21.1 Introduction

Israel Koren
Balu Patel
Bhoga Menelection

Chapter 21

Data Flow Paradigm

Co-processors Using Special-purpose Micropipelines
new algorithm must be developed for the application. The performance of
the algorithm must be determined. If the estimated performance is not satisfactory, then a
preliminary estimate of the performance of the application pro-
gram is obtained. Before the detailed design of the ASIC is in-

Program Evaluation: Before the detailed design of the ASIC is initi-

ated, the performance of the application program is evaluated.

The complete procedure for designing data-driven ASICs is shown in Figure 2.1.

2.2 Design Process

Section 2.5 illustrates the design process and some conclusions are

This chapter describes the implementation of the application program in detail and is shown in Figure 2.2. The design process in Section 2.1 of the

The steps in the design process are outlined as follows. Section 2.1 describes a

Designing Co-Processors Using the Data-Flow Paradigm

Designing Co-Processors Using the Data-Flow Paradigm
A heuristic algorithm which yields optimal results.

Performance constraints.

Initial Mapping: The first step in mapping a DFG is to assign an input or output to an allocated DFG node. At this stage, the implementation details are ignored.

Figure 3.1: The design process

1. Initial Mapping
2. Area Minimization
3. Buffer Allocation
4. Layout Generation
5. Performance Est.
Designing Co-processors Using the Data-Flow Paradigm

2.3 Performance Evaluation

The standard co-processors generally suffer from the short queues of dependent descretions as well as the long queues of multiple descretions, which are often created out of order. Consequently, the main goal is to minimize the number of buffers, queues, and processors. The buffer allocation algorithm is not provided here for the sake of brevity.

Buffer Allocation: If a DFG with non-minimum path lengths is directly connected to the processor speed, then the performance of the algorithm is severely affected. The performance of the algorithm is improved when the problem is mapped to a queue-based programming model. The algorithm also allows for multiple execution levels for nodes and multiple delays, which are also included in the performance evaluation of the algorithm.

The complete application is the base structure for the evaluation of the DFG. We analyze the performance of the algorithm, conditional expressions, and loops and estimate the potential parallelization of the DFG. We decompose the DFG into the three parts of structure: algorithmic, logical, and physical.
21.3.2  Conditional expressions

21.3.1  Arithmetic/Logic expressions

In what follows, we present the data-flow graphs of the basic structures and
the ratio between the maximum overlap time and the average pipeline period in the "long" path only during $AVP(C_{\text{cond}})$ clock cycles. We denote, for the entire computation in the "short" path, the case where an additive contribution of exactly $AVP(C_{\text{cond}})$ clock cycles apart. The net

$$\frac{d}{2} - 1 = g$$

By $\mathcal{D}$ is equal to $\mathcal{G}$ if the $(d - 1)$th iteration of the selected paths is denoted $AVP(\text{short})$. Assuming a geometric distribution of the selected paths, the average number of times that the "short" path will be selected as a case study is denoted $\mathcal{P}$. The corresponding probability of passing through the "short" path is denoted $\mathcal{P}$. The average periods of the state $\mathcal{P}$ and $\mathcal{Q}$ between the corresponding average pipeline periods of the "short" path and the other in the state $\mathcal{P}$ is defined as the expected average pipeline period of the state $\mathcal{P}$ and $\mathcal{Q}$.

The worst case pipeline period is given by the longest operation in the state.

\[
\begin{align*}
\text{AVL(\text{short})} & = (\text{AVL(\text{short})})^{(d-1)} + (\text{AVL(\text{short})})^{(d)} + (\text{AVL(\text{short})})^{(d+1)} + \cdots \\
& = \text{AVL(\text{short})} \\
\end{align*}
\]

The expected latency of the complete If-then-else structure is therefore

$$\text{AVL(\text{short})} \cdot \text{AVL(\text{cond})} + \text{AVL(\text{cond})} \cdot \text{AVL(\text{short})} + \text{AVL(\text{cond})} \cdot \text{AVL(\text{cond})}$$

The probability of passing through the Then and the Else parts is denoted

**Figure 2.3:** If-then-else structure

Designing Co-processors Using the Data-Flow Paradigm
These values may be determined through simulations on typical input data.\[\text{outcomes like in a round robin fashion. By filling the loop body, we can}
\]

outfit nodes (S/M巧) that route the incoming data stream to the appropriate

module nodes (stream-1...stream-n). These synchronization nodes are shown

and then we analyze the other cases.

next estimate the performance when the loop structure produces a single result.

In general, a loop may generate a single result or a stream of results. We

use the same notation to measure and worst case measures. Theorems, we

will in turn yield average or worst case estimations respectively. Therefore, we

have supplied values of the average or worst case number of iterations, which,

in many cases, the number of iterations needed is not known at compile time.

The above estimations are based on the assumption that the incoming data is

\begin{align*}
U > D & \quad \text{then estimate} \\
& = \frac{1}{(\sum_{i=0}^{D} + D_{\text{Avg}})}
\end{align*}

\[\text{21.3.}\quad \text{Loop structures}

In this chapter, we will attempt to develop and analyze loop structures.

The above estimations are based on the assumption that the incoming data is

\[\text{21.3.}\quad \text{Performance estimation}

\]
is not a constant we may use its expected value.

Here, for simplicity, we assume that \( l \) is a constant. If the inner-iteration time

The input to the loop body is a stream of data elements with inter-arrival

of the \( f \) block and a summation tree as shown in Figure 2.3.

Next, the loop body consists of \( n \) replacement

iterations (e.g., Sum nodes). In summary, the loop body consists of \( n \) iterations of the summation tree (e.g., for (+) nodes) and then accumulates the partial results of the various

layers in one step and then updates the partial results using a computation

table. This is accomplished in two steps: and the partial results using a computation

table. The partial results are accumulated by all copies of the \( f \) block that have to be accumulated to produce the loop output. This

is achieved by a better performance and therefore we annotate a loop structure.

Figure 2.3: Single Result Loop Structure

Designing Co-processors Using the Data-Flow Paradigm
\[ u > \left( \frac{u}{(\text{lop})_{d1}} \right) ^{2} \left( \frac{(\text{rop})_{d1} + (\text{rop})_{d1}}{u} \right) + \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\text{otherwise} 

\[ \left( \frac{(\text{rop})_{d1} + (\text{rop})_{d1}}{u} \right) + \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\text{otherwise} 

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]

\[ \left( \frac{1}{(\text{rop})_{d1}} \right) J! \] 

\[ \text{otherwise} \]
Stream of results loop structure

\[
\left\{ \text{control} \right\} (f) X \mod (S, \text{Merge}) = (f) X \mod (S, \text{Merge})
\]

The minimal period of the loop structure is

\[
\left\{ \text{control} \right\} (f) X \mod (S, \text{Merge})
\]

where \( m \) is the minimal period of the \( X \) block, and \( f \) is the period of the \( f \) block. The period of the \( f \) block is shown in Figure 2.14, which guarantees the property of being equal to the minimal period of the loop structure.

Stream of results loop structure

\[
(f) X + (\text{control}) (\text{loop}) T = (\text{loop}) T
\]

Finally, the latency of the single result loop structure is

\[
[11] (f) + (\text{loop}) T + (\text{loop}) T
\]

Figure 2.14: Stream of results loop structure

Designing Co-processors Using the Data-Flow Paradigm
that point, as can be seen from the figure, the latency of the complete DFG in number marked on each arc represents the accumulated worst case latency at the analysis of this example, we use the execution times from [1]. The
DFG generated by our compiler.
Figure 2.5. show the problem and its corresponding
We demonstrate the performance estimation method through a simple nested

2.1.4 Examples

\( T(\text{first free}) + (f) T = T(\text{last free}) + (s) T + (X) T + (X)^T \)

and the corresponding latency is

\[
\begin{align*}
\text{if } a &= b \\
\text{else if } &c \neq d \\
\text{end if} \\
\text{else if } &e = f \\
\text{end else if} \\
\text{end if}
\end{align*}
\]

Figure 2.1: Nested if-then-else expression and its DFG representation

2.1.3 Performance Estimation
Performance measure is the throughput. Since the pipeline period of the designed processor is not a pipelined array, the simple most important parts of multi-operation processors, to determine the overall area for each multiplication procedure starts with the initial mapping and rescaling.

21.4 Area Minimization

cal and edge notes.

The area minimization procedure begins with the initial mapping and rescaling.

The average case pipeline period.

The average case pipeline period is 31 clock cycles, which is substantially higher than the worst case pipeline period.

In Figure 21.6, we compare the estimated values of the average pipeline period.
The example in Figure 21.6: Comparing the estimated phosphate period to simulation results for the inner conditional is 0.8. The probability to pass through the Then path.

(a) The probability to pass through the Then path.

(b) The probability to pass through the Then path.

21.4 Area Minimization
A different system area and execution time. For example, if the DFG contains
for every node, there are many implementations and each of these may lead to
The above are minimization in general is an NP-complete problem since,
Furthermore, the design process will be repeated for a large value of tolerance to
acceptable, the design process will be repeated for a large value of tolerance to
the overall area of the AADD is minimal. If the area of the final design is not
the area minimization will select an implementation for each node such that
In general, there may be many different implementations for a node, and
the latency
Implementation will increase the length of the critical path and thus increase
increase
A paralleladder should still be used for AADD because any slower
A parallel adder should still be used for AADD because any slower
A parallel adder should still be used for AADD because any slower
A parallel adder should still be used for AADD because any slower
A parallel adder should still be used for AADD because any slower
We define the freedom of a node as the amount by which the execution time
supplies time, have been used for several scheduling problems (e.g., [9]).
We (as late as possible), the gap and gap times, also referred to as slack and
gap times, are denoted by gap (as soon as possible) and gap (as late as possible).
The two time instances are denoted by gap (as soon as possible) and gap
time instances at which the output must be available to the successor nodes.
The earliest time instance at which the required inputs from the predecessors nodes are available.
Thus, the earliest time it can execute is the maximum of the earliest time
A node cannot start its execution before all the required inputs are available.

Relevant papers and on the set of implementations available for each type of node.

21.4.1 A Greedy Algorithm

The interconnected area reduces with the smaller (e.g., smaller) implementation of a
node. However, the area minimization step, however, we observe that, in most cases, the
between the nodes. Hence, the final layout may be larger than the area estimated
present. Our algorithms ignore the area required for the interconnection be-
area solution with a brunch and bound algorithm is used for optimal solution. A
node design. Therefore, a greedy algorithm was developed to obtain a "good"
area and those are its implementations of the address, then there are 5 -
An implementation of a node can be replaced by a smaller one if the execution time is smaller than its freedom. Therefore, the lower decrease in the execution time is smaller than the freedom. Hence, the decrease in the execution time is smaller than the freedom. Therefore, it makes sense to replace the node with the smallest freedom in a non-replacement list in the order of the other nodes (with larger freedom) may be reduced. Therefore, if a node with the smallest freedom is chosen for the execution time for a node is increased, the execution time for some other nodes is not increased. The node with the smallest freedom is selected. However, if no such node is found, then one node is chosen.

The Greedy algorithm attempts to minimize the overall area by selecting the node that provides maximum area savings to be replaced first. If there is more than one node that provides maximum area savings, then the area of the node with the smallest area is chosen. The Greedy algorithm will solve the solution and hence, determining the overall area of the final design.

In order to minimize the overall area, some of the nodes in the relaxed implementation can be increased without increasing the execution time of the overall design.

### Node Selection

The freedom of a node on the critical path is zero. Which is defined by the difference between the gap and gap times. Otherwise, the freedom can be increased without increasing the execution time of the overall design.

### Algorithm: Area Minimization

1. Compute the freedom of each node.
2. Let S be the set of candidate nodes for which there are improvements.
3. If S is empty, then exit.
4. Let \( S = \{ V \} \) and \( \forall S \) has maximum area savings.
5. Choose node \( V \) with smallest freedom.
6. Replace current implementation by a smaller one.
7. Go to step 1.
2.4 Examples

When the current smallest solution is smaller than the area lower bound, the current smallest solution is selected and a new area lower bound is calculated. If the smallest lower bound is selected, a new lower bound is evaluated in the next iteration. If a more efficient solution is found, then the solution is updated. The area lower bound is used to determine the next iteration of the algorithm.

Table 2.1: The area and execution time for different implementations of a

<table>
<thead>
<tr>
<th>Area (mm²)</th>
<th>0.976</th>
<th>0.977</th>
<th>0.978</th>
<th>0.98</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (clocks)</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>
Figure 21.2: An example from MIMA [14].

Designing Co-processors Using the Data-Flow Paradigm
We have also generated layouts for several implementations of the DFG of

The initial implementations were created at least once by each of the

The layouts are shown in Figure 2.12, and their area and latency are shown in Figure 2.14. The

The complete layout for the ASIC

The chosen cell placed, placed and routed program above [16]. Generation

The complete area of the DFG using BDST [21]. The area, when placed and routed using

The complete area is then obtained by combining the areas of the nodes and the

The CEO tools [18]. The DFG description of a node is presented in the language BDST [18]. The DFG description of a node as mentioned below. A library of the behavioral description of all

The first approach encounters the main layout when standard cells only (all

The second approach is layout generation from scratch, as outlined below.

The DFG for the application is translated into a netlist from which the

The handshake in the synthesis of data-driven areas is the layout generation.

2.15 Layout Generation

Figure 2.14: Area vs. time for the DFG of Figure 2.10

Layout (clock cycles)

Area (µm²)

0 5 10 15 20 25 30 35 40
Figure 2.1.12: The BDS description of a 16-bit adder

```
ENDMODEL.
ENDROUTINE.

send[0:10] = send[11:12][0:0] AND send[12:0];
ack[12:0] = ack[0:0][0:0];
ack[11:0] = ack[0:0][1:1];

ROUTEADD:
ack[0:0] =
send[1:10], send[12:0];
in[1:1:1], in[2:16:1:1];
ack[12:0], ack[11:10];
send[11:0], send[10:0];
out[1:16:1:1];

MODEL add 16
```

Designing Co-processors Using the Data-Flow Paradigm
Conclusions

Figure 21.6: Area vs. time for the DEG of Figure 21.3.
References


April 1986.


R. Spachman, Editor. OCL, J. ACM, 17:18, 7-20, October 1970.

R. Seihi and I. Dillman. The generation of optimal code for arithmetic


C. Sorenson and A. Singerman-Vincelli. 3D Pose of the


University of Massachusetts, Amherst, 1989.

B. Paul, I. Krotov, and D. K. Pradhan. Designing highly parallel ASICs


J. E. Outchoul, A switch-level timing verifier for digital MOS VLSI.

Designing Co-processors Using the Delta-Flow Paradigm.