OPTIMIZING CYCLIC DATA-FLOW GRAPHS VIA ASSOCIATIVITY

Liang-Fang Cuo
lf@iastate.edu
Dept. of Electrical and Computer Engineering
Iowa State University
Ames, IA 50011

Abstract

An iterative or recursive algorithm with inter-iteration precedence relations is represented by a cyclic data-flow graph (DFG), where nodes represent operations. Such a DFG has a lower bound on the schedule length, which is determined by the loops (cycles) in the cyclic DFG. Associativity of the operations can be applied to restructure a DFG while preserving the behavior of the given recursive algorithm. We propose a measure of criticalness on regions of a DFG in order to guide the application of associativity to effectively reduce the lower bound on schedule length. Experimental results show that the transformed data-flow graph gives the best known schedules even under resource constraints.

1 Introduction

In VLSI high-level synthesis or compiler design, a program is usually transformed into a data-flow graph to expose the parallelism in the given description. Each node in a data-flow graph corresponds to an operation, such as addition, multiplication, etc. Edges represent the data flows or precedence relations among operations. The chains of precedence relations (paths) are the barriers to produce short schedules. A data-flow graph can be improved by algebraic transformations to reduce path lengths. Iterative algorithms or recursive algorithms can be modeled as cyclic data-flow graphs with delays [1]. Loop pipelining can be used to produce pipelined schedules to provide better throughput.

Algebraic transformation is usually applied to acyclic data-flow graphs to reduce the length of critical paths, which is the lower bound of the schedule lengths. In compiler design, much work has been done on algebraic transformation or so-called tree-height reduction for acyclic data-flow graphs [2, 3]. In high-level synthesis, algebraic transformations are also used to optimize acyclic data-flow graphs [4, 5].

A cyclic data-flow graph is usually treated as an acyclic graph in order to apply algebraic transformations. Delays in cyclic data-flow graphs are used to mark iteration boundaries. Possibilities of optimization over iteration boundaries are not exploited if algebraic transformations are limited to the areas without delays. Two previous works [6, 7] find schedules for data-flow graphs with both algebraic transformations (associativity and distributivity) and loop pipelining. In this paper, we propose techniques to optimize a data-flow graph with associativity according to some criteria and then find pipeline schedules using a loop pipelining algorithm in [1]. The experimental results are quite promising. Shorter pipeline schedules are obtained in almost all cases under different resource constraints.

For an acyclic data-flow graph, the length of the longest path gives a lower bound for schedule length when the schedule is not pipelined. When pipelined schedules are considered, the graph structure does not give any lower bound. For cyclic graphs, if the schedule is allowed to be pipelined, the lower bound is determined by the maximum time-to-delay ratio of every loop. The loop that gives the lower bound is called a critical loop. The objective of algebraic transformation is to reduce the lower bound in order to produce faster pipelined schedules.

In order to measure whether a certain algebraic transformation would improve the iteration bound, we define a measure of criticalness on paths. Then, certain paths good for certain algebraic transformations to reduce the lower bound can be identified. Since the graph is never cut into an acyclic graph, algebraic transformations over delays are naturally considered.
2 Criticalness of Paths

A data-flow graph (DFG) is a directed weighted graph \( G = (V, E, d, t) \) where \( V \) is the set of computation nodes. \( E \) is the edge set which defines the precedence relations from nodes in \( V \) to nodes in \( V \), \( d(e) \) is the number of delays (registers) for an edge \( e \), and \( t(v) \) is the computation time of a node \( v \). We define one iteration to be the execution of each node in \( V \) exactly once. An edge \( e \) from \( u \) to \( v \) with \( d(e) \) delays means that the computation of node \( v \) at iteration \( j \) depends on the computation of node \( u \) at iteration \( j-d(e) \). Edges without delays are precedence relations within the same iteration.

It is well known that any DFG which involves loops, feedbacks or recursions has a lower bound on the pipeline schedule length \([8]\). This iteration bound \( B(G) \) for a DFG \( G \) is given by the maximum time-to-delay ratio of all loops, i.e. \( B(G) = \max_{l \in LOOPS(p)} \frac{T(l)}{D(l)} \), where \( T(l) \) is the sum of computation times in loop \( l \), and \( D(l) \) is the sum of delay counts in loop \( l \). The loop \( l \) that gives the iteration bound is called the critical loop.

We define the criticalness of a path to reflect the effect on the iteration bound if the path length is reduced. Let \( PATHS(p) \) be the set of paths containing the path \( p \), and \( LOOPS(p) \) the set of loops containing the path \( p \). Transformations affecting \( p \) will affect the paths in \( PATHS(p) \). The time-to-delay ratio of loops in \( LOOPS(p) \) affects the criticalness of path \( p \).

**Definition 1.** Let \( G \) be a data-flow graph and \( p \) a simple path \( v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_k \). The criticalness of the path \( p \) is defined as the maximum time-to-delay ratio of all loops containing the path \( p \), i.e.

\[
\max_{l \in LOOPS(p)} \frac{T(l)}{D(l)}.
\]

After a transformation involving edges with delays, the number of delays for any path going through the restructured area should remain the same. Any transformation that follows this principle should preserve the behavior of the original DFG. The good patterns are identified in the next section. The handling of patterns over delays is discussed in Section 4.

3 Good Patterns

Based on the measure of criticalness, we can identify transformations that have the potential of reducing the iteration bound \( B(G) \). This section identifies a set of patterns in a graph to perform transformations on. These transformations can then be applied according to a given priority.

![Figure 1: Effect of Associativity](image)

Although only associativity is considered here in this paper, our experimental results, shown in Section 5, outperform the previous published results using both associativity and distributivity \([6, 7]\). We are interested in finding the shortest pipeline schedule for the given graph in order to provide good throughput. Since the iteration bound \( B(G) \) is a lower bound for the pipeline schedule length, our objective is to find an equivalent graph with lower iteration bound.

If a value is eliminated after a transformation, and is used elsewhere in the graph, the value or the node producing the value has to be preserved. The application of associativity does not introduce extra nodes unless an eliminated value has to be preserved.

The application of associativity can reduce the length of a path at the cost of increasing the length of another path. If associativity is applied in the right places, the length of the critical loop can be reduced; thus the iteration bound \( B(G) \) can be reduced.

The effect of associativity, as shown in Figure 1, can be summarized as follows.

- The paths in \( PATHS(A \rightarrow D \rightarrow E) \) are shortened.
- The paths in \( PATHS(C \rightarrow D) \) are lengthened.
- If the fanout of \( D \) is larger than 1, one more adder, denoted by Node \( Z \), is needed to preserve the value \( D \). (If the other path/loop that \( D \) belongs to is not very critical, the extra addition will probably not degrade the resource usage.)

Since the pipeline schedule is considered, those nodes not belonging to any loop does not affect the lower bound of schedule length, and they can be allocated anywhere in a schedule under the concept of loop pipelining by retiming \([9]\). Based on the above observations, we first identify those patterns where transformations can be applied without extra nodes. If the lengthened paths do not belong to any loop, the transformation has no bad effect under any resource constraints. If the lengthened paths do not belong to any almost critical loop, the extra node \( Z \) on the
lengthened path can be fit into the schedule without affecting the overall resource usage.

**Pattern ASS1**: Associativity on $D \rightarrow E$
- $D$ has a fanout of one.
- The paths in $\text{PATHS}(C \rightarrow E)$ do not belong to any loop. (Node $C$ might be a constant input to Node $E$.)

**Pattern ASS2**: Associativity on $D \rightarrow E$
- $D$ has a fanout of one.
- The loops in $\text{LOOPS}(C \rightarrow E)$ are not critical loops (or almost critical loops).

Basically, Patterns ASS1 and ASS2 can be applied to edges satisfying the conditions without overhead. Therefore, we would apply Patterns ASS1 and ASS2 even in areas where loops are not very critical. Nevertheless, if $D \rightarrow E$ belongs to a critical loop and $C \rightarrow E$ belongs to a nearly critical loop, a new critical loop may replace the critical loop that is removed. In this case, such transformation may not be favored unless further good patterns can be exposed after the transformation.

In the effort of trying to reduce the iteration bound, we focus on the critical loops. Some overhead is tolerable if the length of a critical loop can be reduced.

**Pattern ASS3**: Associativity on $D \rightarrow E$
- One of $\text{LOOPS}(D \rightarrow E)$ is critical.
- $D$ has a fanout larger than one, and the loops containing $D$ but not $E$, $\text{LOOPS}(D)$ \textbf{-} $\text{LOOPS}(D \rightarrow E)$, are not critical. (Thus, the extra copy of node $D$ will not hinder the resource usage much.)
- The paths in $\text{PATHS}(C \rightarrow E)$ do not belong to any loop, or the loops in $\text{LOOPS}(C \rightarrow E)$ are not critical.

### 4 Transformations over Iteration Boundaries

Algebraic transformations are usually applied on acyclic data-flow graphs. It is perceived that such transformation preserves the behavior when there are no delays on the edges involved. In [9], it is observed that the retiming technique preserves the behavior while allowing the delays of the data-flow graph to be moved around. Delays represent iteration boundaries. To perform algebraic transformations over iteration boundaries, we use the concept of retiming to move delays out of the area of transformation before transformation.

Retiming technique [10] has been effectively used to obtain the minimum cycle period for a data-flow graph by rearranging the delays. The technique of retiming moves around delays by using the following way: a delay is drawn from each of the incoming edges of $v$, and then a delay is pushed to each of the outgoing edges of $v$ and vice versa. A retimed graph is legal is the number of delays on every edge is nonnegative. It is shown that such movement of delays preserve the behavior of the data-flow graph if the resultant graph is legal.

Consider an associativity applied on edge $(D, E)$, as shown in Figure 2 where the round-corner box is the area of transformation. Suppose that the number of delays on $(D, E)$ is $h$ before the transformation. We first push $h$ delays backwards through Node $D$. Note that $h$ delays have to be borrowed from edge $(D, F)$ even if there are not enough delays. We need to return those delays back after the transformation so that the graph remain legal. Figure 2-(a) shows the differences in delays for edges incident on Node $D$ from the original graph, and Figure 2-(b) shows the delay differences after the transformation. Then, the final delay configuration is shown in Figure 2-(c). The $h$ delays borrowed from edge $(D, F)$ are returned, and $h$ extra delays remain on the edges $(A, E)$ and $(B, Z)$.

In this transformation, one new node $Z$ and four new edges $(A, E)$, $(B, Z)$, $(C, Z)$ and $(Z, E)$ are intro-
duced. Their delay function is defined as follows to preserve the behavior of the graph.

\[
\begin{align*}
d(A, E) &= d(A, D) + d(D, E) \\
d(B, Z) &= d(B, D) + d(D, E) \\
d(C, Z) &= d(C, E) \\
d(Z, E) &= 0
\end{align*}
\]

Notice that all the paths going through the area of transformation still have the same number of delays after the transformation. Therefore, the transformation preserves both inter-iterational and intra-iterational behaviors of the description. For example, the path \( A \rightarrow D \rightarrow E \) is transformed into path \( A \rightarrow E \) with the same number of delays; the path \( C \rightarrow Z \rightarrow E \) has the same number of delays as the original path \( C \rightarrow E \) although one more edge is added to the path.

Example: Consider the data-flow graph in Figure 3-(a), where the number of bars on an edge \( e \) represents the delay value \( d(e) \). The data-flow graph has a lower bound of 3, and the loop ADE is critical. After applying ASS3, we obtain the data-flow graph in Figure 3-(b), and the lower bound is reduced to 2.

5 Experimental Results

We have demonstrated effectiveness of the proposed technique with experimental results, which produce the best known schedules under resource constraints. First, we apply associativity to locations with matching patterns, and then we use Rotation Scheduling [1], a loop pipelining algorithm, to find pipeline schedules for the transformed graph under resource constraints.

Our results are compared against the following two systems, which perform both algebraic transformation and loop pipelining: Percolation-based scheduling (PBS) [6] and HYPER [11, 7]. Our results are as good as or better than the previous results. Notice that the two systems, PBS and HYPER, use both associativity and distributivity in their transformations, but we perform associativity only.

It is assumed that the computation time of an adder for one addition is 40ns, the computation time of a multiplier for a multiplication is 80ns, and the clock cycle period for a control step is 50ns with 10ns for the latch time of buffers. The pipelined multiplier, denoted by \( M_p \), is assumed to consist of 2 stages, each of which takes no more than 40ns. The numbers in the tables are the number of control steps.

The results for the 5-th order elliptic filter are shown in Table 1. The figures of other systems appearing in the tables are adopted from the papers referenced above. RS denote the pipeline schedule length

<table>
<thead>
<tr>
<th>Resources</th>
<th>RS</th>
<th>PBS</th>
<th>HYPER</th>
<th>TR1</th>
<th>TR2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nonpipelined Multipliers</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3A 3M</td>
<td>16</td>
<td>16</td>
<td>15</td>
<td>15</td>
<td>11</td>
</tr>
<tr>
<td>3A 2M</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>2A 2M</td>
<td>17</td>
<td>17</td>
<td>16</td>
<td>16</td>
<td>15</td>
</tr>
<tr>
<td>2A 1M</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>16</td>
</tr>
<tr>
<td>Pipelined Multipliers</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3A 3M_p</td>
<td>16</td>
<td>n/a</td>
<td>n/a</td>
<td>15</td>
<td>14</td>
</tr>
<tr>
<td>3A 2M_p</td>
<td>16</td>
<td>16</td>
<td>n/a</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>3A 1M_p</td>
<td>16</td>
<td>16</td>
<td>n/a</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>2A 1M_p</td>
<td>17</td>
<td>18</td>
<td>n/a</td>
<td>16</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 1: Results for the elliptic filters

<table>
<thead>
<tr>
<th>Pipelined Multipliers</th>
<th>Resources</th>
<th>RS</th>
<th>TR</th>
<th>Nonpipelined Multipliers</th>
<th>Resources</th>
<th>RS</th>
<th>TR</th>
</tr>
</thead>
<tbody>
<tr>
<td>3A 2M_p</td>
<td>8</td>
<td>8</td>
<td>3A 2M</td>
<td>8</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2A 2M_p</td>
<td>9</td>
<td>8</td>
<td>2A 2M</td>
<td>9</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2A 1M_p</td>
<td>9</td>
<td>8</td>
<td>2A 1M</td>
<td>10</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1A 1M_p</td>
<td>11</td>
<td>12</td>
<td>1A 1M</td>
<td>11</td>
<td>12</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Results for the all-pole lattice filters
obtained by Rotation Scheduling from the original data-flow graph (without algebraic transformation). Patterns ASS1 and ASS2 are first applied to obtain the graph used in TR1 where no extra nodes are introduced. The iteration bound is reduced from 16 to 15 in TR1. Pattern ASS3 is applied on the graph of TR1 to obtain TR2 with one extra adder introduced. The iteration bound is reduced further to 14 in TR2. As shown in Table 1, our results provide substantial improvements over the previous results.

The results for the all-pole lattice filter are shown in Table 2. The graph of TR is obtained after Patterns ASS1 and ASS2 are applied. Although the iteration bound of the graph in TR is the same as the original data-flow graph, our technique reduces the number of critical loops from 3 to 1. Hence, there are much more freedom in the scheduling phase to provide better resource usage.

References


