McGill University
School of Computer Science
ACAPS Laboratory

Advanced Compilers, Architectures
and Parallel Systems

A Dynamically Scheduled Parallel DSP Architecture for Stream Flow Programming

Guoning Liao †
Guang R. Gao
Vinod K. Agarwal †

ACAPS Technical Memo 45
June 4, 1993

The authors of this paper would like to acknowledge the support of Micronet.

†Department of Electrical Engineering, McGill University

ACAPS • School of Computer Science • 3480 University St. • Montréal • Canada • H3A 2A7

Abstract

This paper presents a dynamically scheduled parallel DSP architecture for general purpose DSP computations. The architecture consists of multiple DSP processors and of one or more scheduling units. DSP applications are first captured by stream flow graphs, and then stream flow graphs are statically mapped onto a parallel architecture. The ordering and starting time of DSP tasks are determined by the scheduling unit(s) using a dynamic scheduling algorithm.

The main contributions of this paper are summarized as follows:

- *A scalable parallel DSP architecture*: The parallel DSP architecture proposed in this paper is scalable to meet signal processing requirements. For parallel DSP architectures with large configurations, the scheduling unit may become a performance bottlenecks. A distributed scheduling mechanism is proposed to address this problem.
- *A mapping algorithm*: A algorithm is proposed to systematically map a stream flow graph onto a parallel DSP architecture.
- *A dynamic scheduling algorithm*: We propose a dynamic scheduling algorithm that will only schedule a node for execution when both input data and output storage space are available. Such scheduling algorithm will allow buffer sizes to be determined at compile time.
- *A simulation study*: Our simulation study reveals the relationships among the grain-size, the processor utilization, and the scheduling capability. We believe these relationships have significant impact on parallel computer architecture design involving dynamic scheduling.

Contents

List of Figures

List of Tables

1 Introduction

In recent years, significant improvements have been achieved in computational power of programmable digital signal processors. New advances in architecture and technology have enabled DSP processors to achieve throughput of up to 16.7 MIPS and 50 MFLOPS [14]. Their high performance, programmability and low cost make them ideal for implementation in a number of real-time DSP applications, such as speech detection [2] and speech encoding. Unfortunately, we have recently experienced an even greater increase in the computational requirements of DSP applications. For instance, a computation rate of 1 GFLOPS is typical for High Definition Digital Television applications.

Currently, the only means to meet the high throughput demands of DSP applications is with special hardware, which can be quite expensive and time consuming to build at the prototyping stage. Given the success of DSP processors, one approach to obtaining a greater computational power while maintaining a rapid prototyping capability is to employ multiple DSP processors working in parallel. As an example, a system of 20 Motorola DSP96002 DSPs can yield a peak throughput of 334 MIPS and 1 GFLOPS.

In addition, parallel processing of DSP applications is well supported by the fact that there is plenty of inherent parallelism in DSP applications. A DSP application often operates on virtually infinite streams of incoming data samples. So, pipelining the signal processing operations will speedup the throughput of DSP applications.

Large grain data flow graphs are natural representations of DSP applications even without the motivation of exposing the inherent parallelism. Data flow techniques have been used by DSP designers for decades in the guise of block diagram languages. A large grain data flow graph consists of a set of nodes and a set of arcs, where a node represents a fundamental DSP task (such as FIR filtering or FFT), and an arc represents the data dependency between two nodes. In the rest of the paper, we use the term *node* and *task* interchangeably.

Synchronous Data Flow (SDF)[13] is a representative data flow technique for DSP applications. The term *synchronous* means two things: (1) the amount of input data consumed by each task in the application, and the amount of output data produced are known at compile time and are invariable, (2) the execution time of tasks is data independent and known at compile time. This fixed execution pattern allows a multiprocessor scheduler to produce a schedule at compile time. Since a static schedule can be obtained, DSP applications described by SDF graphs are implemented on *statically scheduled* parallel architectures. Examples of statically scheduled parallel DSP architectures include MOMA [12], a distributed memory machines in [16], and a 16-processor MIMD machine built of commercial DSP chips and connected by FIFO queues in [17].

DSP applications captured by SDF are just a subclass of all possible DSP applications. It should not be inferred that the use of synchronous fundamental DSP tasks is sufficient to perform every practical DSP application. The problem is that there is always a substantial amount of computation that does not fall nicely into such structures, particularly, tasks involving heuristic decision making and data dependent conditionals where a steady streaming of data throughput can not be utilized [1].

To solve the problem, *Stream Flow* [8, 9] extends SDF technique by including two data flow nodes that are not synchronous — the *switch* node and the *merge* node. Further, in stream flow, the execution time of nodes is not assumed to be fixed. Different from DSP applications captured by SDF graphs, DSP applications described by stream flow graphs can not be implemented on statically scheduled parallel DSP architectures. Due to the introduction of *switch* and *merge* nodes, and data dependent execution time, static schedules can not be obtained at compile time. The ordering and timing of stream flow graph execution can only be determined at run-time. So, implementing stream flow graphs requires dynamically scheduled parallel DSP architectures.

In a dynamically scheduled parallel architecture, the execution sequence of nodes is determined at run-time by the presence of data at the inputs of a node and the availability of storage space for the outputs.

In this paper, we propose a dynamically scheduled parallel DSP architecture for stream flow programming. The proposed parallel DSP architecture consists of tightly coupled multiple DSP processors and one or more scheduling units. A node is enabled for execution whenever there is sufficient input data and storage space for output data. When a processor is free, an enabled node may be scheduled for execution on the processor. The scheduling task is carried out on the dedicated scheduling unit outside the DSP processors.

One obvious advantage of the dynamically scheduled parallel DSP architecture is that it can be used to implement general DSP applications which have decision making and data dependent computations. Another advantage is the capability of graceful performance degradation when a DSP processor breaks down. Such a feature is especially desirable in real-time applications requiring high performance in terms of reliability.

The main contributions of this paper are summarized as follows:

- *A scalable parallel DSP architecture*: The parallel DSP architecture proposed in this paper is scalable to meet signal processing requirements. When more DSP processors are used, the scheduling unit may become a performance bottleneck. A distributed scheduling mechanism is proposed to address this problem.

- *A mapping algorithm*: A algorithm is proposed to systematically map a stream flow graph onto a parallel DSP architecture.

- *A dynamic scheduling algorithm*: We propose a dynamic scheduling algorithm that will only schedule a node for execution when both input data and output storage space are available. Such scheduling algorithm will allow buffer sizes to be determined at compile time.

- *Simulation study*: Our simulation study reveals the relationships among the grain-size, the processor utilization, and the scheduling capability. We believe these relationships have significant impacts on parallel computer architecture design involving dynamic scheduling.

The paper is organized as follows. Section 2 gives the notation and abbreviations used in this paper. The stream flow programming model is described in Section 3. Dynamically scheduled DSP architecture with a single DSP processor is detailed in Section 4 and parallel DSP architectures are proposed in Section 5. Implementing DSP applications requires mapping stream flow graphs onto the proposed parallel DSP architecture. The mapping problem is addressed in Section 6. Simulation results are presented and discussed in Section 7. Related works are addressed in Section 8. Finally, we summarize and point out future research directions in Section 9.

2 Preliminaries

The following notation and abbreviations are used throughout this paper.

2.1 Basic Notations

$G = (V, E)$: the stream flow graph, where V is a set of nodes and E is a set of arcs.

$N_v = |V|$: the number of nodes in a stream flow graph.

$N_e = |E|$: the number of arcs in a stream flow graph.

$M_i, i = 1, 2, ..., N_v$: a node in a stream flow graph.

d_i: the degree of M_i.

$P_1, P_2, ..., P_n$: DSP processors.

a_{in_k}: an input arc of a node.

a_{out_l}: an output arc of a node.

m_k: the number of tokens consumed by the node's each firing from a_{in_k}.

n_l: the number of tokens produced by the node's each firing in a_{out_l}.

R_n: an address register.

N_p: the number of data tokens produced when a node fires.

N_c: the number of data tokens consumed when a node fires.

N_s: the size of the buffer allocated to an arc.

N_b: the current number of data tokens in a buffer.

$NODE(1 : N_v)$: the node-array of a stream flow graph.

$ARC(1 : 2N_e)$: the arc-array of a stream flow graph.

$D - bit$: double-count bit.

$p/\overline{c}$: the $producer/\overline{consumer}$ bit.

w_i: the weight of node M_i.

$c_{i,1}, c_{i,2}, ..., c_{i,d_i}$: the communication costs of M_i.

c_i: the communication cost from M_i to M_{i+1}.

w: the load constraint for a homogeneous processor.

N_{c1}: the number of data tokens consumed by M_1's each firing.

C_p: the processing capability of processor per time unit in terms of weight.

T_{in}: the time interval of input data token at M_1.

2.2 Abbreviations

DSP: Digital Signal Processing.

FIR: Finite Impulse Response.

SDF: Synchronous Data Flow.

PM: Program Memory.

DM: Data Memory.

SPU: Signal Processing Unit.

MDP: Main Data-Path.

SU: Scheduling Unit.

NSR: Node Status Register.

SP: Scheduling Processor.

SGM: Signal Graph Memory.

SDT: Scheduling Data Table.

NPAP: Node Program Address Pointer.

3 Stream Flow Programming Model

Stream flow programming model, as a representation of DSP applications, consists of two parts: capturing DSP applications with stream flow graphs, and the enabling rule that guides the execution of stream flow graphs. In this section, two kinds of nodes of stream flow graphs are defined; a stream flow graph analysis technique is described; and finally the enabling rule is presented.

3.1 Regular Nodes and Non-Regular Nodes

A stream flow graph consists of a set of nodes and a set of arcs, where a node represents a task and an arc represents the data dependency between two nodes. Stream flow graph differs from other data flow techniques in that it contains two kinds of nodes: *regular* and *non-regular nodes*, which we will define in this section.

3.1.1 Regular Nodes

Suppose a node has i input arcs, $(a_{in_1}, a_{in_2}, ..., a_{in_i})$ and j output arcs, $(a_{out_1}, a_{out_2}, ..., a_{out_j})$. When the node fires, it consumes $m_1, m_2, ..., m_i$ tokens from each of the input arcs respectively and produces $n_1, n_2, ..., n_j$ tokens on each of the output arcs respectively. If $m_1, m_2, ..., m_i$, and $n_1, n_2, ..., n_j$ are predetermined in compile time, then the node is said to be a *regular* node. Figure 1 depicts a regular node. The number of data tokens consumed and produced by the node on each arc is marked near the arc; for example, m_1 indicates the number of tokens consumed from input arc a_{in1} when the node is fired.

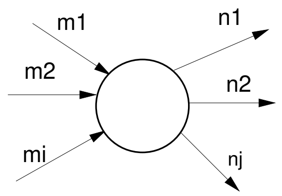

Figure 1: Regular Node

The difference between a *synchronous* node in *SDF* and a *regular* node in stream flow is that the execution time of a *regular* node is not specified, but could be variable. The expressive power of *regular* nodes is limited. For instance, we cannot express simple conditionals using *only* regular nodes.

3.1.2 Non-Regular Nodes

Since the expressive power of regular nodes is limited, we can express conditionals, recursions, as well as loops by adding two non-regular nodes, namely switch and merge.

The switch node has one input arc, one control arc, and two output arcs which are designated as T and F. The data tokens presented on the control arc are *boolean tokens*. If the boolean token is *true*, the input data are copied to the T output arc, otherwise to the F arc.

The merge node has two input arcs which are designated as T and F, one control arc, and one output arc. If the boolean token presented at the control arc is *true*, then the data at the T arc are copied to the output arc; otherwise the the data at the F arc are copied to the output arc.

Switch and merge nodes are not regular nodes because the number of data tokens consumed and produced is determined during the run-time according to the value of boolean tokens on the control arcs. Since only switch and merge nodes are non-regular nodes, when a non-regular nodes appears, we will use **switch** and **merge** specifically, and simply use *node* to mean *regular* node.

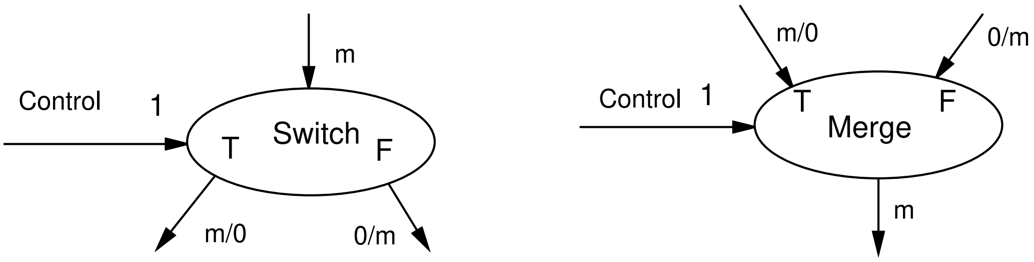

Figure 2: Switch and Merge Nodes

3.2 Stream Flow Graph Analysis

We learned that a node may consume and produce multiple data tokens when it fires. Arbitrary interconnection of nodes may lead to data token accumulation in some arcs. Let's examine an example in Figure 3. When each of M_1, M_2 and M_3 fires once, there will be a token accumulated in arc $a0$. Since the stream flow graph is supposed to be applied to infinite input data, the graph in Figure 3 will result in requiring infinite memory to hold the accumulated data.

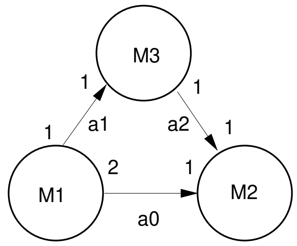

Figure 3: Defective Stream Flow Graph

Stream flow graphs like the one in Figure 3 imply improper designs of DSP applications. Intuitively, in a stream flow graph, the data tokens on each arc should be balanced without accumulation during the course of execution. Since an arc is connected to exactly two nodes, we can formulate a flow balance equation for each arc in terms of the number of firing of the nodes. It is observed that the equation for each arc must have a solution if, in the graph, none of the arcs has token accumulation during the course of execution. The system of equation is called the *state equations* [9].

Take the graph in Figure 4 for example. If the nodes M_1, M_2, M_3, and M_4 are fired S_1, S_2, S_3, S_4 times repetitively, then the state equations standing for the flow balance of the arcs for the graph are:

$$arc1: \quad 1 * S_1 = 2 * S_2$$

$$arc2: \quad 1 * S_1 = 2 * S_3$$

$$arc3: \quad 1 * S_2 = 10 * S_4$$

$$arc4: \quad 1 * S_3 = 10 * S_4$$

A possible solution to the above state equations is (20, 10, 10, 1).

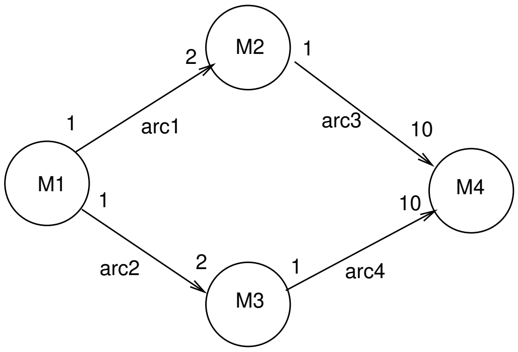

Figure 4: An example stream flow graph

3.3 Enabling Rule

The enabling rule gives the condition under which a node may be executed to achieve correct computation result. In traditional data flow techniques, a node is enabled when there is sufficient input data. However, the enabling rule used by the stream flow programming model considers both input data and output storage space. A node is enabled only when there is sufficient input data and output storage space. Formally, the enabling rule is defined as follows.

Definition 3.1 (Enabling Rule) *Consider a node with i input arcs and j output arcs. The node becomes enabled when each input arc $a_{in_k}, k = 1, 2, ..., i$ has at least m_k tokens, and for each output arc $a_{out_l}, l = 1, 2, ..., j$, there are at least n_l empty slots.*

We will see in the next section that the enabling rule permits us to construct a dynamic scheduling algorithm and determine the memory requirements when implementing DSP applications captured by stream flow graphs on our parallel DSP architecture.

4 Single Processor Architecture and Scheduling Unit

4.1 Single Processor Architecture

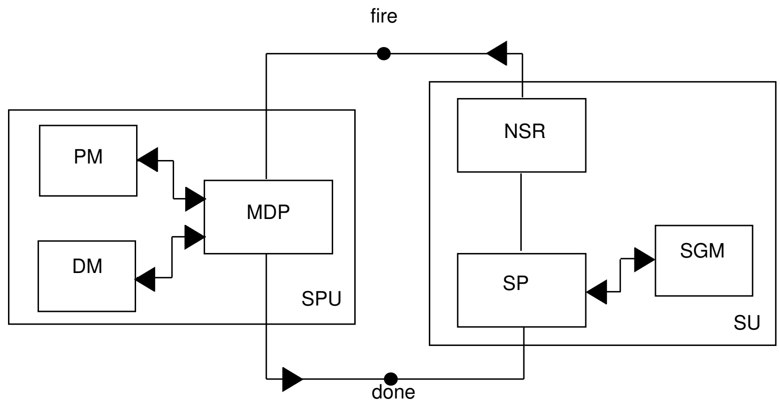

Figure 5: Single Processor Architecture

The single processor architecture consists of a DSP processor which is designated as *signal processing unit* (*SPU*) and a dynamic scheduler which is designated as *scheduling unit* (*SU*) as shown in Figure 5. *SPU* executes enabled nodes, and *SU* determines the order and the starting time of the node execution. Our architecture is referred to as dynamically scheduled architecture, because the ordering and the timing are determined at run time by *SU* according the availability of enabled nods and the SPU. *SPU* has a *program memory* (*PM*), a *data memory* (*DM*) and a *main data path* (*MDP*). *SU* consists of a *scheduling processor* (*SP*), a *signal graph memory* (*SGM*) and *node status registers* (*NSR*).

The whole procedure of program execution is described step by step. First of all, a DSP application captured by a stream flow graph is compiled into a tuple (P, S). The P-portion, is a set of *node programs*, which are sequences of instructions performing fundamental DSP operations; while the S-portion keeps the structure of the stream flow graph, and is kept in *SU* for scheduling purpose. Then *SPU* starts to execute a node program when the first instruction address of the node program is presented at the *fire* link from *SU* to *SPU*. During the execution of the node program, *SPU* fetches input data from *DM*, operates on the data following the instructions, and puts output data back to *DM* which is to be consumed by successive nodes. In our architecture, buffer memories are allocated to arcs to allow data be read from and written to *DM*. After the termination of the execution, *SPU* sends the first instruction address of the node program through the *done* link to *SU* to flag the completion of processing, and is ready to process another enabled node. Finally, upon receiving the first instruction address of the terminated node program, *SU* will apply the enabling rule, find newly enabled nodes, and send the address of the first instruction of one of the enabled node programs to *SPU* through *fire link*.

As described in the above, the first instruction addresses are used to execute (fire) node programs and flag the termination (done) of node programs. The first instruction addresses can be called *fire tokens* when they are presented on the fire link, or *done tokens* when presented on the done link.

To facilitate the receptionœ[34 of fire tokens, a special register, named the *node program address pointer* (*NPAP*) is used. A *fire* token is first loaded into the *NPAP*. *NPAP* is then used to fetch the first instruction of the node program. To initiate the execution of a node program, two instructions are required.

```
MOVE  NPAP, Rn;
JMP Rn;
```

The *MOVE* instruction copies the contents of *NPAP* to an address register, R_n, and the *JMP* instruction branches to the first instruction of the node program.

SPU and *SU* interacts with each other through the flow of *fire* and *done* tokens. The mechanism of accepting a new *fire* token and sending out a *done* token is triggered by the termination event of current node program. The diagram depicting the implementation of *SU* and *SPU* interface is shown in Figure 6.

The following steps describe the procedure of starting a node program in *SPU*:

1. Apply the contents of NPAP to the data bus.
2. Move the contents of the transmit register to done token queue.
3. Copy the contents of NPAP to transmitter register.
4. Move a new fire token into NPAP from the top of the fire token queue.

4.2 Scheduling Unit

As indicated in the above section, the ordering and the starting time of node program execution is determined by *SU* at run-time. *SU* starts dynamic scheduling by checking if there is any node enabled as a consequence of the completed node. In order to determine which node is enabled, *SU* needs to know the structure of the stream flow graph being

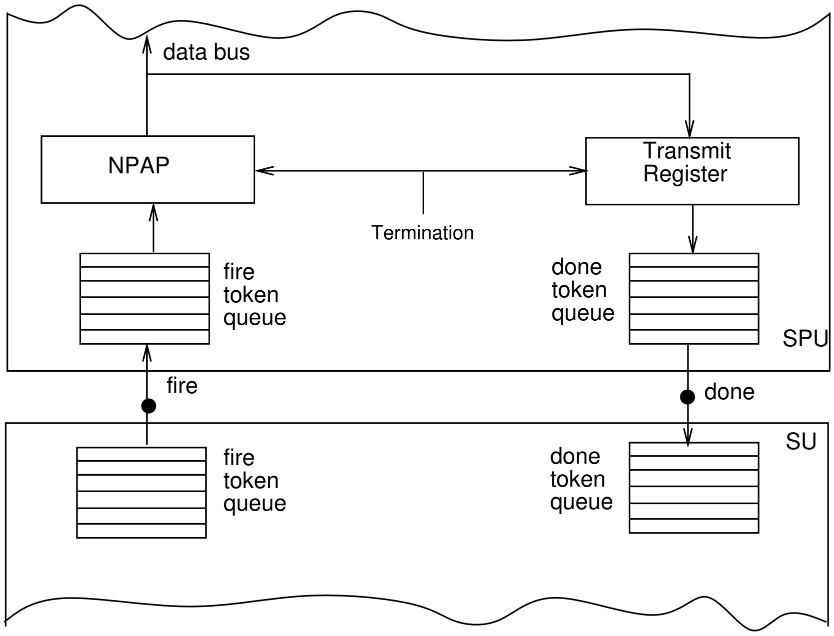

Figure 6: Interface of SPU and SU

processed. This section describes how to compile stream flow graphs in *SU*, followed by a description of the dynamic scheduling algorithm, and some precautions in using the dynamic algorithm.

An arc has four parameters: (1) the number of data tokens produced by the node that put data to the arc (producer), N_p; (2) the number of data tokens consumed by the node that removes data from the arc (consumer), N_c; (3) the buffer size allocated to the arc, N_s; and (4) the number of data tokens left in the buffer, N_b.

An arc has three states which are used to determine which node can be enabled:

1. $N_b > N_c$ and $N_s - N_b > N_p$: in this state, the consumer has sufficient data and the producer has enough output space, so both consumer and producer are allowed to fire;

2. $N_b > N_c$ and $N_s - N_b < N_p$: in this state, the consumer has sufficient data and the producer has not enough output space, so only consumer is allowed to fire;

3. $N_b < N_c$ and $N_s - N_b > N_p$, consumer does not have sufficient data and the producer has enough output space, so only producer is allowed to fire.

If all arcs connected to a node are in the state to allow the node to fire, this node is enabled to fire.

4.2.1 Stream Flow Graph Compilation in *SU*

A stream flow graph is compiled in *SU* in the following ways in this paper: (1) the structure of the stream flow graph is represented by two arrays: the node array and the arc array; (2) the number of data tokens consumed and produced by node programs is compiled into *Scheduling Data Table* (*SDT*); (3) Whether a node can be enabled is finally determined by the contents of the *node status registers* (*NSR*).

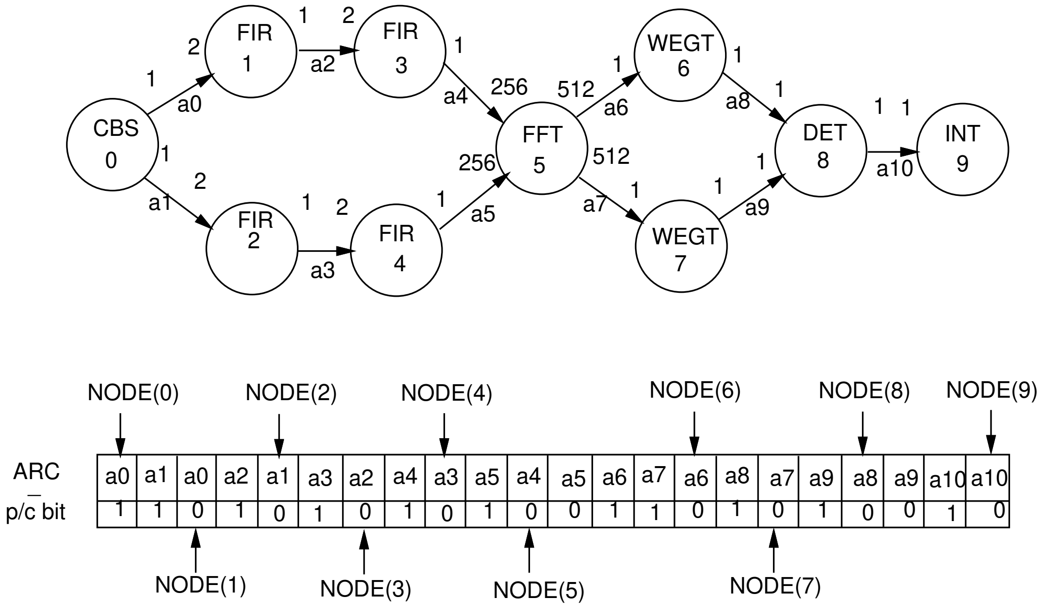

Figure 7: Spectrum analysis and its structure representation

The precedence relationship are expressed by the node array $NODE(1:N_v)$, and the arc array $ARC(1:2N_e)$. The i-th element in the node array, $NODE(i)$ ($1 \leq i \leq N_v$), gives the starting point of the arc list connected to the i-th node. The arcs connected to the i-th node are stored in $ARC(j)$, where $NODE(i) \leq j < NODE(i+1)$. $ARC(j)$ has two items: a pointer to the *Scheduling Data Table* (*SDT*) and a *producer*/$\overline{consumer}$ ($p/\overline{c}$) bit. If the $p/\overline{c}$ bit is '1', the i-th node is a producer, otherwise a consumer. We use spectrum analysis to explain how to compile a stream flow graph in *SU* which is shown in Figure 7.

Each arc has an entry in *SDT*, containing the location of the producer and the consumer status registers, N_s, N_p, N_c and N_b. The SDT of the sample graph is shown in Table 1. Each node has a *NSR* to indicate whether it can be enabled. *NSR* has three fields — the enable count, the reset count and the the first instruction address of the node program in *PM*. When the enable count of a node is decremented to 0, the node is enabled, the first instruction address of the node program is sent to *SPU* as a fire token, and the enable count is set to the value of the reset count. The reset count is a constant number, and equals to the degree of the node, which means that the node has to receive permission to fire from each of its connected arcs. The NSR for the sample stream flow graph is shown in Table 2. The enable count values in Table 2 is obtained by assuming that there is no data token in all arcs at the beginning of stream flow graph execution.

Table 1: SDT for the sample graph

Arc	Producer	Consumer	Ns	Np	Nc	Nb
a0	node 0	node 1	2	1	2	0
a1	node 0	node 2	2	1	2	0
a2	node 1	node 3	2	1	2	0
a3	node 2	node 4	2	1	2	0
a4	node 3	node 5	256	1	256	0
a5	node 4	node 5	256	1	256	0
a6	node 5	node 6	512	512	1	0
a7	node 5	node 7	512	512	1	0
a8	node 6	node 8	1	1	1	0
a9	node 7	node 8	1	1	1	0
a10	node 8	node 9	1	1	1	0

Table 2: Node Status Register

	Count		
Node	Enable Count	Reset Count	Node Address
node 0	2	2	address 0
node 1	1	2	address 1
node 2	1	2	address 2
node 3	1	2	address 3
node 4	1	2	address 4
node 5	2	4	address 5
node 6	1	2	address 6
node 7	1	2	address 7
node 8	2	3	address 8
node 9	1	1	address 9

4.2.2 Dynamic Scheduling Algorithm

The dynamic scheduling algorithm used by *SU* is described as follows.

1. *A done token of the i-th node is received. NODE(i) is fetched from the node-array.*

2. *Fetch the SDT pointer and the* $p/\overline{c}$ *bit from ARC (NODE(i)).*

3. *Fetch the entry of the arc in SDT.*

4. *If the* $p/\overline{c}$ *bit is 1,* $N_b \leftarrow N_b + N_p$*, otherwise* $N_b \leftarrow N_b - N_c$*.*

5. *If* $N_b > N_c$*, the consumer's enable count is fetched and decremented by one. If the enable count reaches zero, the consumer is enabled, the fire token of the consumer sent to the fire token queue, and the enable count set to the value of its reset count.*

   If $N_S - N_b > N_p$*, the producer enable count is fetched and decremented by one. If the enable count counts zero, the producer is enabled, the fire token of the producer sent to the fire token queue, and the enable count set to the value of its reset count.*

6. $NODE(i) \leftarrow NODE(i) + 1$.

   If $NODE(i) < NODE(i+1)$*, then go to step 2, otherwise continue.*

7. *If the done token queue is not empty, get another done token and go to step 1; otherwise wait for done token.*

4.2.3 Precautions

At this point, it is necessary to point out that the dynamic scheduling algorithm has a potential danger of violating the enabling rule. Let's examine the following scenario: An updated arc is in the state that allows both consumer and producer to fire. Then both the enable counts of both consumer and producer will be be decremented by one. Suppose the producer is enabled to fire and the consumer's enable count has a value of 1. The consumer waits for *one* more count down signal from another arc. The producer may complete execution before the consumer's enable count is decremented to 0. Now the arc has definitely enough data for consumer to fire. Unfortunately, this will make the consumer's enable count be decremented to 0, causing the consumer to fire before receiving permission to fire from the right arc. Obviously, the firing of consumer violates the enabling rule and results in an incorrect computation!

A double-count bit, *D-bit*, is introduced to each entry of *SDT* to prevent violations of enabling rule. The usage of *D-bit* is described as follows:

- *a D-bit and preset it to 0.*
- *If an updated arc allows both consumer and producer to fire and D-bit is 0, send each node a count down signal and set the D-bit to 1.*
- *When the arc is updated again,*
  - *D-bit is set, and $p/\overline{c}$ bit is 1:*

    If both producer and consumer are allowed to fire, a count down signal is sent to producer only and keep D-bit set; If only consumer is allowed to fire, D-bit is set to 0.
  - *D-bit is set, and $p/\overline{c}$ bit is 0:*

    If both consumer and producer are allowed to fire, a count down signal is sent to consumer only and keep D-bit set; If only producer is allowed to fire, D-bit is set to 0.

If the ***D-bit*** is used, a node will not receive more than one count down signal continuously from one of its arc. Thus, correct execution sequence is enforced.

4.3 Determination of the Size of Buffers

Our focus has been on the architectural design, yet the size of buffers allocated to arcs is an important factor in determining the states of arcs and influences the ordering and timing of node program execution. Now we discuss how to determine buffer sizes.

In the stream flow model, a node is enabled if two conditions are met: (1) there is enough data tokens at input arcs, (2) there is enough storage space at output arcs. Such enabling rule permits us to determine the buffer size allocated to arcs at compile time, although dynamic scheduling algorithm is used.

First, we define the deadlock in stream flow graph execution.

Definition 4.1 *A* **deadlock** *happens when the buffer allocated to an arc has neither sufficient data tokens to enable the consumer to fire nor enough empty space for the producer to fire.*

In implementing DSP applications, the allocated buffer size must be large enough to avoid deadlocks. For a stream flow graph having solutions to the state equations, we can easily determine the buffer size that will not cause deadlocks. Suppose the producer of an arc produces N_p data tokens when firing, and the consumer consumes N_c when firing, then we have

$$S_p * N_p = S_c * N_c$$

where S_p and S_c are the solution for the producer and the consumer respectively. In worst case, the producer is ordered to execute continuously S_p times, followed by S_c time execution of the consumer. So, a buffer size of $S_p * N_p$ is enough to sustain stream flow graph execution without causing deadlocks.

The minimum buffer size can be found by lazy evaluation of the stream flow graph, where a node is not enabled if, at its output arcs, there is sufficient data for consumption by its immediate successors. By examining the profile of the number of data tokens in an arc during lazy evaluation, we may determine the minimum buffer size to be the maximum value of the arc's profile.

5 Parallel DSP Architectures

So far, we have addressed the design of single processor architecture for stream flow programming. As stated in the introduction section, multiple DSP processors are used to implement high throughput DSP applications. In this section, we address the architectural design of a multiple-processor system.

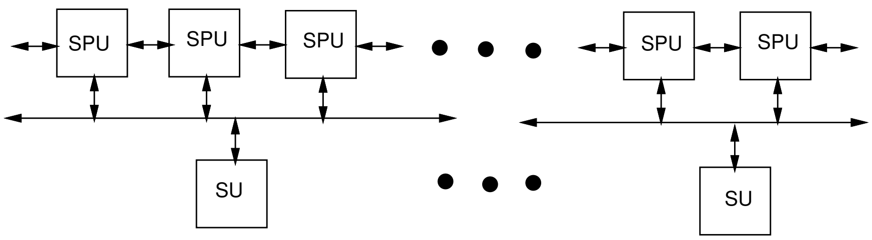

Figure 8: The parallel DSP architecture

Our parallel DSP architecture is composed of multiple SPUs and one SU. There is no restriction on what interconnection topology the parallel architecture has to assume. However, we are specially interested in looking into linearly connected parallel DSP architectures to speedup the throughput of the DSP applications by pipelining. As we described earlier, pipelining is well supported by the fact that DSP applications are applied to infinite streams of incoming data.

Figure 8 depicts our parallel DSP architecture, where *SPUs* are interconnected in a linear array.

When implementing a stream flow graph on our parallel DSP architecture, more *SPUs* are added if more processing power is required to meet real-time constraints, such as the signal rate. Since the architecture can be expanded by adding *SPUs*, *SU* will likely become a performance bottleneck in large configurations. Therefore it is desirable to increase the processing capability of *SU*.

5.1 Implementation of Distributed Scheduling Mechanism

In this section, we propose a distributed scheduling mechanism, which is used to increase the scheduling capability and avoid performance bottlenecks. Here, *SPUs* are partitioned into clusters, with one *SU* handling the scheduling processing of one cluster. A stream flow graph is partitioned into sub-graphs and assigned to clusters of *SPUs*. Since *SU* belonging to a cluster is only responsible for scheduling the sub-graph assigned to the cluster, the scheduling load is distributed among a number of *SUs*.

However, a *SU* belonged to a cluster has to know whether the input data to the assigned sub-graph is available from the cluster in the previous pipelining stage, and whether output data produced by the sub-graph have been consumed by the cluster in the next pipelining stage. So, *SUs* have to exchange scheduling data (done tokens) over partitions with neighboring clusters.

Exchanging scheduling data is facilitated by adding two full-duplex signal lines to a *SU*. One of the lines is connected to the *SU* of the cluster in the previous stage, while the other is connected to that of the next stage. The local SU accepts *done* tokens from three sources — the local cluster, the cluster in the previous stage, and the cluster in the next stage. The block diagram of the circuitry performing *done* token exchange mechanism is shown in Figure 9.

A *SPU* simultaneously sends done tokens to the *SU* of the cluster in the previous stage and the *SU* of the cluster in the next stage. Note that only the done tokens of the nodes at the front and end partitions are exchanged. Therefore, two circuit modules are used to check the done tokens from the local cluster whether they belong to the front or end partition. The done tokens of those nodes in the front partition are stored in an identity memory of the front check module. Those of the rear end partition are stored in the rear check module. If a match is found with those done tokens stored in the identity memory, a copy of the incoming *done* token is sent to the cluster of either the previous stage or the next.

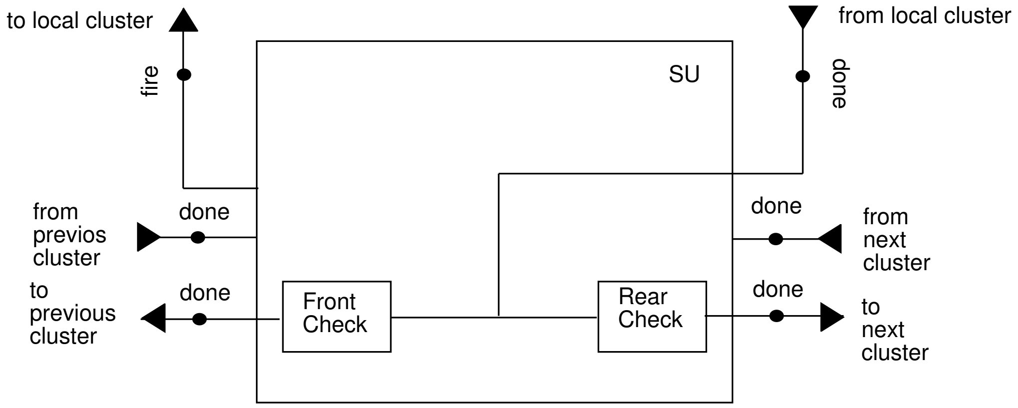

Figure 9: The block diagram for *done* token exchange mechanism

6 Mapping Stream Flow Graphs

Implementing a stream flow graph on a parallel DSP architecture requires that the graph be partitioned and assigned to an individual processor. Finding a partition and assigning processors is called *mapping*. We first formulate the mapping problem, then present a systematic mapping algorithm for implementing stream flow graph on our parallel DSP architecture.

6.1 Problem Formulation

A node, M_i, in a stream flow graph has an associated execution time, say $w_i > 0$. Suppose M_i has a directed arc to M_{i+1}. At the end of M_i execution period, data tokens are available for consumption by M_{i+1}. M_{i+1} may access these data without delay if both node reside in one SPU. If instead, these data are produced in a different SPU, there must be an explicit communication. In this case, the arc between M_i and M_{i+1} is exposed. We assume that when an SPU is engaged in communication with another, it can not execute node programs. We label this communication cost by c_i.

We formulate the mapping problem by looking at the signal rate constraint. Suppose M_1 is the initial node connected to the outside world. The number of data tokens consumed by M_1 is given by $S_1 * N_{c1}$, where N_{c1} is number of data tokens consumed from its input. The arriving interval of data tokens at the input of M_1 is given by T_{in} time units. The load constraint to meet the signal rate is given by

$$w = S_1 * N_{c1} * T_{in}$$

If a sub-graph is assigned to processor P_i, the load, L_i is given by

$$L_i = \sum_j w_j + \sum_k c_k,$$

where w_j is the execution time for a node in the sub-graph, and c_k is the communication cost of an arc on the two ends of the sub-graph.

Definition 6.1 *The mapping problem under signal rate constraint for a stream flow graph is to find a partition such that every processor's load is less than or equal to w.*

6.2 Mapping Algorithm

The mapping algorithm consists of the following four steps: (1) *expanded precedence graph* *EPG* construction, (2) node coalescence, (3) graph conversion, and (4) graph partition.

6.2.1 Expanded Precedence Graph Construction

Since the number of firings for a node might be greater than one, the stream flow graph has to be expanded to reflect the multiple firings before the stream flow graph is mapped. Here we define a specific solution to the state equations.

Definition 6.2 *A solution of the state equation is a* **basic solution** *if the greatest common divisor is 1.*

Basic solution is the smallest solution to the state equations. It also defines the iteration of a DSP application. When all nodes are fired the number of times specified by the basic solution, we say that the DSP application has just finished an iteration. A directed graph representing the computations of an iteration is called *expanded precedence graph* (*EPG*).

A stream flow graph is expanded by the number of firing times of node programs. The procedure to construct an *EPG* from a stream flow graph is described as follows:

1. *Construct state equation of the stream flow graph.*
2. *Find the basic solution of the state equation,* $(S_1, S_2, ..., S_{N_n})$.
3. *Reconstruct the stream flow graph.*

(a) *The weight of each node is given by* $w_i = S_i * w_i$;

(b) *Suppose there is a directed arc from* M_i *to* M_{i+1}. *Then the communication cost of the arc is given by*

$$c_i = S_i * N_p * c_0,$$

where N_p *is number of tokens produced by* M_i *and* c_0 *is the communication cost for passing one data token from a processor to a neighboring processor.*

6.2.2 Node Coalescence

Node coalescence avoids assigning two heavily communicating nodes to different processors. The criteria of coalescing nodes is given by two theorems. An example is given to further explain the intuition of node coalescence.

Theorem 6.1 *Suppose a stream flow graph has a partitioning on* n *processors of weight* w, *then a similar partition or partition of less weight exists even if we merge node* i *and* $i+1$ *provided the communication cost between node* i *and node* $i+1$ *is equal to or greater than the weight of node* i *plus the sum of communication cost of all other arcs of node* i.

A proof of this theorem can be found in Appendix A. The following theorem could be easily proved in the same way as the above theorem.

Theorem 6.2 *Suppose a stream flow graph has a partitioning on* n *processors of weight* w, *then a similar partition or partition of less weight exists even if we merge node* i *and* $i+1$ *provided the communication cost between node* i *and node* $i+1$ *is equal to or greater than the weight of node* $i+1$ *plus the sum of communication cost of all other arcs of node* $i+1$.

The above two theorems mean that the weight of a partition can be less even if two nodes are coalesced given the conditions are met. Node coalescence of a stream flow graph is accomplished by the following procedure:

1. *Start with the first node program module. Let* $i = 1$.

2. *For* M_i, *check each of its arcs,* $c_{ik}, k = 1, 2, ..., d_i$, *and the node across the arc,* M_j, *to see if following conditions are met:*

$$\sum_{l=1, l\neq k}^{d_i} c_{il} + w_i \leq c_{ik}$$

$$\sum_{l=1,l\neq k}^{d_j} c_{jl} + w_j \leq c_{ik}$$

If one of the conditions is met, then merge node M_i and M_j.

3. *Let $i = i + 1$, repeat 2 until $i = N_v$.*

An example of node coalescence is shown in Figure 10. If the left node and the right node are assigned to two different processors. The load for the left side processor with the left node is

$$19 + W_l$$

where W_l stands for the rest of the load of the processor; while the load for the right side processor with the right node is

$$13 + W_r$$

where W_r stands for the rest of the load of the processor. After coalescing the two nodes, and assigning both nodes to the left side processor, the load for the left side processor will be

$$14 + 3 + W_l < 19 + W_l$$

and the load for the right side processor will be

$$3 + W_r < 13 + W_r$$

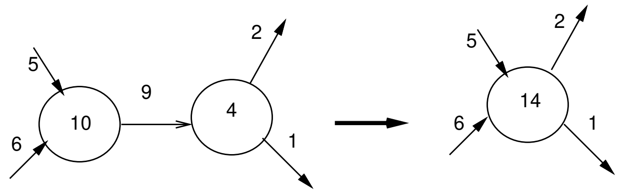

Figure 10: The merge of nodes

6.2.3 Graph Conversion

Generally the stream flow graph is not a chain structured graph, here we study how to convert an arbitrary stream flow graph into a chain structured one. We first studied 2 simple cases, and then a conversion rule is conceived based on the results of the two cases.

Case 1: Removing transitive arcs

Definition 6.3 *An arc $A \rightarrow B$ is said to be* **transitive** *if there exists a path from A to B.*

For example, arc (A,B) is a transitive arc. Since the precedence relationship between A and B is automatically satisfied by (A,C) and (C,B), removing (A,B) will not change the precedence relationship of node A and B. By removing arc (A, B), we add the communication cost of (A,B) to arc (A,C) and (C,B).

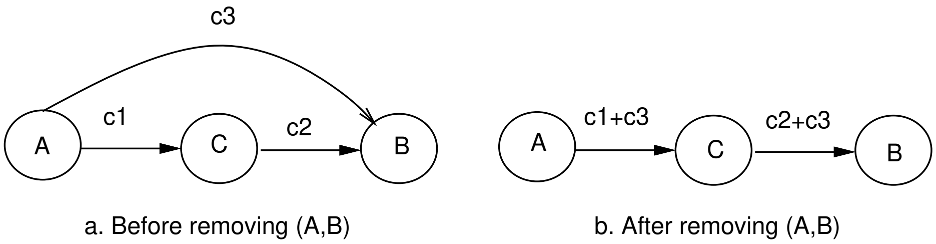

Figure 11: Removing transitive arcs

Case 2. Simple conversion

Given a stream flow graph, where M_x and M_y are two parallel nodes between node M_i and M_j as shown in Figure 12a. This graph is converted to a chain structured graph by eliminating a parallel arc and moving the communication cost of that arc onto those arcs on a path from the original node to the destination node of the eliminated arc as shown in Figure 12a.

As is shown in Figure 12b, M_x is chosen as the first node to convert the graph into an linear array. Here M_y could also be chosen as the first node after M_i. Which node comes first depends on which of the values is smaller, $w_x + c_2 + c_3$ or $w_y + c_1 + c_4$. An intuition for this arrangement is that when we partition a stream flow graph, we always take the node which will incur the smallest load to be included in a partition. This will guarantee that if a partition of w is able to take the extra load, the partition will always have a node of the smallest weight available to choose. Otherwise, a node with larger load may prevent other nodes with smaller load from being included in a partition without surpassing the partition load constraint of w. Figure 12c depicts the resulting graph by just removing the transitive arc (M_x, M_j).

General Stream Flow Graph Conversion

Conversion rule for general stream flow graphs is based on the previous two cases. A stream flow graph can be partitioned into segments containing concurrent nodes. Given a subgraph, where there are $g+1$ nodes in between M_i and M_j, it is converted to a chain structured graph in the following step:

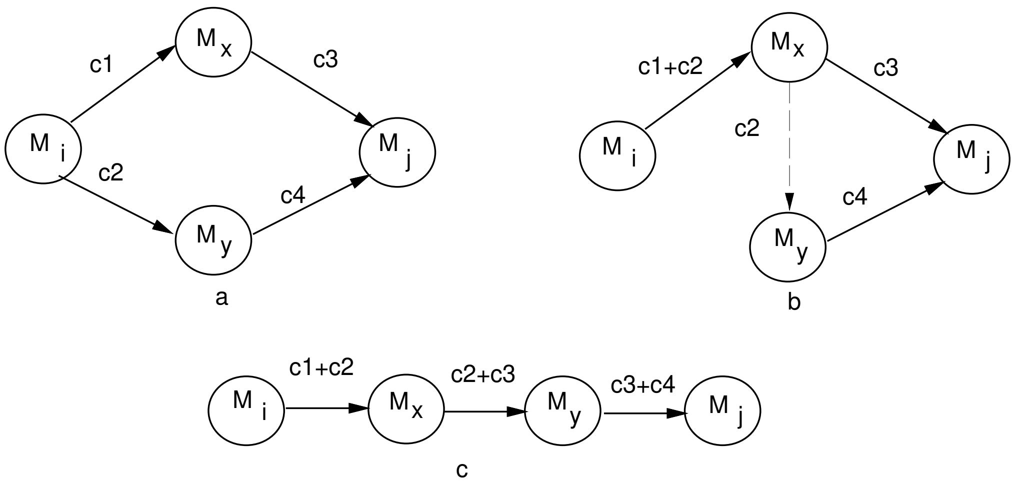

Figure 12: Simple conversion

1. *Starting from M_i, choose a node that has an arc connected to it, whichever has the smallest sum of weight and the communication cost across the cut. Name this node as M_x.*

2. *Keep the arc from M_i to M_x and eliminate all other arcs by moving the communication costs of those arcs to the arc from M_i to M_x. Find a path from M_x to every node whose arc from M_i is eliminated. Add the communication cost of that arc to the arcs along the path. If such path does not exist, add an arc from M_x to the node and assign a communication cost of the removed arc.*

3. *Removing transitive arcs if there are any.*

4. *Starting from M_x, choose another node M_{x+1} according to the same rule as in 1.*

5. *Repeat Step 2 and Step 3.*

6. *Repeat Steps 4 and 5 until the last node, M_{x+g} in between M_i and M_j is reached.*

As an example, we show a sub-graph with 5 nodes in between M_i and M_j. Figure 13b shows the graph when M_x is chosen and the rest of the arcs connecting to M_i are eliminated. Figure 13c shows the graph after removing transitive arcs. Figure 13d shows the result sub-graph one more conversion iteration, and Figure 13e shows the final result after removing the transitive arc from M_{x+3} to M_j.

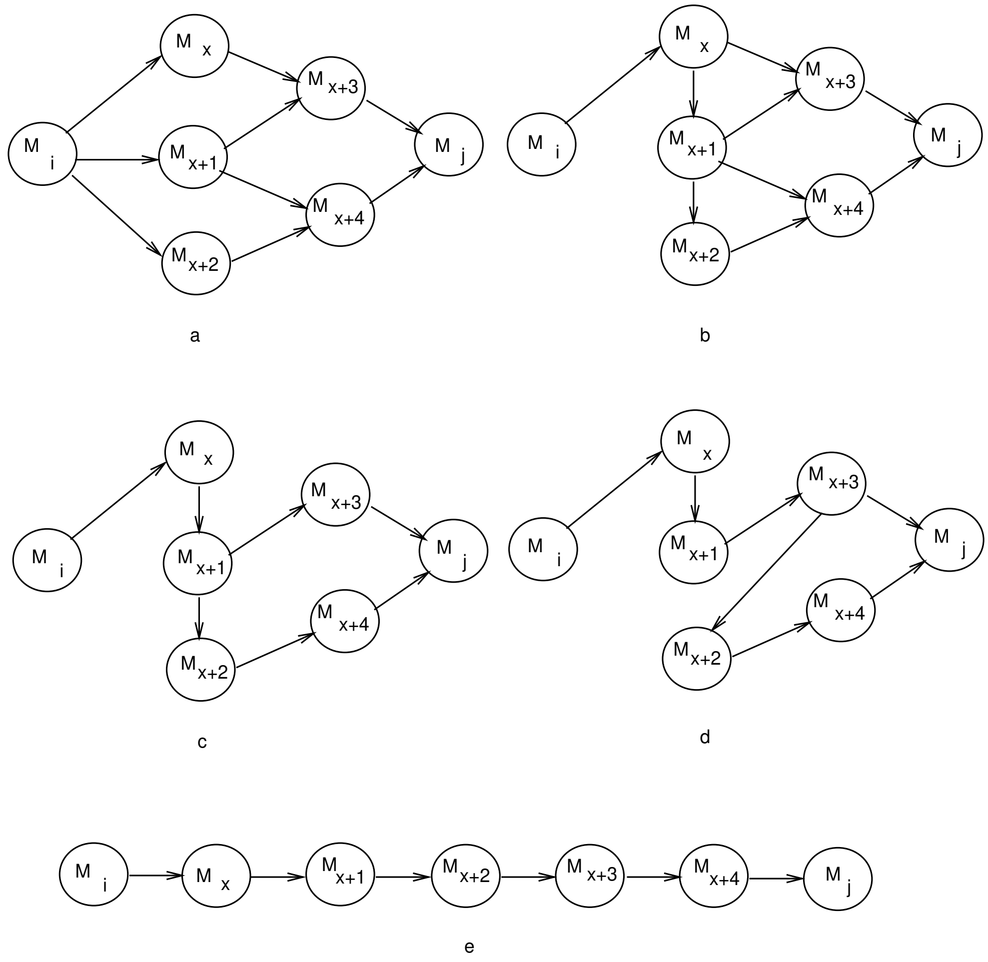

Figure 13: Example for general graph conversion

6.2.4 Finding a Partition

Finally, we present the last step of the mapping algorithm. As defined in Definition 6.1, we are trying to find out a partition such that the load for every SPU is less or equal to $w = S_1 * N_{c1} * T_{in}$.

Suppose after *EPG* construction, node coalescence, and graph conversion, there are N'_v nodes left in the chain structured graph. We denote the nodes in the graph as M'_1, M'_2, ..., $M'_{N'_v}$ with weights as w'_1, w'_2, ..., $w'_{N'_v}$ respectively. The communication cost from M'_i to M'_{i+1} is denoted as c'_i. If a sub-chain from M'_i to M'_j is assigned to a processor, then the load for the processor is given by:

$$L_{i,j} = c'_{i-1} + c'_j + \sum_{k=i}^{j} w'_k$$

The following procedure partitions the chain structured graph and gives the smallest number of processors sustaining the partitioning.

1. *Let i= 1, j = 1, p =1.*
2. *While* $L_{i,j} \leq w$,

   $j = j + 1$, *if* $j = m'$, *then return (p)*

   Repeat
3. *Let i = j,*

   $p \leftarrow p + 1$
4. *Goto 2.*

7 Simulation study

This section presents simulation results on the effectiveness of our parallel DSP architectures for stream flow programming. We are interested in obtaining the relationship among the grain-size, scheduling capability, and the utilization of processors.

A Petri net is constructed for the purpose of the simulation. As is shown in Figure 14, the Petri net is composed of two transitions and two token queues. Transition t_0, which represents *SPU*, consumes a fire token from token queue q_0. The time spent in t_0 is determined by the *weight* attribute of the token being processed. After a fire token is consumed and processed, a *done token* is produced and put into done token queue: q_1. Transition t_1, which represents *SU*, consumes a done token from q_1. The delay of t_1 is used to represent the scheduling processing capability of *SU*. When a done token is consumed by t_1, a C program is invoked to emulate the mechanism of the scheduling algorithm.

In order to collect the simulation data about the utilization of the SPU, *idle* token is introduced, and the queue q_2 is created to accommodate the *idle* token. An idle token bears the attribute of total idle time of the SPU. A fire token and an idle token are consumed together by t_0. When t_0 is activated, the idle time between this activation and the completion of last fire token is computed and added to the total idle time. The utilization is given by the following equation:

$$utilization = \frac{simulation\ time - idle\ time}{simulation\ time}$$

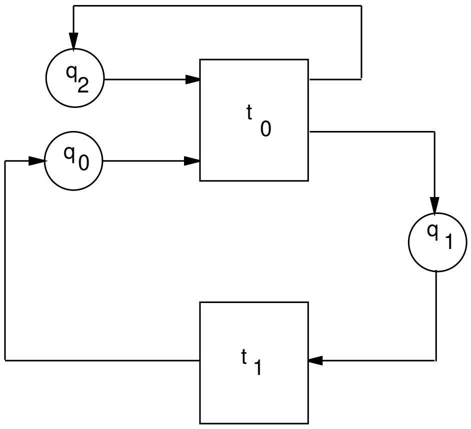

Figure 14: Petri-Net for simulation

Spectrum analysis, a typical DSP application, is used in our simulation study. The weight of the primitive DSP operations are shown in Table 3.

Table 3: Weight of primitive DSP operations (micro-seconds)

CBS	FIR	FFT	WEGT	DET	INT
5.8	60	1960	5.5	4.8	4.2

The utilization of SPU for the spectrum analysis stream flow graph with different grain-size are shown in Figure 15. The horizontal axis is the schedule time for processing a done token. One observation is that the utilization of SPU is decreased when the schedule time becomes larger. This is because when SU needs longer time to process a done token, SPU is starved. Therefore,a better utilization of SPU is achieved by faster SU.

Another observation is that when the grain-size decreases, utilization decreases with the same scheduling processing capability. This is because when the grain-size becomes smaller, the scheduling unit becomes the bottleneck. In this case, done tokens accumulated in the done token queue, q_1, while the fire tokens were "eaten up" quickly. The schedule time is of no importance to utilization when the grain size is larger than schedule time. However, the utilization starts to decrease when the schedule time is comparable to the grain size. The utilization decreases sharply when the grain-size is smaller than schedule time. This suggests that *SU* become the performance bottleneck.

Figure 16 shows the relationship between the throughput and the schedule-time. The throughput is calculated with respect to the time interval of two batch processed

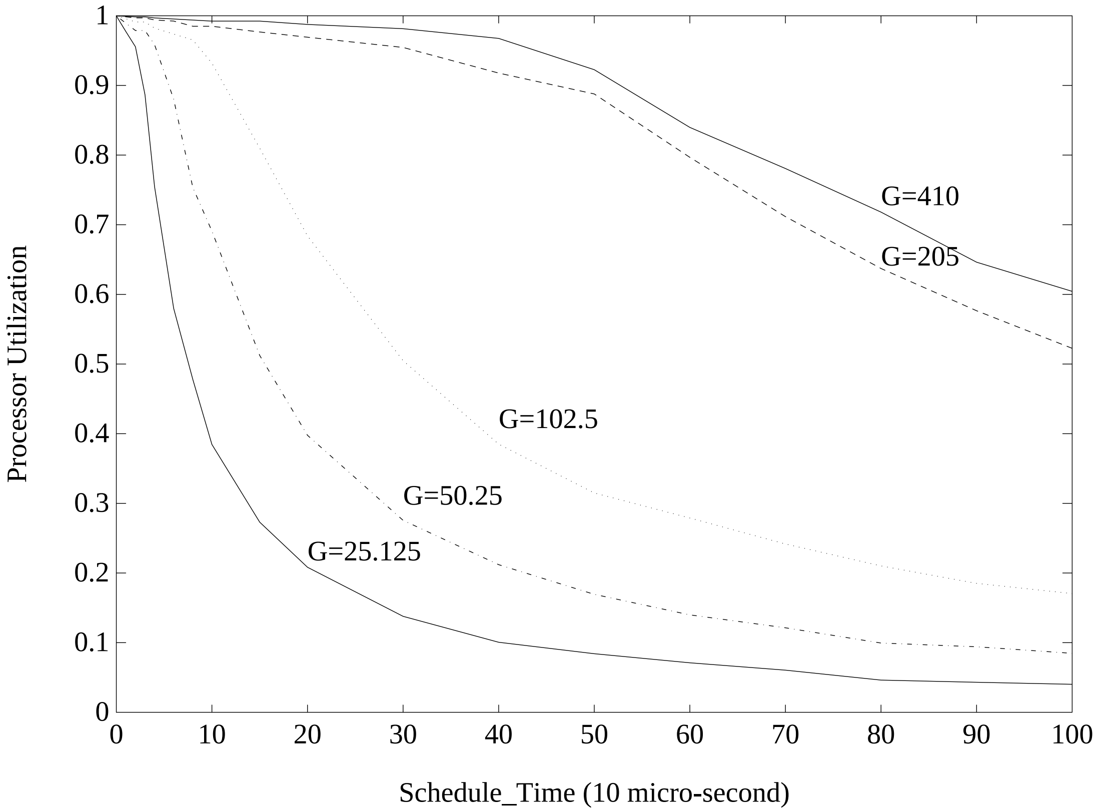

Figure 15: The utilization and the schedule time with different grain-size

spectrum data. The throughput is the maximum data samples that could be supported by the signal processing unit per second. An obvious observation is that the throughput is high (to the extent of SPU) when the schedule time is small. There is almost no difference in throughput when the schedule time is not larger than 40 μs. However, the utilization of the processor showed some differences. The utilization difference is caused by the initialization of the system when data samples start to arrive at the first node in our simulation. The total idle time mainly comes from this initialization. The throughput, however, is obtained from the time interval between two processed data group when SPU is working in full scale.

The figure also shows that larger grain-size gives better throughput. That is true for two reasons: one is that the processing of batch data samples saves the setup time in SPU, the other is that the scheduling processing requirement is low. However, it is dangerous to conclude that we can use large grain-size stream flow graph without limitation, because increasing the grain size needs larger buffers and results in long latency. Real-time signal processing systems do not tolerate storing data tokens and producing output data in a long interval, even though higher data sample rate is supported.

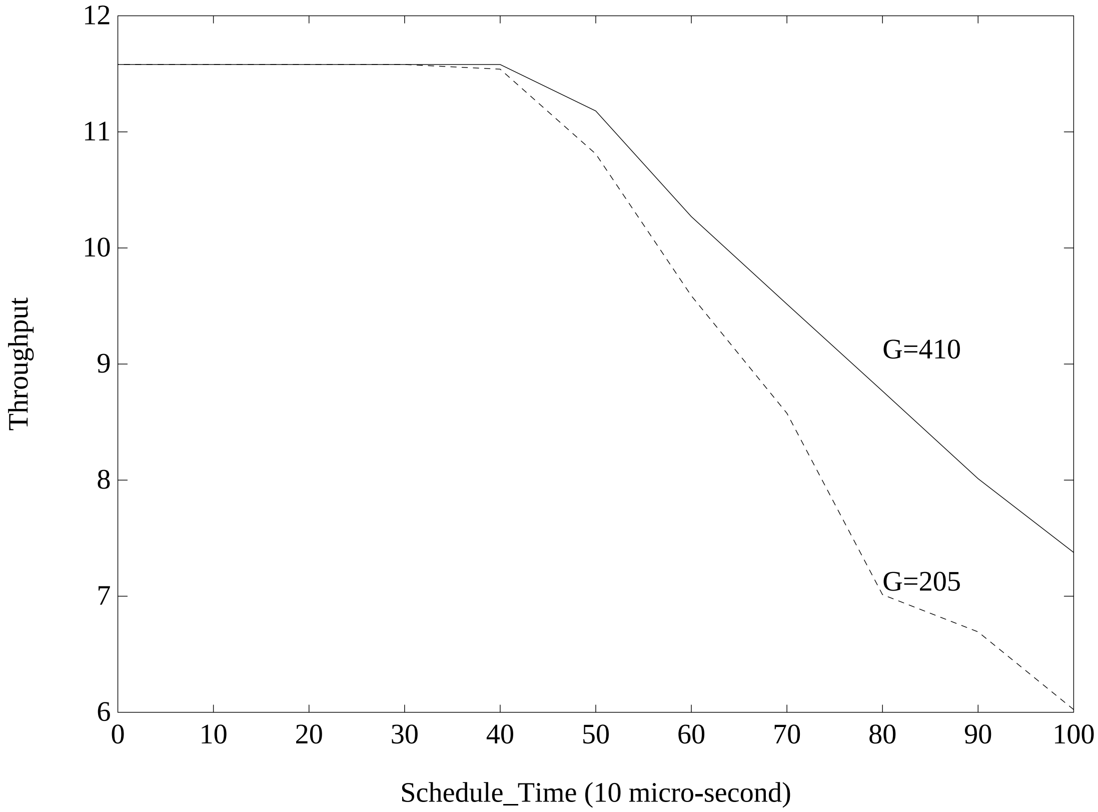

Figure 16: The throughput of the system

8 Related work

We compare our work in two research topics. One is parallel architectures for general purpose DSP computation including multi-threaded architectures for data flow programming, the other is mapping data flow programs.

One related work is the parallel architectures for *SDF* — MOMA[12]. MOMA is a statically scheduled parallel DSP architecture, which employs a shared data memory and is used for implementing DSP applications captured by *SDF*. Our architecture will not only allow DSP applications with conditionals and data dependent computations to be implemented, but also degrade gracefully when a processor breaks down. Another difference is that the memories are distributed in our architecture.

Another architecture we will compare is the AT&T Enhanced Modular Signal Processor (*EMSP*). Like our architecture, it is a large grain, dynamically scheduled architecture. However, *EMSP* does not have compile time mapping of DSP applications. *EMSP* assigns tasks to processors at run time by a separate functional unit, called scheduler, that monitors both the state of the program graph to determine when nodes are ready to execute, and the processors to determine when they are ready to accept new task assignment. One notable aspect of *EMSP* is that there is only one single scheduler, and there is no obvious way to add a second one. Since *EMSP* can be expanded by adding more memory and processors, there is some danger that the scheduler becomes the performance bottlenecks. Whereas, *SU* in our architecture can be scaled with distributed scheduling mechanism to avoid performance bottleneck.

Finally, a class of related works in parallel architecture is *multi-threaded architectures* for data flow programming [16] [6] [15] [7] [10]. The concept of multi-threaded architectures is to directly support multiple instruction threads at the processor level. A thread is a sequence of von Neumann instructions. A thread may be in any of the following states: waiting for operands, enabled, activated, and suspended. A thread can be enabled for execution when some of the required operands are available. It will be be suspended if an operand fetching is failed. Then another thread may take the processing unit without causing overhead of context switching. A suspended thread may be activated again once its operands are available again. Multi-threaded architecture has the potential to keep the processing unit usefully busy by rapid switching between threads on long latency operations, such as fetching operands from remote memory.

Our architecture is similar to multi-threaded architecture in that multiple node programs (threads) are enabled and ready for execution. In multi-threaded architectures, multiple register sets, program counters, and interleaved execution mechanism have to be provided. However, in our system, there is no such need for multiple active threads to support inside *SPU*, since when a node is enabled, operands and storage space are guaranteed to be available. Once a node is activated, it will not be suspended due to unavailable data. Making the DSP processor usefully busy is guaranteed as long as there are sufficient available enabled nodes. Commercially available DSP processors with only one program counter and one register set can be used.

Earlier works on partitioning of chain structured program modules for linear processor array are found in [5] [11]. They were applicable to chain-structured programs. Unfortunately, stream flow graphs do not come nicely with nodes connected in a chain. In this paper, we proposed a systematic method to map a stream flow graph onto linear processor array through graph transformations, including construction of *EPG*, node coalescence and graph conversion. This mapping algorithm has substantially extended the earlier works to a wider signal flow graphs. It can be used for static assignment for fully static schedules as well.

9 Summary

In this paper, we presented a dynamically scheduled scalable parallel DSP architecture. A distributed scheduling mechanism is proposed to avoid potential performance bottlenecks when the configuration of the parallel architecture is large. We developed a systematic mapping algorithm to map a stream flow graph onto a parallel DSP architecture. The mapping algorithm substantially extended earlier works on partitioning. We proposed a dynamic scheduling algorithm that will only schedule a node for execution when both input data and output storage space are available. Such scheduling algorithm will allow buffer sizes be determined at compile time. Further, our simulation study revealed the relationship among the grain size, the processor utilization and the scheduling capability. The simulation study results will benefit parallel computer designers to determine the resource requirements in multiprocessor architecture design involving dynamic scheduling.

Future work involves the prototyping of the proposed parallel DSP architecture. As a first step, a microprocessor will be used as the scheduling unit. Application-specific programmable hardware implementation of the scheduling unit can also be considered. Another research direction is to implement stream flow programming environment for DSP and communication system design targeting the proposed architecture. We plan to implement such environment on top of an existing environment — Ptolemy[4] [3]. Our compilation techniques for dynamically scheduled architecture are planned to be introduced into Ptolemy within the stream flow programming environment.

References

[1] Jonathan Allen. Computer architectures for digital signal processing. *Proceedings of IEEE*, 73(5):852–873, May 1985.

[2] H. Alrutz. Implementation of a multi-pulse coder on a single chip floating-point signal processor. In ICASSP'86, Tokyo, Japan, pages 2367-2370. 1986.

[3] J. Buck, S. Ha, E. Lee, and D. Messerschmitt. Ptolemy: A mixed-paradigm simulation/prototyping platform. In *Proc. of Speech Tech 1991*, April 23-25 1991.

[4] J. Buck, S. Ha, E. Lee, and D. Messerschmitt. Ptolemy: A platform for heterogeneous simulation and prototyping. In *Proc. of European Simulation Conference*, June 17 -19 1991.

[5] S. Bokhari. Partitioning problems in parallel pipelined and distributed computing. *IEEE Transactions on Computers*, 37(1), Jan. 1988.

[6] J. B. Dennis. Evolution of the static dataflow architecture. In *Advanced Topics in Dataflow Computing*. Prentice-Hall, 1991.

[7] G. R. Gao. A flexible architecture model for hybrid dataflow and control-flow evaluation. In Jean-Luc Gaudiot and Lubomir Bic, editors, *Advanced Topics in Data-Flow Computing*, chapter 12, pages 327–346. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[8] G. Gao, R. Govindarajan, and Prakash Panangaden. Construction rules for well-behaved stream programs. ACAPS Technical Memo 26, School of Computer Science, McGill University, Montreal, Que., April 1992.

[9] G. Gao, R. Govindarajan, and Prakash Panangaden. Well-behaved dataflow for DSP computation. In *ICASSP-92, 1992 International Conference on Acoustics, Speech, and Signal Processing*, March 1992.

[10] G. R. Gao, H. H. J. Hum, and J. M. Monti. Towards an efficient hybrid dataflow architecture model. In *Proceedings of PARLE '91, Parallel Architectures and Languages Europe*, Eindhoven, the Netherlands, June 1991.

[11] M.A. Iqbal. Efficient algorithms for partitioning problems. In *Proceedings of the 1990 International Conference on Parallel Processing*, Aug. 1990.

[12] E.A. Lee and J. Bier. Architectures for statically scheduled dataflow. *Journal of Parallel and Distributed Computing*, 10:333–348, 1990.

[13] E.A. Lee and D. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, 36(1):24–35, Jan. 1987.

[14] Motorola. "DSP96002 User's Manual," 1990.

[15] G. M. Papadopoulos and D. E. Culler. Monsoon: An explicit token-store architecture. In *Proceedings of the Seventeenth Annual International Symposium of Computer Architecture, Seattle, WA*, pages 82–91, 1990.

[16] Burton Smith. The architecture of HEP. In J. S. Kowalik, editor, *Parallel MIMD Computation: HEP Supercomputer and its Application*, pages 41–55. The MIT Press, 1985.

A Appendix

Theorem 6.1 *Suppose a stream flow graph has a partitioning on n processors of weight w, then a similar partition or partition of less weight exists even if we merge node i and $i+1$ provided the communication cost between node i and node $i+1$ is equal to or greater than the weight of node i plus the sum of communication cost of all other arcs of node i.*

Proof: When node i and node $i+1$ reside in one processor, the proof is trivial. We assume that node i and node $i+1$ reside in two separate processor, P_i and P_{i+1}.

Suppose node i has $j+1$ arcs. The communication cost of are denoted as $(c_{i,1}, c_{i,2}, ..., c_{i,j})$ and the arc connecting node i and node $i+1$ is denoted as c_i.

The load for P_i is given by

$$w_{P_i} = w_i + c_i + w_{r_i},$$

where w_{r_i} is the sum of all other node weights in the partition for P_i plus the communication costs to the outside of the partition.

The load for P_{i+1} is given by

$$w_{P_{i+1}} = w_{i+1} + c_i + w_{r_{i+1}},$$

where $w_{r_{i+1}}$ is the sum of all other node weights in the partition for P_{i+1} plus the communication costs to the outside of the partition.

Now we merge node i and node $i+1$ and move node i to P_{i+1}. We calculate the loads for the processors after the merge assuming the worst case where all arcs of node i except c_i reside in P_i.

$$w'_{P_i} = \sum_{l=1}^{j} c_{i,l} + w_{r_i}$$

$$w'_{P_{i+1}} = w_i + w_{i+1} + \sum_{l=1}^{j} c_{i,l} + w_{r_{i+1}}$$

Since

$$\sum_{l=1}^{j} c_{i,l} + w_i \le C_i,$$

so,

$$w'_{P_i} < c_i + w_{r_i} < w_i + c_i + w_{r_i} = w_{P_i},$$

and

$$w'_{P_{i+1}} \le c_i + w_{i+1} + w_{r_{i+1}} = w_{P_{i+1}}.$$