??, ??, 1–12 (??)
© ?? Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Reducing Data Hazards on Multi-pipelined DSP Architecture with Loop Scheduling

SISSADES TONGSIMA, CHANTANA CHANTRAPORNCHAI, EDWIN H.-M. SHA

{stongsim,cchantra,esha}@cse.nd.edu

Dept. of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN 46556

NELSON L. PASSOS

fpassosn@nexus.mwsu.edu

Dept. of Computer Science, Midwestern State University, Wichita Falls, TX 76308

Received ??; Revised ??

Editors: ??

Abstract. Computation intensive DSP applications usually require parallel/pipelined processors in order to meet specific timing requirements. Data hazards are a major obstacle against the high performance of pipelined systems. This paper presents a novel efficient loop scheduling algorithm that reduces data hazards for such DSP applications. This algorithm has been embedded in a tool, called SHARP, which schedules a pipelined data flow graph to multiple pipelined units while hiding the underlying data hazards and minimizing the execution time. This paper reports significant improvement for some well-known benchmarks showing the efficiency of the scheduling algorithm and the flexibility of the simulation tool.

1. Introduction

In order to speedup current high performance DSP systems, *multiple pipelining* is an important strategy that should be explored. Nonetheless, it is well-known that one of the major problems in applying the pipelining technique is the delay caused by dependencies between instructions, called *hazards*. Control hazards are known as the hazards that prevent the next instruction in the instruction stream from being executed, such as branch operations. Likewise, the hazards that encumber the next instruction by data dependencies are called data hazards. Most computation-intensive scientific applications, such as image processing, and digital signal processing, contain a great number of *data hazards* and few or no *control hazards*. In this paper, we present a tool, called *SHARP* (Scheduling with Hazard Reduction for multiple Pipeline architecture), which was developed to obtain a short schedule while minimizing the underlying data hazards by exploring loop pipelining and different multiple pipeline architectures.

Many computer vendors utilize a *forwarding* technique to reduce the number of data hazards in their architectures. This process is implemented in hardware whereby a copy of the computed result is sent back to the input prefetch buffer of the processor. However, the larger the number of forwarding buffers, the higher the cost that will be imposed on the hardware. Therefore, there exists a trade-off between its implementation cost and the performance gain. Furthermore, many modern high speed computers, such as MIPS R8000, IBM Power2 RS/6000 and oth-

ers, use multiple pipelined functional units (multi-pipelined) or superscalar (super)pipelined architectures. Providing a tool that determines an appropriate pipelined architecture for a given specific application, therefore, will be beneficial to computer architects. By using such a tool, one can find a suitable pipeline architecture that balances the hardware and performance costs by varying the system architecture (e.g., a number of pipeline units, type of each unit, forwarding buffers, etc.).

Rearranging the execution sequence of tasks that belong to the computational application can reduce data hazards and improve the performance. Dynamic scheduling algorithms such as *tomasulo* and *scoreboard* are examples of implementing the algorithms in hardware. They were introduced to minimize the underlying data hazards which can not be resolved by a compiler [16]. These techniques, however, increase the hardware complexity and costs. Therefore, special consideration should be given to static scheduling, especially for some computation-intensive applications. The fundamental performance measurement of a *static* scheduling algorithm is the total completion time in each iteration, also known as the schedule length. A good algorithm must be able to maximize parallelism between tasks and minimize the total completion time. Many heuristics have been proposed to deal with this problem, such as *ASAP* scheduling, *ALAP* scheduling, *critical path* scheduling and *list* scheduling algorithms [2, 3]. The critical path, list scheduling and graph decomposition heuristics have been developed for scheduling acyclic *data flow graphs (DFGs)* [7, 14]. These methods, however, do not consider the parallelism and pipelining across iterations. Some studies propose scheduling algorithms to deal with cyclic graphs [5, 15]. Nevertheless, these techniques do not address the issue of scheduling on pipelined machines that exploit the use of forwarding techniques.

Considerable research has been done in the area of loop scheduling based on *software pipelining*—a fine-grain loop scheduling optimization method [6, 10, 12]. This approach applies the *unrolling* technique which expands the target code segment. The problem size, however, also increases proportionally to the unrolling factor. *Iterative modulo scheduling* is another framework that has been implemented in some compilers [13]. Nonetheless, in order to find an optimized schedule, this approach begins with an infeasible initial schedule and has to reschedule *every* node in the graph at each iteration.

The target DSP applications usually contain iterative or recursive code segments. Such segments are represented in our new model, called a *pipeline data-flow graph* (PDG). An example of a PDG is shown in Figure 1(b). In this model, nodes represent tasks that will be issued to a certain type of pipeline and edges represent data dependencies between two nodes. A weight on each edge refers to a minimum hazard cost or pipeline cost. This cost represents a required number of clock cycles that must occur in order to schedule successive nodes. In this work, a proposed novel pipeline scheduling algorithm, SHARP, takes a PDG and some pipeline architecture specifications (e.g., pipeline depth, number of forwarding buffers, type and number of pipeline units etc.) as inputs. The algorithm then efficiently schedules nodes from the PDG to the target system.

After the initial schedule is computed, by a DAG scheduling algorithm such as list scheduling, the algorithm implicitly uses retiming. Only a *small* number of nodes are rescheduled in each iteration of our algorithm. The new scheduling position is obtained by considering data dependencies and loop carried dependencies, i.e., using loop pipelining strategy as a basis to reduce data hazards while improving the total execution time under the hardware constraints given by the user specifications.

As an example, Figure 1(a) presents two pipeline architectures each of which consists of five stages: instruction fetch (IF), instruction decode (ID), execution (EX), memory access (M) and write-back (WR). For simplicity, assume that each of these stages takes one *clock cycle* to finish [4]. The pipeline hazard in this case is 3, since with this architecture, any instruction will put data available to read in the first half of the 5th stage (WR) and read it in the 2nd stage (ID). In Section 2, we will explain how to calculate this cost in more detail. The PDG and its corresponding code segment to be executed in this two-pipeline system are shown in Figures 1(b) and (c) respectively. Notice that each node of the graph also indicates the type of instruction required to be ex-

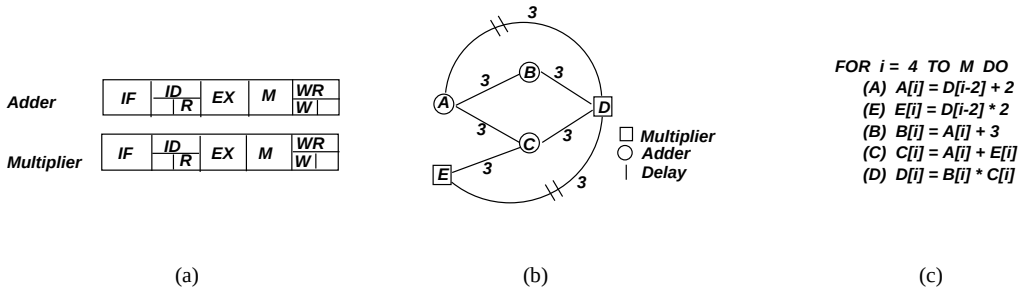


Fig. 1. (a) Target pipeline architecture (b) Pipeline DFG (c) Corresponding code segment

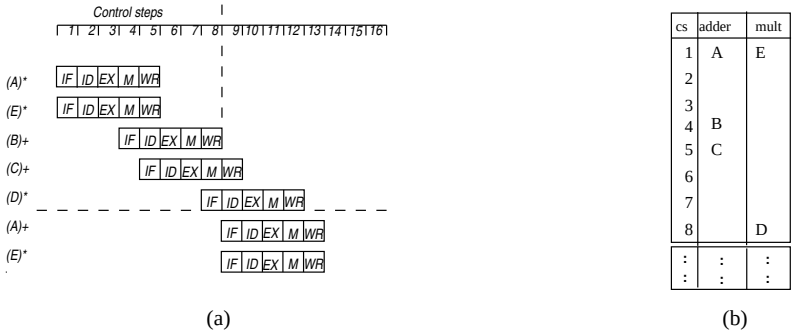


Fig. 2. (a) Pipeline execution pattern (b) An equivalent schedule

ecuted. Assume that the WR and ID stages of two dependent instructions can be overlapped. For example, instruction B can start reading (at the ID stage) data produced by instruction A at the WR stage. A legal execution pattern of the pipeline for this example is illustrated in Figure 2(a).

Since all the pipeline stages of issued instructions are consecutive, only the beginning of each instruction pipeline is required to be shown. Figure 2(b) illustrates a schedule table resulting from Figure 2(b). This table only shows one iteration of the sample code segment (the complete table comprises of $M-3$ identical copies of this table). Such a schedule becomes an *initial schedule* which can be optimized by SHARP. Figure 3(a) and 3(b) show the resulting intermediate PDG and schedule after applying SHARP to the initial schedule. Nodes A and E from the next iteration are rescheduled to current iteration of the schedule. This is equivalent to retiming these nodes in the PDG (see Figure 3(a)). This technique explores the parallelism across iterations (loop pipelining). SHARP repeatedly applies such a method to each intermediate schedule. Figures 3(c) and 3(d) show the third intermediate retimed PDG and its schedule respectively. At the third iteration we obtain the optimized schedule with length 6 (a 25% improvement over the initial schedule).

Using our tool, we obtain not only the reduced schedule length but we can also evaluate other architecture options, such as introducing forwarding hardware in the architecture or even additional pipelines. In order to present our algorithm, the remainder of this work is organized as follows: Section 2 introduces some fundamental concepts. The main idea and theorems behind the algorithm used in SHARP are presented in Section 3. In Section 4, we discuss the experimental results obtained by applying different pipeline architectures to this tool. Finally, Section 5 draws conclusions of this work.

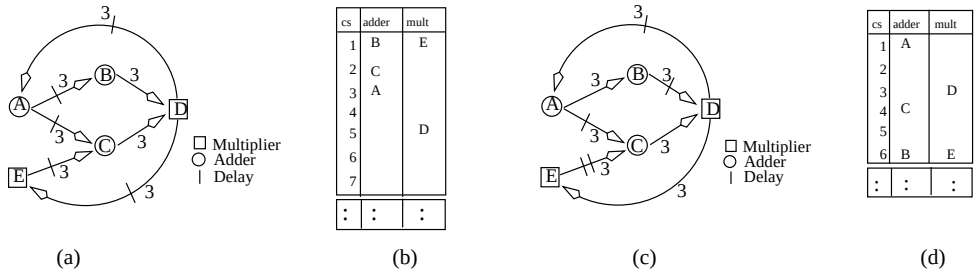


Fig. 3. (a)-(b) PDG and schedule of intermediate step of the algorithm, (c)-(d) PDG and the schedule after third step

2. Background

A *cyclic* DFG, $G = \langle V, E \rangle$, is commonly used to represent dependencies between instructions in an iterative or a recursive loop. However, it does not reflect the type of pipeline architecture to which the code segment is subjected. Therefore, in order to distinguish them, some common hardware factors need to be considered.

The type of architecture may be characterized by considering different types of pipelines in the system. The number of stages in a pipeline, or *pipeline depth*, is one configuration factor that is necessary to be taken into account, since it affects the overall pipeline speedup. The forwarding hardware is also a factor because it can diminish the data hazards. Furthermore, the system may consist of a number of forwarding buffers, responsible for how many times a pipeline is able to bypass a datum [9].

In this paper, we assume our algorithm guarantees that no delays occur during the execution of one instruction. In other words, the number of cycles from the execution of the first to the last pipeline stage for one instruction is equal to the pipeline depth. In a multi-pipelined machine, if the execution of an instruction I_2 depends on the data generated by instruction I_1, and the starting moment of I_1 and I_2 are t_1 and t_2 respectively, we know that $t_2 - t_1 \geq S_{\text{out}} - S_{\text{in}} + 1$, where S_{out} is the index of the pipeline stage from which the data is visible to the instruction I_2, and S_{in} is the index of pipeline stage that needs the result of I_1 in order to execute I_2. We call $S_{\text{out}} - S_{\text{in}} + 1$ the *pipeline cost* of the edge connecting the two nodes, representing instructions I_1 and I_2. Figure 4 illustrates the concept of the pipeline cost. Such a cost can be qualified in three possible situations depending on the characteristics of the architecture:

case 1: The pipeline architecture does not have a forwarding option. The pipeline cost is similar to the data hazard, which may be calculated from the difference between the pipeline depth and the number of the overlapped pipeline stages. For example in Figure 4(a), this pipeline reads data at the end of ID and the data is ready after WR. The S_{out} stage is 7 and S_{in} is 3. Hence, for this case, the pipeline cost is $7 - 3 + 1 = 5$.

case 2: The pipeline architecture has an *internal* forwarding, i.e., data can merely be bypassed inside the same functional unit. The pipeline cost from this case may be obtained in a similar way as above. For instance, the pipeline in Figure 4(b) has the internal forwarding such that the data will be available right after the EX stage. Then, S_{out} is stage 5 and S_{in} is stage 3, so the pipeline cost is $5 - 3 + 1 = 3$. In this case, the special forwarding hardware is characterized into two sub-cases.

1. The forwarding hardware has a limited number of feedback buffers. The pipeline cost will be the value without forwarding when all the forwarding buffers are utilized.
2. The forwarding hardware has an unlimited number of feedback buffers (bounded by the pipeline depth). In this case, the pipeline cost will always be the same.

case 3: The pipeline architecture has an *external* or *cross* forwarding hardware, such as it is capable of passing data from one pipeline to

Fig. 4. (a) Pipeline cost w/o forwarding hardware (b) Pipeline cost w/ forwarding capability

another pipeline. We calculate the pipeline cost by the same way described above. Again, limited number of buffers or unlimited number of buffers are possible sub-cases.

2.1. Graph Model

In order to model the configuration of each multi-pipelined machine associated with the problem being scheduled, the pipeline data-flow graph is introduced.

Definition 1. A pipeline data-flow graph (PDG) $G = \langle V, E, T, \mathbf{d}, \mathbf{c} \rangle$ is an edge-weighted directed graph, where V is the set of nodes, $E \in V \times V$ is the set of dependence edges, T is the pipeline type associated with a node $u \in V$, $\mathbf{d}$ is the number of delays between two nodes, and $\mathbf{c}(e) = (c_{fo}, c_{no})$ is a function from E to the positive integers representing the pipeline cost, associated with edge $e \in E$, where c_{fo} and c_{no} are the cost when considering with and without forwarding capability respectively.

Each node in a PDG represents an instruction, and the type of pipeline in which the instruction will be executed. An edge from node u to node v, exhibited by the notation $u \rightarrow v$, conveys that the instruction v depends on the result from the instruction u. The number of delays $\mathbf{d}(e)$ on any edge $e \in E$ such that u precedes v, where $u, v \in V$, indicates a data dependence from node u to v, such that the execution of node v at iteration j relies on the data produced by node u at iteration $j - \mathbf{d}(e)$. The tuple associated with each edge in a PDG, $u \stackrel{(c_{fo}, c_{no})}{\longrightarrow} v$, is architecture-dependent where c_{fo} is the number of clock cycles required when there exists a forwarding hardware, and c_{no} is the number of clock cycles needed when executing the two instructions u and v considering no forwarding. If there is no forwarding hardware, the value of c_{fo} will be the same as c_{no}.

As an example, Figure 5(a) illustrates a simple PDG associated with two types of functional units, adder and multiplier. Each of which is a five-stage pipeline architecture with a forwarding function. Therefore, the pipeline cost

$c_{fo} = 1$ and $c_{no} = 3$. Nodes A, B, C, D, and F represent the addition instructions and node E indicates the multiplication instruction. The bar lines on $D \rightarrow A$ and $F \rightarrow E$ represent the number of delays between the nodes, i.e., two delays on $D \rightarrow A$ conveys that the execution of operation D at some iteration j produces the data required by A at iteration $j + 2$.

2.2. Initial Scheduling in SHARP

In this subsection, we introduce some important considerations in acquiring a static pipeline schedule from the PDG. Considerable research has been conducted in seeking a scheduling solution for a DFG [8]. In this research, we tailor the list scheduling heuristic so that it agrees with conditions of the PDG.

A static schedule consists of multiple entries in a table. Each row entry indicates one clock cycle—the synchronization time interval, also called *control step*. Each column entry represents one of the pipeline units in the multi-pipelined system where nodes that have the same corresponding types of pipeline will be assigned. The first pipeline stage of each scheduled node starts executing whenever the node appears in the table.

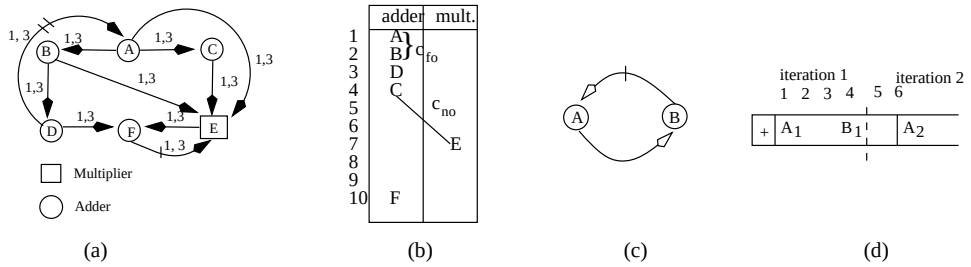


Fig. 5. (a) Example when $c_{fo} = 1$ and $c_{no} = 3$ (b) Corresponding initial schedule (c)-(d) PDG with inter-iteration dependency between node A and B

In order to obtain a static schedule from a PDG feedback edges (i.e., edges that contain delays) are temporarily ignored in this initial scheduling phase. For instance, $D \rightarrow A$ and $F \rightarrow E$ in Figure 5(a) are temporarily ignored. Our scheduling guarantees the resulting initial schedule is legal by satisfying the following properties. Further, the following scheduling properties must be preserved by any scheduling algorithm.

Property 1. *Scheduling properties for intra-iteration dependencies*

1. *For any node n preceded by nodes m_i by edges e_i such that $\mathbf{d}(e_i) = 0$ and $m_i \overset{(c_{fo_i}, c_{no_i})}{\longrightarrow} n$. If $\mathsf{cs}(m_i)$ is the control step to which m_i was scheduled and b_i is the number of available buffers for the functional unit required by m_i, then node n can be scheduled at any control step (cs) that satisfies the following rules.*

$$cs \geq \max_i \{\mathsf{cs}(m_i) + cost(b_i)\}$$

$$where \begin{cases} cost(b_i) = c_{fo} & if\ b_i > 0 \\ cost(b_i) = c_{no} & otherwise \end{cases}$$

2. *If there is no direct-dependent edge between nodes q and r, i.e. $\mathbf{d}(e) \neq 0$, and q is scheduled to control step k, node r may be placed at any unoccupied control step which does not conflict with any other data dependency constraints.*

Note that if the architecture does not use forwarding hardware, then we set $c_{fo} = c_{no}$ and $b_i = 0$. As an example, Figure 5(b) presents the resulting schedule table when we schedule the graph shown in Figure 5(a) to a multiple pipelined system consisting of one adder and one multiplier with one internal buffer for each unit.

The last step of initial scheduling is to check the inter-iteration dependency which is implicitly represented by the feedback edges of the PDG. A certain amount of empty control steps has to be preserved at the end of the schedule table if the number of control steps between two inter-dependent nodes belonging to different iterations is not sufficient to satisfy the required pipeline cost of the two corresponding instructions. Figures 5(c) and (d) illustrates this situation. Assume that the pipeline depth of the adder is 5. If we did not consider the feedback edge, the schedule length would be only 4. However, the schedule length actually has to be 6 since node A in the next iteration, represented by A_2, cannot be assigned to the control step right after node B_1 due to the inter-iteration dependency between nodes A and B. Hence two empty control steps need to be inserted at the end of this initial schedule and the final schedule length becomes six rather than four.

Note again that the execution of all pipeline stages, except for the first one, of any scheduled node are hidden in an initial schedule table. Those stages are overlapped and only the first stage of each node is displayed, e.g., see Figure 2(a). After applying a list scheduling algorithm that enforces Property 1 to the example in Figure 1, the initial schedule of Figure 2(b) is produced. The static schedule length for that case is 8.

3. Reducing Schedule Length

In the previous section, we discussed the scheduling conditions for assigning nodes from a PDG to a schedule table. These conditions are also applied to the optimization process of our algorithm. Our algorithm is able to reduce the underlying static schedule length of an initial schedule previously obtained. It explores the parallelism across iterations by implicitly employing the retiming technique [11]. The following section briefly reviews the retiming and rotation techniques.

3.1. Retiming and Rotation

The *retiming* technique is a commonly used tool for optimizing synchronous systems. A retiming $\mathbf{r}$ is a function from V to $\mathcal{Z}$. The value of this function, when applied to a node v, is the number of delays taken from all incoming edges of v and moved to its outgoing edges. An illegal retiming function occurs when one of the retimed edge delays becomes negative. This situation implies a reference to a non-available data from a future iteration. Therefore, if we consider $G_{\mathbf{r}} = \langle V, E, T, \mathbf{d_r}, \mathbf{c} \rangle$ to be a PDG G retimed by a function $\mathbf{r}$, a retiming is legal if the retimed delay count $\mathbf{d_r}$ is nonnegative for every edge in E. For an edge $u \rightarrow v$, the number of additional delays is equal to the number of delays moved to the edge through node u, subtracted by the number of delays drawn out from the edge through node v. The retiming technique can be summarized by the following properties:

Property 2. *Let $G_{\mathbf{r}} = \langle V, E, T, \mathbf{d_r}, \mathbf{c} \rangle$ be a PDG $G = \langle V, E, T, \mathbf{d}, \mathbf{c} \rangle$ retimed by $\mathbf{r}$.*

1. *$\mathbf{r}$ is a legal retiming if $\mathbf{d_r}(e) \geq 0$ for every $e \in E$.*
2. *For any edge $u \xrightarrow{e} v$, we have $\mathbf{d_r}(e) = \mathbf{d}(e) + \mathbf{r}(u) - \mathbf{r}(v)$.*
3. *For any path $u \overset{p}{\rightsquigarrow} v$, we have $\mathbf{d_r}(p) = \mathbf{d}(p) + \mathbf{r}(u) - \mathbf{r}(v)$.*
4. *For a loop l, we have $\mathbf{d_r}(l) = \mathbf{d}(l)$.*

Property 2 demonstrates how the retiming method operates on a PDG. An example of retiming is shown in Figure 6. The retiming $\mathbf{r}(A) = 1$ conveys that one delay is drawn from the incoming edge of node A and pushed to all of its outgoing edges, $A \rightarrow B$ and $A \rightarrow C$.

After a graph has been retimed, a *prologue* is the set of instructions that must be executed to provide the necessary data for the iterative process. In our example, the instruction A becomes the prologue. An *epilogue* is the other extreme, where a complementary set of instructions will need to be executed to complete the process. The time required to run the prologue and epilogue is assumed to be negligible when compared to the total computation time of the problem.

Chao, LaPaugh and Sha proposed a flexible algorithm, called rotation scheduling, to deal with scheduling a DFG under resource constraints [1]. Like its name, this algorithm analogously moves nodes from the top of a schedule table to its bottom. The algorithm essentially shifts the iteration boundary of the static schedule down, so that nodes from the next iteration can be explored. We now introduce some necessary terminology and concepts used in this paper.

Definition 2. Given a PDG $G = \langle V, E, T, \mathbf{d}, \mathbf{c} \rangle$ and $R \subset V$, the rotation of R moves one delay from every incoming edge to all outgoing edges of nodes in R. The PDG now is transformed into a new graph (G_R).

For a schedule table with length L, this definition is applicable when moving the first row of the schedule table to the position $L + 1$. Therefore, this operation implicitly retimes the graph. The benefit of doing the rotation is that a few number of nodes are rescheduled. Therefore only a small part of an input graph is modified instead of rescheduling the whole graph. Rotation scheduling must preserve the following property:

Property 3. *Let $G = \langle V, E, T, \mathbf{d}, \mathbf{c} \rangle$ be a PDG and $R \subset V$. A set R can be legally retimed if and only if every edge from $V - R$ to R contains at least one delay.*

This property implies that the rotation operation always preserves Property 2. After performing the rotation strategy, the dependencies in a new graph are changed, since some delays in the

Fig. 6. An example of retiming data-flow graph (a) Original (b) When $\mathbf{r}(A) = 1$

graph are now moved to new edges. This allows us to explore the possibility of parallelizing those nodes that do not have a direct-dependent edge from their predecessors. Carefully re-scheduling those rotated nodes to new positions, the schedule length can be decreased.

Nevertheless, as mentioned earlier, we also have to consider the inter-iteration dependency. Hence a new schedule position assignment for a node has to be carefully chosen to avoid conflicts in the dependency constraint between iterations. Finding a valid scheduling position now becomes more complex since the problem now has incorporated pipeline hazards. The major different from the traditional algorithm is that our algorithm requires checking not only dependencies of the new graph after rotating a node but also pipeline hazards which may occur only if schedule a node to different processors. The following section discusses how to find such a dependency and avoiding the underlying pipeline hazards in detail.

3.2. Minimum Schedule Length

We know that a number of delays on any edge $u \rightarrow v$, where $u, v \in V$, indicates in which iteration, prior to the current iteration, node u legally produces data for node v. This conveys that in order to schedule the rotated nodes, the pipeline cost constraints must also be satisfied, e.g., the inter-iteration dependency between nodes u and v.

Definition 3. Given a PDG $G = \langle V, E, T, \mathbf{d}, \mathbf{c} \rangle$ and nodes $u, v \in V$ where $u \rightarrow v \in E$, the minimum schedule length with respect to nodes u and v, $ML(u, v)$, is the minimum schedule length required to comply with all data-dependent constraints.

The following theorem presents the ML function.

Theorem 1. *Given a PDG $G = \langle V, E, T, \mathbf{d}, \mathbf{c} \rangle$, an edge $e = u \rightarrow v \in E$, and $\mathbf{d}(e) = k$ for $k > 0$, a legal schedule length for G must be greater than or equal to $ML(u, v)$, where*

$$ML(u, v) = \left\lceil \frac{pipe\_cost + cs(u) - cs(v)}{k} \right\rceil$$

with $cs(node)$ being the starting control step of that node and $pipe\_cost$ is either c_{no} or c_{fo} depending on the architecture.

Proof: Let L be the schedule length for one iteration. We know that the minimum number of control steps between node u at iteration j and node v at iteration $j + k$ is the pipeline cost associated with $u \rightarrow v$. There are $k - 1$ iterations between iterations j and $j+k$. Since all iterations have the same length L, the following equation is the relationship of the distance between $cs(u)$ and $cs(v)$: $L \times (k - 1) + (L - cs(u)) + cs(v) + \Delta \geq pipe\_cost$ where Δ represents a number of compensated control steps fulfilling the pipeline cost requirement. Hence, Δ can be expressed as: $\Delta \geq pipe\_cost - L \times (k - 1) - L + cs(u) - cs(v)$. In order to obtain a uniform schedule, Δ is distributed over all k iterations preceding iteration $i+k$. This distribution results in a minimum value $\delta = \lceil \frac{\Delta}{k} \rceil$, and the new static schedule length that satisfies the constraints with respect to u is $ML = \delta + L$. After substituting, we obtain

$$ML(u, v) = \left\lceil \frac{pipe\_cost + cs(u) - cs(v)}{k} \right\rceil$$

```
1.  Algorithm: SHARP
2.  Input: G = ⟨V, E, T, d, c⟩, # forwarding buffers, and # pipelines
3.  Output:  shortest schedule table S
4.      S := Initial-Schedule(G), Q := S;
5.      for i := 1 to |V|
6.          (G, S) :=Pipe_rotate(G);
7.          if length(S) < length(Q)
8.              then Q := S;
```

Fig. 7. SHARP framework

```
1.  Procedure: Pipe_rotate
2.  Input: PDG, input schedule
3.  Output: Resulting schedule
4.      N := Deallocate(S);                          /* extract nodes from the table */
5.      G_r := Retime(G, N);                         /* retime nodes in N */
6.      S := Re-schedule(G, S, N);
7.      return(G_r, S);
```

Fig. 8. Pipeline rotation scheduling routine

```
1.  Procedure: Re-schedule
2.  Input: PDG (G), set or rotated nodes (N), input schedule (S)
3.  Output: Resulting schedule
4.      foreach v ∈ N do
5.          cs_min := max{parent(v).cs + cost(parent(v).b_i)};
6.          cs_max := length(S);                     /* get schedule length */
7.          cs := cs_min;
8.          while (cs < cs_max) do
9.              increment cs and processor number pid until finding
10.             the first "legal" processor available between cs and cs_max
11.         if cs ≥ cs_max then
12.             do nothing
13.         else
14.             schedule node v to the resulting cs and pid obtained above
```

Fig. 9. Routine for re-mapping nodes to a schedule

□ *3.3. Algorithm*

Since a node may have more than one predecessor, in order to have a legal schedule length, one must consider the maximum value of *ML*. In other words, the longest schedule length that is produced by computing this function will be the worst case that can satisfy all predecessors.

The scheduling algorithm used in SHARP applies the *ML* function to check if a node can legally be scheduled at a specific position. Therefore, it may happen that the obtained schedule will require some empty slots to be added to compensate for the inter-iteration dependency situation. We summarize this algorithm in Figures 7–9 where Figures 8 and 9 show how we implement the two important optimization functions *Pipe_rotate* and *Re-schedule* in SHARP. Note again that the initial schedule in the algorithm can be obtained by

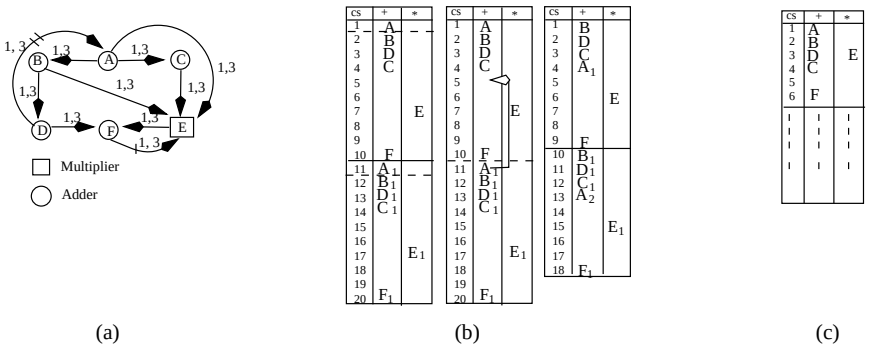


Fig. 10. (a) The 6-node PDG (b) the first optimized schedule table and (c) the final schedule

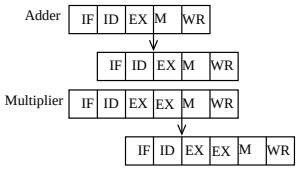
any DAG scheduling algorithm, e.g., a modified list scheduling that satisfies Property 1. Next, the procedure *Pipe_rotate* is applied to shorten the initial schedule table. It first deallocates nodes from the schedule table. These nodes are then retimed and consequently rescheduled. The procedure *Re-schedule* finds a proper position such that the schedule length will not exceed the previous length. A scheduling position has to satisfy Property 1 and Theorem 1. This process is computed in the while loop (Lines 8–10) which calculates an appropriate control step considering a pipeline cost and buffer size as well as *ML*. Then, if the obtained control step is smaller than the current one and a required unit is available, the node can be re-scheduled. Otherwise, it remains at the same position. As a result, the new schedule table can either be shorter or have the same length.

Consider now the PDG shown in Figure 10. In this example, there are 5 addition-instructions and 1 multiplication-instruction. Assume that the target architecture is similar to the one presented in the introduction section (i.e., one adder and one multiplier with one-buffer internal forwarding). After obtaining the initial schedule, shown in Figure 10(b), the algorithm attempts to reduce the schedule length by calling the function *Pipe_rotate* which brings A from the next iteration, called A_1, and re-schedule it to $cs5$ (which is $cs4$ after re-numbering the table) of the addition unit. By doing so, the forward buffer of A, which was granted to B in the initial schedule, is free since this new A_1 does not produce any data for B. Then, the static schedule length becomes 9 control steps. After running SHARP for 4 iterations, the schedule length is reduced to six control steps as illustrated in Figure 10(c).

4. Experimental Results

We have used SHARP in experiment on several benchmarks with different hardware assumptions: no forwarding, one buffer-internal forwarding, sufficient buffer-internal forwarding (in-frw.), one buffer-external forwarding, two buffer-external forwarding and sufficient buffer-external forwarding (ex-frw.). The target architecture is comprised of a 5-stage adder and a 6-stage multiplier pipeline units. When the forwarding feature exists, the data produced at the end of the EX-stage can be forwarded to the next execution cycle of EX-stage as shown in Figure 11(a).

Note that the *sufficient*-xx forwarding assumption conveys that its architecture has sufficient number of forwarding buffers. Furthermore, the internal and external modifiers for each assumption convey that the forwarding technique can be done within one functional unit and between two functional units respectively. The set of benchmark problems and their characteristics are shown in Figure 11(b) Tables 1 and 2 exhibits the simulation results from a system that contains one adder/one multiplier and 2 adders/2 multipliers respectively. Note that the results presented in these tables were collected after running SHARP against each benchmark until there is no improvements for 7 consecutive intermediate schedules (i.e., seven iterations of the algorithm). Both



(a)

Benchmark	Mul	Add	Sum
1. 5-th Order Elliptic Filter	8	26	34
2. Differential Equation Solver	6	5	11
3. All-pole Lattice Filter	11	4	15
4. WDF for Transmission Line Computation	4	8	12
5. Unfolded All-pole Lattice Filter (uf=7)	66	24	90
6. Unfolded Differential Equa. Solver (uf=5)	24	20	44
7. Unfolded 5-th Elliptic Filter (uf=2)	16	52	68

(b)

Fig. 11. (a) 5-stage adder with internal forwarding unit and 6-stage multiplier with internal forwarding unit (b) Characteristics of the benchmarks

tables present an initial schedule length of each benchmark and the final length after applying the algorithm to the initial schedule (see column int/aft). The reduction percentage of each benchmark is presented in column %.

From experiments, the performance of the one buffer-internal forwarding scheme is very close to the sufficient buffer-internal forwarding one. This is because most of the selected benchmarks have only one or two outgoing edge(s) (fan-out degree) for each node. Increasing the number of internal forwarding buffers may slightly increase performance. The performance of a system with one buffer could be worse than the one with sufficient buffers for some applications with large fan-out degrees. In this case, only one successive node can be scheduled earlier by consuming the data from an only buffer and the rest of the successive nodes would cause the underlying data-dependent hazards, i.e., waiting for data being ready from its parent at WR-stage. For a system with external forwarding, data can be forwarded to any functional unit in the system. Therefore, the resulting schedule length is shorter than that of the system with internal forwarding capability.

Selecting an appropriate number of buffers depends on the maximum fan-out degree and the pipeline depth. In some cases only one or two buffers are enough with additional buffers not producing a significant improvement. As an example, consider column 4 of Table 1 which describes a system with external forwarding. Particularly for the wave digital filter application (benchmark 4) using only one buffer is the most appropriate since the algorithm results in the maximum reduction, 33%, over the initial schedule length. Adding 2 or more buffers results in an 11% reduction. For the differential equation solver application (benchmark 2), selecting two buffers is a good choice since the algorithm yields the maximum reduction.

The number of available units is also another significant criterion. Since most of the tested applications require more than one addition and one multiplication, increasing the number of functional units can reduce the underlying completion time. Doubling the number of adders and multipliers makes the initial schedule length shorter than that of the single functional unit version. According to the result presented in Table 1, for the system with an external forwarding hardware, processing a large application, such as the unfolded elliptic filter, the adder unit is occupied at almost every control step. Adding more functional units is the only approach that would reduce the schedule length. Table 2 shows the experimental results for the system with 2 multipliers and 2 adders.

According to the data from Table 2, even though we have added more functional units to the target system the hazard reduction percentage (ranging from 2–80 %) still relies on the characteristic of the applications as well as the pipeline architectures. For example, without the forwarding feature, in the lattice filter application, hazards can be reduced up to 45 percent in the 2-adder and 2-multiplier system. For the wave digital filter (benchmark 4), the reduction is 80%.

The experimental results from both tables show that SHARP can reduce a large number of hazards by considering all available hardware and overlapping pipeline instructions. Further, in each iteration of SHARP, the algorithm only needs to reschedule a few number of nodes. Our algorithm can also help designers choose the appropriate hardware architecture, such as the number of

pipelines, pipeline depth, the number of forwarding buffers and others, in order to obtain good performance when running applications subject to their overhead hardware costs.

5. Conclusion

Since computation-intensive applications contain a significant number of data dependencies and few or no control instructions, data hazards often occur during the execution time which degrades the system performance. Hence, reducing the data hazards can dramatically improve the total computation time of such applications. Our algorithm, SHARP, supports modern multiple pipelined architectures and applies the loop pipelining technique to improve the system output. It takes the application characteristics in the form of a pipeline data-flow graph and target system information (e.g., the number of pipelines and depth, their associated types, and their forwarding buffer mechanism) as inputs. SHARP reduces data hazards by rearranging the execution sequence of the instructions and produces a schedule in accordance with the system constraints. Not only does SHARP serve as a scheduling optimization tool, it can be a simulation tool for a system designer as well.

References

1. L. Chao, A. LaPaugh, and E. Sha. Rotation Scheduling: A Loop Pipelining Algorithm. In *Proceedings of ACM/IEEE Design Automation Conference*, pages 566–572, June 1993.

2. S. Davidson, D. Landskov, et al. Some Experiments in Local Microcode Compaction for Horizontal Machines. *IEEE Transactions on Computers*, c-30:460–477, July 1981.

3. D. D. Gajski, N. D. Dutt, A. C-H Wu, and S. Y-L Lin. *High-Level Synthesis: Introduction to Chip and System Design.* Kluwer Academic Publishers, 1992.

4. J. L. Hennesy and D. A. Patterson. *Computer Architecture A Quantitative Approach.* Morgan and Kaufmann Publisher Inc., California, 1990.

5. P. D. Hoang and J. M. Rabaey. Scheduling of DSP Programs onto Multiprocessors for Maximum Throughput. *IEEE Transactions on Signal Processing*, 41(6), June 1993.

6. R. B. Jones and V. H. Allan. Software Pipelining: An Evaluation of Enhanced Pipelining. In *Proceedings of the 24th Annual International Symposium on Microarchitecture*, volume 24, pages 82–92, 1991.

7. R. A. Kamin, G. B. Adams, and P. K. Dubey. Dynamic List-Scheduling with Finite Resources. In *International Conference on Computer Design*, pages 140–144, Oct. 1994.

8. A. A. Khan, C. L. McCreary, and M. S. Jones. A Comparison of Multiprocessor Scheduling Heuristics. In *1994 International Conference on Parallel Processing*, volume II, pages 243–250. IEEE, 1994.

9. P. M. Kogge. *The Architecture of Pipelined Computers.* McGraw-Hill, New York, 1981.

10. M. Lam. Software Pipelining. In *Proceedings of the ACM SIGPLAN'88 Conference on Progaming Language Design and Implementation*, pages 318–328, June 1988.

11. C. E. Leiserson and J. B. Saxe. Retiming Synchronous Circuitry. *Algorithmica*, pages 5–35, June 1991.

12. K. K. Parhi and D. G. Messerschmitt. Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding. In *Proceedings of the International Conference on Parallel Processing*, volume I, pages 209–216, 1989.

13. B. Ramakrishna Rau. Iterative Modulo Scheduling: An Algorithm For Software Pipelining Loops. In *Proc. 27th Annual International Symposium on Microarchtecture*, pages 63–74, November 1994.

14. B. Shirazi et al. PARSA: A PARallel program Scheduling and Assessment environment. In *International Conference on Parallel Processing*, 1993.

15. S. Shukla and B. Little. A Compile-time Technique for Controlling Real-time Execution of Task-level Data-flow Graphs. In *1992 International Conference on Parallel Processing*, volume II, 1992.

16. R. M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, 11(1):25–33, January 1967.

Table 1. Performance comparison for 1 adder and 1 multiplier system

Ben.	no-frw.		1 buf/in-frw.		in-frw.		1 buf/ex-frw.		2 buf/ex-frw.		ex-frw.	
	int/ aft	%	int/ aft	%	int/aft	%	int/aft	%	int/aft	%	int/aft	%
1	58/56	2	43/42	2	43/42	2	29/28	3	29/28	3	29/28	3
2	20/17	15	18/15	17	18/15	17	14/13	7	14/12	14	12/ 11	8
3	50/29	42	43/24	44	42/24	42	24/14	42	15/12	20	15/12	20
4	25/ 8	68	24/8	67	22/8	64	12/ 8	33	9/8	11	9/8	11
5	191/ 150	21	164/145	12	164/145	12	89/83	7	70/67	4	70/67	4
6	76/63	13	54/30	44	53/28	47	31/24	23	28/24	14	27/24	11
7	115/ 111	4	84/80	5	84/80	5	57/52	9	54/52	4	54/52	4

Table 2. Performance comparison for 2-adder and 2-multiplier system

Ben.	no-frw.		1 buf/in-frw.		in-frw.		1 buf/ex-frw.		2 buf/ex-frw.		ex-frw.	
	int/ aft	%	int/ aft	%	int/aft	%	int/aft	%	int/aft	%	int/aft	%
1	57/56	2	39/38	3	37/36	3	18/17	5	18/17	5	18/17	5
2	19/18	5	16/15	6	16/15	6	12/11	8	9/8	11	9/8	11
3	49/27	45	40/23	43	40/23	43	24/11	54	12/9	25	12/9	25
4	23/5	78	23/4	83	21/4	81	11/6	46	6/4	33	6/4	33
5	180/158	12	155/136	12	155/136	12	76/69	9	47/44	6	47/44	6
6	73/67	8	49/42	14	49/41	16	23/16	30	19/13	32	18/13	27
7	114/112	2	77/74	3	75/73	3	39/37	5	32/31	3	31/30	3