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) + t(v)} where v is a member of the set of nodes preceding u by an edge e, and d(e) = (0, 0). In order to simplify the examples presented in this paper, we assume t(v) = 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 → 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). This retiming allows the schedule of v at time AVAIL(fu). 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 $u$. \textit{Push-up Scheduling} generates a retiming vector $r(u)$ for each node in the DFG such that the retimed delay of each edge is given by $d_i(e_i) = d(e_i) + r(u)$, where $e_i$ represents all the outgoing edges from $u$, and $d_i(e_i) = d(e_i) - r(u)$, 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 DFG.

\begin{figure}[h]
\centering
\begin{tikzpicture}
  \node (A) at (0,0) {A};
  \node (B) at (1,-1) {B};
  \node (C) at (1,1) {C};
  \node (D) at (0,1) {D};

  \draw (A) -- (B) node [midway, below] {B};
  \draw (A) -- (C) node [midway, above] {A};
  \draw (B) -- (D) node [midway, below] {B};
  \draw (C) -- (D) node [midway, above] {C};

  \draw (A) -- (0,0) node [midway, below] {(0,0)};
  \draw (B) -- (1,-1) node [midway, below] {(1,-1)};
  \draw (C) -- (1,1) node [midway, above] {(1,1)};
  \draw (D) -- (0,1) node [midway, above] {(0,1)};

\end{tikzpicture}
\caption{(a) A retimed DFG. (b) Its corresponding loop body.}
\end{figure}

\begin{figure}[h]
\centering
\begin{tikzpicture}
  \node (A) at (0,0) {A};
  \node (B) at (1,-1) {B};
  \node (C) at (1,1) {C};
  \node (D) at (0,1) {D};

  \draw (A) -- (B) node [midway, below] {B};
  \draw (A) -- (C) node [midway, above] {A};
  \draw (B) -- (D) node [midway, below] {B};
  \draw (C) -- (D) node [midway, above] {C};

  \draw (A) -- (0,0) node [midway, below] {(0,0)};
  \draw (B) -- (1,-1) node [midway, below] {(1,-1)};
  \draw (C) -- (1,1) node [midway, above] {(1,1)};
  \draw (D) -- (0,1) node [midway, above] {(0,1)};

\end{tikzpicture}
\caption{(a) A valid DFG. (b) Corresponding loop body.}
\end{figure}

The essence of \textit{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 DFG for the example in Figure 1 is presented in Figure 2. \textit{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 \textit{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 $c+i*\text{II}$. The goal of IMS is to find the smallest $\text{II}$ while satisfying the resource and recurrence constraints.

In order to use IMS, two pseudo operations must be added to the dependence graph: \textit{START} and \textit{STOP}. \textit{START} is assumed to be the predecessor of all the other operations, and \textit{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 $c_p$ clock cycles and there are $n_r$ copies of this resource, the ResMII is

$$R_{\text{es}} = \max_{p \in P} \left( \max_{r \in R} \left[ \frac{c_p}{n_r} \right] \right)$$

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 edge $e$ in a cycle of the dependence graph has latency $l_e$ and the operations connected to it are $d_e$ iterations apart, then RecMII is

$$RecMII = \max_{c \in C} \left( \sum_{e \in E_c} \frac{l_e}{d_e} \right)$$

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 RecMII.

For a given $\text{II}$, 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$[i,j]$ 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 $\text{II}$ 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 to their priorities and the scheduling priorities are assigned in such a way that the most constrained have the highest priority. Every operation is 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

$$r_i = 1 - \frac{E_{S_i}}{II}$$

where $E_{S_i}$ is the earliest start time of operation $i$. The relationship between the retiming vector and the schedule time is illustrated in Figure 3.

$$d(i) = d_x \times N + d_y$$

where $d_x$ is the original delay in the $x$ direction and $d_y$ is the original delay in $y$ direction.

![Figure 3](image)

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 $d(i) = dx \times N + dy$, 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](image)

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.
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 (II=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.

### Table 1. IMS Scheduling Results

<table>
<thead>
<tr>
<th>II</th>
<th>Adder</th>
<th>Multiplier</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>C</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>D</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>A</td>
</tr>
</tbody>
</table>

II = 3

II = 4

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

### Figure 6.
(a) Retimed graphs for problem a when solved by IMS. (b) Retimed graphs for problem b when solved by IMS.

### Table 2. Schedule table for problems solved by Push-up Scheduling

<table>
<thead>
<tr>
<th>II</th>
<th>Adder</th>
<th>Multiplier</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>C</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>B</td>
</tr>
</tbody>
</table>

(a) problem a (b) problem b

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

<p>| II = 2 |</p>
<table>
<thead>
<tr>
<th>CS</th>
<th>Adder</th>
<th>Multiplier</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>D</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td></td>
</tr>
</tbody>
</table>

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

![Diagram of MDFG for nested loop problem](image)

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

<table>
<thead>
<tr>
<th>CS</th>
<th>Adder</th>
<th>Adder</th>
<th>Multiplier</th>
<th>Multiplier</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>D</td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

II = 1

(a) original schedule

(b) solved by IMS and PUS

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.
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 retie 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 $O(N^2)$ [12]. In practice, an algorithm for finding SCC (strongly connected components) is used instead. 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 $O(Final II - MII +1 \times U^2)$. 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