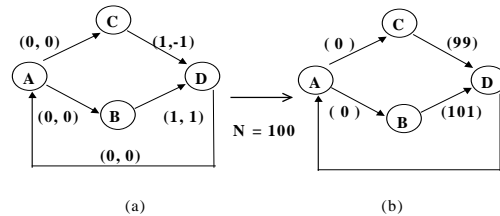


Figure 4. Linearization of two-dimensional problem to one-dimensional problem.

The experiments were roughly divided into four categories, innermost loop problems, nested loop problems, single units versus multiple units, and impact of the number of inner loop iterations.

Innermost Loop Problem

Innermost loop problem is defined as a problem for which the outer loop dependencies for all the edges are zero. The dependencies only exist in the inner loop as shown in Figure 5. Two cases are used for the tests.

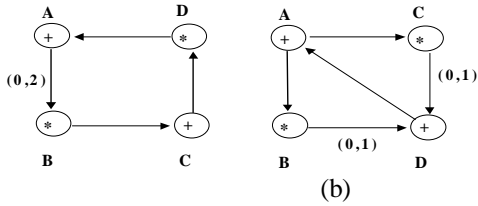


Figure 5. (a) *Problem a* for innermost loop problem. (b) *Problem b* for innermost loop problem.

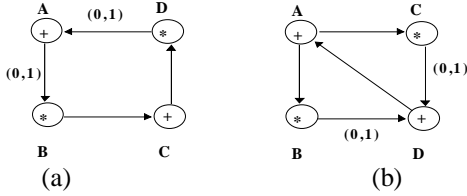


Figure 6. Retimed graphs for (a) of *problem a* (b) *problem b* when, solved by IMS.

CS	Adder	Multiplier
0		B
1	C	
2		D
3	A	B

$\Pi = 3$

(a)

CS	Adder	Multiplier
0	D	
1	A	
2		B
3		C

$\Pi = 4$

(b)

Table 1. IMS Scheduling Results (a) *Problem a* (b) *Problem b*, both solved by IMS with budget ratio = 2.

Assume that in each case there is only one adder unit and one multiplier unit. The schedule tables and retimed graph obtained by *IMS* are shown in Figure 6 and Table 1. The solutions from *Push-up Scheduling* are presented in Table 2 and Figure 7. As one can see from Table 1, the schedule length computed for *problem a* by the *IMS* is 3 ($\Pi=3$), while the schedule length computed for *problem b* is 4. As shown in Table 2, the schedule length of both problems obtained by *Push-up Scheduling* is 2.

Nested Loop Problem

In the case of nested loops, both outer loop and inner loop dependencies may exist, which means non-zero delays may appear in dx or dy for any edge in an MDFG. The problem used for the experiment is shown in Figure 8. It is slightly different from *problem b* in the previous section in that the delays change when moving from one dimension to two dimensions. Assume that one adder and one multiplier are available for both methods. The solutions for this problem are shown in Table 3 and Figure 9. Note that both methods found a schedule length of 2, and produced the same retimed MDFGs.

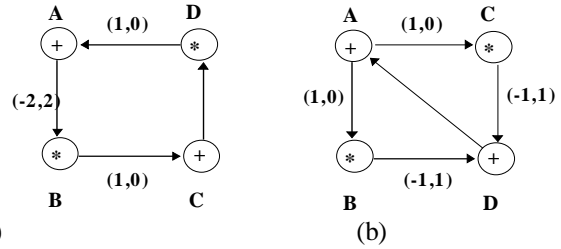
CS	Adder	Multiplier
0	C	B
1	A	D

(a)

CS	Adder	Multiplier
0	D	B
1	A	C

(b)

Table 2. Schedule table for problems solved by *Push-up Scheduling* (a) *problem a* (b) *problem b*



(a)

(b)

Figure 7. Retimed graphs for (a) *problem a* (b) *problem b* when solved by *Push-up Scheduling*

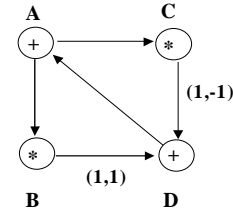


Figure 8. A nested loop problem

CS	Adder	Multiplier
0	D	
1	A	
2		B
3		C

$\Pi = 2$

(a)

CS	Adder	Multiplier
0	D	B
1	A	C

(b)

Table 3. Nested loop scheduling results (a) original schedule (b) solved by both PUS and IMS (budget ratio = 2 and $N = 100$)

Multiple Units Problem

In the previously described experiments, the number of functional units was assumed to be one for each type of operation. To make the experiments non-trivial, it is assumed that there are two adders and two multipliers. As both methods obtained the shortest schedule length in the Nested Loop Problem section, the problem used for that section will again be used in this test. The budget ratio is still 2, and the iteration number of the innermost loop fed to IMS is 100. The results from IMS and PUS are shown in Table 4 and Figure 10. Note that both methods achieved optimal solutions in this case.

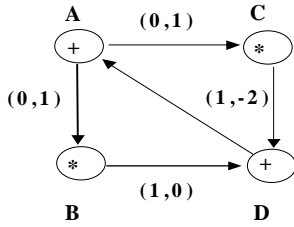


Figure 9. The retimed MDFG for the nested loop problem solved by both IMS and PUS

CS	Adder	Adder	Multiplier	Multiplier
0	D			
1		A		
2			B	C

$II = 1$

(a) original schedule

CS	Adder	Adder	Multiplier	Multiplier
0	D	A	B	C

(b) solved by IMS and PUS

Table 4. Schedule table for Multiple Units Problem by both IMS and Push-up.

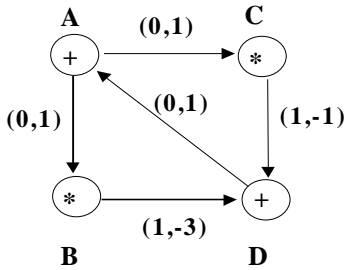


Figure 10. Retimed Graph - Multiple Units Problem.

Impact of the Number of Inner Loop Iterations

As stated in the previous section, a linearization process was added to the original *IMS* algorithm to make it applicable for *MD* problems. The process simply takes a pseudo iteration number of the inner loop from the user input, and uses this number to generate a new dependency graph, which has delays only in one direction. To determine if this user input number would affect the schedule length, a series of tests were performed based on different iteration numbers, resulting in no change to the schedule lengths found.

4. Discussion

In previous sections, the solutions given by *IMS* and *Push-up Scheduling* are sometimes different. These differences can be explained as follows.

Approach - During the study a basic difference in the approaches of *IMS* and *Push-up Scheduling* was noted. The approach of *Push-up Scheduling* is based on the idea of “move-then-schedule”. The intent is to eliminate all dependencies within the same iteration so that full parallelism may be achieved. In *Push-up Scheduling*, this is done by legally retiming the nodes in an MDFG in order to find a new MDFG with non-zero delays for all its edges. It has been proven that for *MD* problems, such loop transformation is always achievable. But for one-dimensional problems, it may not be attainable in some cases [11]. The results of Section 3 demonstrated that *Push-up Scheduling* can always find the optimal schedule. On the other hand, the approach of *IMS* is “schedule-then-move”. This approach is directly focused on the creation of a legal schedule for one iteration of the loop. After a legal schedule is found, the subsequent code motion can be determined by the schedule table and the *II*. In this study, the *MD* problem is still treated as a one-dimensional (single loop) problem to the *IMS* by linearization. However, *IMS* failed to find the optimal schedule for some cases in the previous experiments. Another difference between these two methods is that while *Push-up Scheduling* is a structural algorithm, *IMS* is a heuristic algorithm. During *IMS*, some of the previously scheduled instructions may be unscheduled due to the recurrence or resource conflict with the current scheduled instructions. The impact of this approach will be discussed later in this section.

Schedule Quality - The schedule quality of both methods in this study can be interpreted as the schedule length. The shorter the schedule length, the better. The optimal schedule length is equivalent to the *II* bounded by the resource constraint – *ResMII*. For *Push-up Scheduling*, the optimal schedule length will always be *ResMII* because finding the non-zero delay edges will eventually eliminate all the recurrence constraints. This explains why *Push-up Scheduling* successfully found the optimal schedules for the cases in Section 3. On the other hand, the optimal schedule obtained by *IMS* is bounded not only by the *ResMII* but also by the *RecMII*. The *RecMII* is calculated prior to the scheduling process in *IMS*. The algorithm used to calculate *RecMII* is ComputeMinDist, which is a variant of the Floyd-Warshall algorithm [3] for computing all shortest paths of a directed graph. By taking a closer look at the algorithm, it can be seen that the distances (cross iteration dependencies) between each pair of nodes have a great influence on the final value of *RecMII*. When the distance values are small, which means that the cross iteration dependencies cannot be neglected, the final value of *RecMII* is large, and is usually larger than *ResMII*. When the distance values are large, which means that the cross iteration dependencies may be ignored between the pair of nodes, the final value of *RecMII* is small and is usually equal to *ResMII*. In the experiments of Section 3, when using *IMS* for inner loop

problems, the distances were usually small because the outer loop distances were not counted, and the resulting schedule lengths were larger than the optimal solutions. On the other hand, during the experiments with nested loop problems, the outer loop dependencies were merged into inner loop dependencies by linearization, which resulted in large distance values, and consequently an optimal or near-optimal schedule length. The distances for a nested loop problem were directly calculated from a user input value – the number of inner loop iterations. From the results of the previous section, this user input value has little influence on the schedule length as long as this number is greater than the number of the nodes in the graph.

Complexity Issues - The core of *Push-up Scheduling* is the Chained Multidimensional Retiming algorithm [11]. The time required to retime one node in an MDFG is $O(|E|)$. Since there are N nodes in an MDFG, the total time complexity of the *Push-up Scheduling* algorithm is $O(N+|E|)$. The computational complexity of *IMS* is calculated from two stages. The first stage is the process of finding the *MII*. The computational complexity of finding *ResMII* is $O(N)$, while finding *RecMII* is $O(N^3)$ [12]. In practice, an algorithm for finding SCC (strongly connected components) is used instead. The complexity of the SCC algorithm is $O(N+|E|)$. So, the complexity of the first stage is $O(N+|E|)$. The next stage is *ModuloSchedule*, which in turn calls the *IterativeSchedule Final II - MII + 1* times. The *IterativeSchedule* is an $O(UN^2)$ process, where U is an user input value (*BudgetRatio*). Thus, the total computational complexity of *IMS* is $O((Final II - MII + 1) UN^2)$. When *Final II - MII + 1* is considered as a constant value, the complexity of *IMS* is simplified as $O(UN^2)$. Rau [12] claimed that U can also be dropped out of the complexity function. But a controversy is raised in this study. An observation from experiments is that, only when there exists a legal schedule at a given value of *II*, can *IterativeSchedule* find the solution in one pass. This phenomena is due to the characteristics of heuristic algorithms.

Implementation Issues - Since the target of *Push-up Scheduling* is the *MD* problem, its usage is limited to two or more dimensional problems. On the other hand, the *IMS* is designed for innermost loop problems, and its extension can also be used for *MD* problems. The implementation of *IMS* in a real compiler is relatively simple because the scheduling process is directly operated on by the instructions of the loop body, i.e. there is no need to find out the cross iteration dependencies. Extensive implementations and research have been done in this area. The implementation of *Push-up Scheduling* in real compilers is more complicated, and requires a high level of abstraction of the instructions into MDFGs.

5. Conclusion

In this paper, a comparative study of *Push-up Scheduling* and *Iterative Modulo Scheduling* for *MD* problems is presented. The experimental results showed that *Push-up Scheduling* can always achieve the optimal solution in linear time, while the *Iterative Modulo Scheduling* fails to obtain the optimal schedule in some cases. The computing complexity of *Push-up Scheduling* is $O(N+|E|)$ time, while the complexity of *Iterative Modulo Scheduling* is $O(UN^2)$. The implementation of *Push-up Scheduling* in a compiler is currently under research.

References

1. Aiken, A. and A. Nicolau, Fine-Grain parallelization and the wavefront method, *Lang. and Compilers for Parallel Computing*, Cambridge, MA, MIT Press, 1990, pp. 1-16.
2. Banerjee, U., Unicomodular transformations of double loops, *Advances in Languages and Compilers for Parallel Processing*, Cambridge, MIT Press, 1991, pp. 192-219.
3. Corman, T. H., C. E. Leiserson and R. L. Rivest, Introduction to algorithms, *MIT Press*, Cambridge, 1986.
4. Darte, A. and Y. Robert, Constructive methods for scheduling uniform loop nests, *IEEE Trans. on Parallel and Distributed Systems*, Vol. 5, no. 8, 1994, pp. 814-822.
5. Fettweis, A. and G. Nitsche, Numerical integration of partial differential equations using principles of multidimensional wave digital filter, *Journal of VLSI Signal Processing*, 3, 1992, pp. 7-24.
6. Goosens, G., J. Wandewalle, and H. De Man, Loop optimization in register transfer scheduling for DSP systems, *Proceedings of the ACM/IEEE Design Automation Conference*, 1989, pp. 826-831.
7. Hua, J., O. Rashid, N. L. Passos, R. Halverson and R. Simpson, Precise-MD: a software tool for resources constrained scheduling of multi-dimensional applications, to appear in *Proceedings of IEEE International Symposium on Circuits and System*, Monterrey, CA, May, 1998.
8. Liu, L.-S., C.-W. Ho and J.-P. Sheu, On the parallelism of nested for-loops using index shift method, *Proc. of the Int. Conf. On Parallel Processing*, 1990, Vol. II, pp. 119-123.
9. Passos, N. L. and E. H.-M. Sha, Push-up scheduling: optimal polynomial-time resource constrained scheduling for multi-dimensional applications, *Proc. of the In. Conf. On Computer Aided Design*, November, 1995, pp. 588-591.
10. Passos, N. L., E. H.-M. Sha, and S. C. Bass, Loop pipelining for scheduling multi-dimensional systems via rotation, *Proceedings of the 31st Design Automation Conference*, San Diego, CA, 1994, pp. 485-490.
11. Passos, N. L. and E. H.-M. Sha, Full parallelism in uniform nested loops using multi-dimensional retiming, *Proceedings of the International Conference On Parallel Processing*, Saint Charles, IL, 1994, Vol. II, pp. 130-133.
12. Rau, B. R., Iterative modulo scheduling, *HPL Technical Report*, Hewlett-Packard Laboratory, 1994.
13. Rau, B. R. and D. D. Glaeser, Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing, *Proc. 14th Ann. Workshop Microprogramming*, 1981.