Abstract

Scheduling Techniques for Synchronous and Multidimensional Synchronous Dataflow

by

Praveen Kumar Murthy

Doctor of Philosophy in Electrical Engineering and Computer Science

University of California at Berkeley

Professor Edward A. Lee, Chair

In this thesis, we present various scheduling techniques for programs expressed as Synchronous Dataflow (SDF) and Multidimensional Synchronous dataflow graphs. Synchronous dataflow has proven efficient as a specification model for block-diagram based programming environments for signal processing. Two key reasons for its popularity are that a) static schedules can be constructed at compile time, thus eliminating the overhead due to dynamic scheduling, and b) it models multirate signal processing application very naturally and intuitively. The first property is particularly important in environments where code-synthesis for embedded processors is desirable, since embedded signal processing applications have rigid throughput requirements in order to meet hard-real-time constraints. Multidimensional Synchronous Dataflow (MDSDF) is an extension of SDF to multiple dimensions; in this model, applications such as image and video signal processing, which operate on multidimensional signal spaces, are more naturally modeled and specified than in SDF.

The amount of on-chip memory available on embedded processors is often severely limited. Adding off-chip memory is usually not an option because it entails a speed penalty, it increases overall system cost and size, and increases power requirements. Thus, when generating software implementations from SDF specifications for a uniprocessor, the scheduling problem of minimizing code size and buffer memory size becomes crucial. If the scheduling is not done carefully, the blowup in the size of the implementation can preclude implementation on the processor. Previous techniques have focused on scheduling to minimize code size. However, this neglected the buffer memory usage of the schedule; in this thesis, we develop techniques that jointly minimize for code size and buffer-memory size of the software implementation. We give three polynomial-time algorithms: a dynamic programming algorithm that is used as a post-optimization step, and two heuristics that use different approaches to constructing these schedules. The general problem is shown to be NP-complete, thus justifying the use of heuristics. An extensive experimental study is given to show the efficacy of all these techniques.

We extend these scheduling results to MDSDF specifications. We then tackle a problem of a different type. A multidimensional signal can be sampled in many different ways. A straightforward extension of one-dimensional sampling results in the so-called rectangular sampling structure, where the samples lie on a rectangular grid. However, a more general sampling structure is a geometrical lattice; sampling lattices that are not rectangular can have many advantages in certain applications. For example, a signal sampled on a non-rectangular lattice can have a lower sampling density than one sampled on an equivalent rectangular lattice. For real-time processing of multidimensional signals, a lower sampling density means fewer samples to process in a given time interval. The standard MDSDF model suffers from the inability to model multidimensional systems sampled on arbitrary sampling lattices; hence, we give an extension of MDSDF that is capable of modeling such systems. The model we give preserves the property of static, compile-time schedulability. However, constructing such schedules requires the solution to some challenging problems. In particular, we show that an augmented set of balance equations have to be solved simultaneously in the extended model. The additional equations are quite different from the usual balance equations in SDF and MDSDF; they involve computing so-called “integer volumes” of parallelepipeds. This computation turns out to be an interesting number-theoretic problem, and we present several approaches for solving it. Finally, we present a practical example of a video sampling structure conversion system to show the usefulness of the generalized MDSDF model.

Edward A. Lee, Thesis Committee Chairman

Scheduling Techniques for Synchronous and Multidimensional Synchronous Dataflow

by

Praveen Kumar Murthy

Scheduling Techniques for Synchronous and Multidimensional Synchronous Dataflow

by
Praveen Kumar Murthy

B. S. E. E, Georgia Institute of Technology, 1989

M. S. (University of California at Berkeley), 1993

A dissertation submitted in partial satisfaction of the requirements for the degree of

Doctor of Philosophy

in

Engineering-

Electrical Engineering and Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor Edward A. Lee, Chair

Professor Robert K. Brayton

Professor Donald A. Glaser

1996

The dissertation of Praveen Kumar Murthy is approved:

Chair Date

Date

Date

University of California at Berkeley

1996

To my parents

Contents

List of Figures

Chapter 1:

Chapter 2:

Chapter 3:

Chapter 4:

Chapter 5:

Chapter 6:

Acknowledgments

Graduate student life in UC Berkeley has been a rich, rewarding experience and I will sorely miss it once I leave this illustrious university. I have been fortunate enough, over the years, to meet many distinguished faculty members and intelligent colleagues, and I have learnt much in the process. It is a pleasure to acknowledge those who have had an impact on me, both during my time here, and in times past.

I am deeply grateful to Professor Edward A. Lee for being my advisor for this thesis. I appreciate his patience as I kicked around looking for a thesis topic, and I appreciate the feedback I have received over the years, particularly about the “big picture”. I hope some of Edward’s ability to dabble in a variety of research areas has also rubbed off on me. I have also learned much from his insistence on good technical writing!

I thank professors Robert K. Brayton and Donald A. Glaser for serving on my thesis and qualifying examination committee, and professor Martin Vetterli for serving on my qualifying examination committee. I have also greatly enjoyed attending classes here taught by professors Richard Karp, Jean Walrand, and Donald Glaser. Professor Messerschmitt’s many humorous tales at the post-DSP-seminar wine-and-cheese parties will be remembered fondly.

Among colleagues, Shuvra Bhattacharyya has been like a second advisor to me, both when he was a senior student in the group, and later as a researcher at Hitachi Labs in San Jose. In addition to a fruitful collaboration with him that led to the book [Bhat96a], he has been a close friend and guide on all matters. I have enjoyed working with him, and learnt a lot from his meticulousness and professional approach.

S. Sriram and I have been close friends ever since we started playing tennis while living at I-house. I have enjoyed the many conversations we have had in the office, home, on the tennis courts, on all matters from the nature of research to the perfect tennis racket, and the many joint activities undertaken like sailing, skiing, and attempts at wind-surfing. Another longtime friend here has been Amit Lal, my roommate for 4 years before he got married. I have since enjoyed the many dinners at his place after marriage, cooked by him and his wife, Garima.

Amit Marathe, Ramesh Gopalan, Sridhar Rajagopalan, and Sachin Adarkar are all erstwhile housemates (real and virtual) from 2023a Parker street. Some impressions I will cherish: Amit "Marroth" Marathe, who argued persuasively for the arranged marriage system, Amit "The Bunt" Lal, who generally argued and had to go off to Microlab, Ramesh Gopalan, who taught me that when you are feeling pretty low and depressed, and feel that nobody loves you but your mother, and she could be jivin' too[1], breaking out into an English accent and saying random things is a good way to cheer yourself (and others) up, and Sridhar ("Sreehaar"), who proved by demonstration that he knew all about zero-knowledge proofs[2], and also bequeathed his espresso machine to me. Sachin Adarkar, Amit Lal, and I had many interesting philosophical discussions that went nowhere, although Sachin ensured that Magic Johnson should never be subjected to such an indignity. From Sachin, I also developed an appreciation and love for American roots music, like Robert Johnson, Ray Charles, James Brown, and Howlin' Wolf.

Gaku Tsuda has been a good friend and has told me many interesting things about his research in anthropology on the subject of Japanese immigrants in Brazil. I have enjoyed many discussions with Kumud Sanwal on photography. I am grateful to the ASUC Art Studio for maintaining an excellent darkroom; this has enabled me to develop a deep interest in photography and take refuge in "art" when "science" seems to become overwhelming.

In Cory hall, I will miss the friendly chats with Shankar Narayanaswamy, Louis Yun, Richard Han, Rajeev Murgai, Prashant Phatak, Anu Bhat, and Sriram "Praveen yappaddi irke? "Alaha irke" ha ha ha"[3] Krishnan. In my group, I have had many useful discussions with Phil Lapsley, Alan Kamas, José Pino, John Reekie, Alain Girault, Stephen Edwards, Wan-teh Chang, Mike Williamson, and Brian Evans. Karim Patrick Khiar, who visited the group one summer from Thomson CSF in Paris, proved to be an invaluable guide and host when I was in Paris.

[1] Phrase due to B. B. King.

[2] CS theory inside joke.

[3] Sanketi-Tamil inside joke.

Over the past year, I have become good friends with Gitanjali Swamy and Sanjay Sarma. I have enjoyed having Gitanjali's company (and many observations on married life) as a housemate.

I thank Mary Stewart, and Carol Sitea for taking care of all group-related administrative matters, and Christopher Hylands for answering many pesky UNIX-related questions. I am also thankful to Heather Levien in the Graduate office for keeping Sproul hall at bay.

At my alma-mater, Georgia Tech, I am thankful to Professor Richard Shafer for his support while I was a graduate student there briefly, and for teaching many excellent classes on DSP. I enjoyed working with Professor Jim Dorsey on my senior project. Professor Vijay Madisetti gave me useful advice when I set out to Berkeley.

I thank my numerous relatives for their ego-boosting praise given to me over the years.

My sister, Ranjana, while taking absolutely no interest in CAD for DSP, has kept me abreast on all matters political, and made sure that I do not wander into any territory that could be remotely perceived as being right wing (although I am sure I have frustrated her many a time. But really, I am a liberal). She has also provided a pleasant refuge in the south bay. My parents have been a steady source of inspiration, advice, and support, and I am indebted to them for it. Finally, I am grateful to my wife Jyothi for the love and friendship over the past few months, and providing the impetus to finish this thesis!

1

Introduction

This thesis is concerned with certain aspects of static scheduling of programs for digital signal processing (DSP) applications specified as dataflow graphs. In particular, synchronous dataflow (SDF), and multidimensional synchronous dataflow (MDSDF) are two subsets of dataflow that are well suited for programming signal processing systems. A key property that these two models have is that all scheduling decisions can be made at compile time, rather than at run time. The thesis has two main parts: the first is a study of the problem of minimizing the amount of code size and buffer size in the target program when synthesizing code from a SDF or MDSDF specification. The second is a generalization of MDSDF to permit modeling of multidimensional multirate signal processing systems sampled on arbitrary sampling lattices.

Over the past few years, there has been increasing interest in dataflow models of computation for DSP because of the proliferation of block diagram programming environments for specifying and rapidly prototyping DSP systems. Dataflow is a very natural abstraction for a block-diagram language, and many subsets of dataflow have attractive mathematical properties that make them useful as the basis for these block-diagram programming environments.

Visual languages have always been attractive in the engineering community, especially in computer aided design, because engineers most often conceptualize their systems in terms of hierarchical block diagrams or flowcharts. The 1980s witnessed the acceptance in industry of logic-synthesis tools, in which circuits are usually described graphically by block diagrams, and one expects the trend to continue in the evolving field of high-level synthesis and rapid prototyping. The advent of high speed workstations with advanced graphics capabilities will make possible increasingly sophisticated forms of visualizing complex systems.

Another trend driving this research is the advent of high-performance DSP architectures that are capable of executing computationally intensive DSP algorithms in real-time. These architectures are typically used in embedded systems like digital cellular telephones, voiceband data modems, speech recognition systems, video compression and decompression, and music synthesizers, where constraints on speed, memory usage, power consumption, and size are fairly stringent. The time-to-market of these products is also a key issue in hotly competitive areas like consumer electronics.

Traditionally, DSPs have been programmed in assembly language by experienced programmers, a tedious and error-prone process at best. However, because of the constraints mentioned above, it is becoming more desirable to provide tools that make programming easier so that designs can be conceptualized, optimized, and made into products more quickly. The reliance on assembly language has been for performance reasons, and one does not expect that automated techniques will be as good as hand-optimized code in general. But we do expect that as DSPs become more powerful, and algorithms get more complicated, it will not only become reasonable to sacrifice some performance in favor of a quicker and cleaner design, but it will become imperative to do so, since it will become infeasible to hand-code large systems in assembly language due to the high complexity, just as large circuits today can no longer be hand-optimized.

The mass popularity and acceptance of the Internet is also driving the need for new, throughput-hungry applications such as video-on-demand, video tele-conferencing, and image and video signal processing applications in general. The real-time performance requirements of high quality video are orders of magnitude higher than those required for high quality audio. Supporting these applications will certainly require multiprocessor DSP architectures, and it is well known that programming parallel architectures is a difficult problem. A combination of having to program a parallel machine and having to program it in assembly language is an even more unattractive one!

Block diagram languages for specifying systems have several advantages, in addition to those mentioned already. As software systems become increasingly complex, it has been proving more difficult to realize one of the biggest advantages that software supposedly has over dedicated hardware implementations: re-usability. Quite simply, writing arbitrary software does not lead to software that can be re-used or maintained easily. High-level organization is often required to ensure that software reusability is a reality. A major reason for the warm reception that the Java programming language has gotten from Web developers and software developers in general is precisely because of its potential for code reuse: both the concept of applets, which are stand-alone applications that run inside a Web browser, and the concept of the Java virtual machine that abstracts away platform-specific dependencies, make it possible for software engineers to develop code only once, without having to worry about porting issues.

Block diagram environments encourage software reuse because the blocks in the library are modular, reusable components. Block diagrams, combined with an appropriate MoC, enforce the idea that specifications should not *overspecify* the system under consideration; this allows a specification to be re-targeted to a different architecture much more easily. A common example of overspecification in high-level, imperative languages is the total ordering of all statements, making it difficult for a compiler or hardware synthesis tool to extract parallelism from the specification. Since writing parallel programs is a difficult problem in general, it is essential that specification languages make it easier for automated tools to do the partitioning, scheduling, and load balancing.

Finally, block diagram languages are easy to use. Experience has shown that there is often a great resistance to learning a new programming language; people will argue strenuously that everything can be done well by the language they are currently comfortable with. However, ease-of-use and a clear, concise syntax and semantics will allow a language to become used and accepted much more easily, and block diagram languages certainly seem to have this property, as they are very intuitive. It has been remarked that the acceptance of a block-diagram language occurs almost by stealth since users do not even realize that they are learning a new language!

By a *model of computation* (MoC), we mean the semantics of the interaction between the blocks (or modules) in the system. A program in a block diagram language is specified by interconnecting blocks drawn from a library of blocks. A graph is a natural, mathematical structure for describing programs written this way. The interconnections usually specify communication channels through which data is sent and received. The interconnection may also specify precedence constraints in some MoCs. Thus, we can loosely speak of a buffer on an interconnection where data is actually stored while being exchanged between blocks.

In the rest of this chapter, we survey a number of MoCs and programming languages that are relevant to this thesis, and discuss their formal properties. In the next three sections, we discuss a few concepts that are useful when describing mathematical properties of MoCs.

1.1 Reactive Systems

Signal processing applications fall into the class of so-called *reactive* systems—these are systems that must continuously respond to an environment and produce outputs with certain real-time constraints. A key point is that such systems never terminate. For example, an algorithm that digitizes speech for transmission over a telephone line has to run continuously and is never turned off. The amount of data processed by such systems can be vast; just 10 minutes of audio at the CD rate of 44.1 khz entails processing over 26 million input samples. In contrast, *transformational* systems, like conventional programs for computing, operate on data presented to them in the beginning, compute, and halt after producing results.

Hence, we need to allow infinite sequences of block executions in models designed to represent reactive systems. Several problems must be dealt with when infinite sequences of actor executions are allowed. Firstly, there is the potential for buffers to become unbounded. Secondly, the system may deadlock, indicating that the system cannot execute infinitely. Even if a part of the system deadlocks, this is still an error in most practical cases since in a reactive system, all specified operations are assumed to operate infinitely. Lastly, some mechanism is needed to actually sequence the actor executions in accordance with the precedence constraints imposed by the graph. The simplest method is to have a dynamic scheduler that picks any actor that is firable (meaning it has enough input tokens on all of its incoming edges) and executes it. However, the overhead associated with this runtime decision making can be significant, especially if the graph is of fine granularity (meaning that there are many actors that have a small execution time). A better solution is to construct finite schedules and execute them inside an infinite loop. This approach is called static scheduling, and is only possible if periodic schedules exist for the program.

1.2 Computability

As already mentioned, a model of computation refers to the semantics of the interaction between modules in the system. A key concept that is useful for classifying different models of computations is the concept of computability [Epst89]. This concept refers to the class of functions that may be expressed by interconnecting the modules in the system. The terms interconnection and module are used in a broad sense here. For example, a module might be an actor in a dataflow graph, and the interconnection the communication channel through which data tokens are exchanged, or the module can be a program statement, and the interconnection between two program statements the set of memory locations accessed by each.

It can be shown that it is sufficient to consider functions from the integers to integers—functions that have a different range or domain can always be expressed as functions from the integers to integers. The number of functions from the integers to integers that can be expressed in the MoC is usually called the *expressive power* of the MoC[1]. An expression of a program in an MoC is said to be of finite description if the number of modules and interconnections used for expressing the program is finite.

A function is called computable if there is a finite length procedure for computing it in a finite amount of time; this is an intuitive definition of what we understand to be an "algorithm". In the theory of computability, it has been found that all attempts at formalizing the notion of computability lead to the same class of computable functions. Thus, if a function can be computed on a Turing machine, it can be "computed" in any other formal system as well, for example, in the λ-calculus, Post production systems and so on [Epst89]. To be precise, it can be proven that a function is computable by a Turing machine

[1] Expressive power is also used sometimes to mean the degree of succinctness with which programs can be expressed in the MoC. However, we do not use the term this way in the discussion above.

if and only if it is a *partially recursive* function. The partially recursive functions arise in a formalism known as recursive function theory; this is a purely arithmetic formalism and is not based on a machine-like abstraction like the Turing machine. Classes of functions in this theory are built up by giving a set initial functions (the everywhere zero function $n(x) = 0$, the successor function $s(x) = x + 1$, and the projection functions $U_i(x_1, \ldots, x_n) = x_i$), and some allowed operations (functional composition, recursive definitions), and taking the class to be the smallest one closed under such operations. In this theory, the most general class of functions are the partially recursive functions. Hence, a function computed by a Turing machine is often called a partially recursive function, and the following alternative definition of a partial recursive function can be stated in terms of a machine model of computation (rather than the purely arithmetic one used in recursive function theory): it is a function computed by a Turing machine that may or may not halt on a given input. In contrast, the total recursive functions (that are also defined arithmetically in recursive function theory) are functions computed by Turing machines that always halt.

The Church-Turing hypothesis states that the class of computable functions are precisely those computable by Turing machines (or definable in the λ-calculus, or the class of partially recursive functions, and so on.) Of course, we cannot prove this hypothesis since the notion of computability is not a precise one, and is based on an intuitive belief of what a mechanical procedure should be for computing functions. However, the fact that all attempts at formalizing the notion of computability have led to the same class of functions gives a compelling reason to believe the Church-Turing hypothesis. The book [Epst89] provides a good introduction to the theory of computability, and includes several philosophical and mathematical viewpoints regarding the Church-Turing hypothesis.

The number of functions from integers to integers is more than the number of computable functions. This is because the set of computable functions is countably infinite while the set of functions from the integers to integers is uncountably infinite. The surprising thing is that there are many practical functions of interest that are uncomputable. An example is the *halting problem*: this is a function that takes as input a program, and returns 1 if the program halts in a finite number of steps and 0 otherwise. Since programs can be encoded as integers, this is also a function from the integers to integers, and one would like to know whether this function is computable. It turns out that it is not computable (that is, it is *undecidable*): there is no one algorithm that can decide in a finite amount of time whether or not the program given as the input terminates. Other undecidable problems for Turing machines include questions such as whether a program will execute in bounded memory.

If the class of functions that can be computed by an MoC coincides with the class of functions computed by Turing machines, the MoC is called *Turing complete*. Hence, problems that are undecidable for Turing machines will also be undecidable for the MoC.

There is, in general, a trade-off between the expressive power and decision power of the MoC, where the former refers to generality of the class of programs that may be specified, while the latter refers to properties about programs that may be mathematically proven. The more the expressive power is, the less the decision power is, and at the extreme end of the spectrum are Turing complete models since they express the most general class of programs but have the least decision power since many general questions about these programs are undecidable. At the other end of the spectrum are models that can only express a highly restricted class of programs, like marked graphs, but have high decision power since many questions are decidable and can even be answered efficiently (that is, in polynomial time).

1.3 Semantics

In order to prove non-trivial properties about programs, the MoC should have well-defined semantics. In *denotational semantics*, the meaning of a program is defined as the function the program computes (or denotes) [Stoy77][Alli86]. In this approach, the range and domain of the functions are usually partial orders. A partial order (p.o) is simply a set of values, with an ordering ($\leq$) relation on the elements of the set. There need not be an ordering relationship between every two members of the set; hence, it is a *partial* order. If the program is a straight-line program, with no loops or recursion, the problem of determining the function that the program denotes is fairly simple: it is simply the composition of the functions represented in the individual steps. The problem of determining the function is more complicated for recursive, or iterative programs.

In a recursive or iterative program, one can conceptually think of data values generated during the execution of the program as a sequence of progressively better approximations to the final result. The notion of approximation is captured in a partial order in the following way. If a and b are two elements representing "information" (held by the data values) in some manner, then $a \le b$ in the p.o if the information in b is a superset of that in a. The notion of *continuity* for functions that operate on domains that are partial orders is the following: given a sequence of better approximations $a_1 \le a_2 \le a_3 \le \ldots$, the *upper bound* a of the sequence is an element that has all of the information in the a_i; in other words, $a_i \le a \quad \forall i$. The *least upper bound* a is an upper bound that satisfies $a \le b$ for any other upper bound b. A function f is *continuous* if the least upper bound of the sequence $f(a_1), f(a_2), \ldots$ is identical to f applied to the least upper bound of $a_1 \le a_2 \le a_3 \le \ldots$. Basically, there are no "surprises" at the limit: applying f one element at a time will give the same answer as applying f all at once. A fixed point is simply a value x such that $F(x) = x$ where F is some function. A fixed point represents "convergence" since applying the function again to the information does not generate different information. Since recursive or iterative programs can be thought of as "converging" to the final answer, the notions of fixed points and partial orders are useful for giving mathematical meanings to these programs. Moreover, a least fixed point is unique, which is why least fixed points are used to give meanings to programs since they represent convergence to unique final information. Note that convergence need not mean that the program halts: the "final" answer could in fact be some sort of infinite structure, like a real number. Denotational semantics is concerned with what happens in the limit, and if the limit happens to be a well-defined infinite structure, then hopefully any finite behavior of the program will simply be an approximation of the final infinite structure.

For instance, consider the following factorial program [Alli86]:

```
fact(n) = if (n=0) then 1 else n*fact(n-1)      (1.1)
```

In the definition above, we are using `fact` before we have really defined it. We would like to be sure that the program above really does compute the factorial function defined as the set of pairs $\{(0, 1), (1, 1), (2, 2), (3, 6), (4, 24), \ldots\}$. In other words, we would like to be able to prove that the program above *denotes* the factorial function. Consider the program `fact` written in Church’s lambda notation:

```
        F = λf. λx. if x=0 then 1 else x*f(x-1)        (1.2)
```

The λ in the above expression simply means that the symbol following it is a variable in the expression. Hence, F is a function (called a *functional* sometimes) of two variables: an argument f that is a function itself, and an argument x that f is applied to. For example, we can let $f = \text{succ}$, for the successor function, let $x = 7$, and evaluate F:

```
F succ 7 = if 7=0 then 1 else 7*succ(7-1)
```

The above clearly evaluates to 49. However, notice that applying F to succ gives the function

```
F succ = λx. if x=0 then 1 else x*succ(x-1) = λx. if x=0 then
1 else x^2
```

Clearly, this function is not equal to $\text{succ}(x)$; hence, it cannot be the function that the program in equation 1.1 denotes. What we really want is the function f that is a fixed-point of equation 1.2; that is, the function f such that $F\ f = f$. Certainly, applying F to the function $\text{factorial}(x) = x!$ results in a fixed point:

```
F factorial = λx. if x=0 then 1 else x*factorial(x-1) =
factorial
```

So factorial is a fixed point of F. However, recursive equations of the type in 1.2 can have many different fixed points. Indeed, for any fixed point f of F, f is defined unambiguously for all non-negative integers, but on the negative integers, the definition gives us $f(-1) = -1 \times f(-2)$ etc. We could make an arbitrary choice for $f(-1)$ and fix the values on all of the negative integers; for each arbitrary choice, we would get a different function that is a fixed-point of F. However, the least fixed point is unique; this is why the meaning of the program is taken to be the least fixed point solution to the recursive equation 1.2.

The existence of least fixed points is not usually guaranteed for arbitrary functions on arbitrary domains. However, least fixed points do exist if suitable restrictions are made on the domains and functions. For example, if the domain is a complete partial order, where “complete” means that every increasing chain $a_1 \leq a_2 \leq \ldots$ has a least upper bound, and functions operating on this domain are continuous as described before, then the Tarski fixed point theorem guarantees the existence of a least fixed point for any function. Moreover, Tarski's theorem is constructive: it gives us a procedure for finding this fixed point. If the domain is a metric space, and functions are continuous in the usual topological sense, then the Banach fixed point theorem guarantees the existence of fixed points (and again, with further suitable restrictions on functions, the theorem becomes constructive, rather than existential).

Finding the least fixed point using Tarski's theorem is straightforward. In the partial order, we include an element called "bottom"—this element represents complete absence of information; that is, the "undefined" state. "Bottom" is less than every other element in the partial order. If the elements of the partial order are functions, then "bottom" is the everywhere-undefined function. To find the fixed point of F, we simply keep applying F to "bottom" until we converge to the result; that is, until further application of F does not change the answer. So, if we apply F to bottom using equation 1.2, we get the new function f_0 that is undefined everywhere except at 0 where it has value 1. If we apply F to this function, that is, $F(F\ \text{bottom})$, we get the function f_1 undefined everywhere except at 0 and 1, where the values are 1 and 1. In this manner, we get a sequence of functions f_i where each function f_i agrees with the factorial function on the integers $\{0, \ldots, i\}$ and is undefined for all other integers. The least fixed point of this sequence of functions is the factorial function, and hence the *meaning* of the program is that it denotes the factorial function (note also that the function we get as the least fixed point has value bottom for all the negative integers). So, for any finite n, the function denoted by the program is a coarser approximation to the factorial function since the function only agrees with the factorial function on $\{0, \ldots, n\}$ and is undefined everywhere else.

1.4 Dataflow

A dataflow graph is a directed graph where the set of vertices, called actors, represent computations, and a set of directed edges between the actors represent communication channels implemented as first-in-first-out (FIFO) queues. Actors consume data values, called tokens, from their input edges and produce tokens on their output edges; hence, the edges also represent precedence constraints.

The application of dataflow principles to design computer architectures and programming languages was pioneered by Dennis in the 1970s [Denn75][Denn80]. Unlike Von Neumann architectures, where instructions are under the explicit control of a program counter, computations in a dataflow computer are driven by the availability of data. Hardware capability is provided for detecting when data is available, and for routing data tokens to the appropriate actor inputs.

Actors in a dataflow graph are required to be functional: that is, the output produced by a process is uniquely determined by its inputs (in case the reader objects that this precludes the actor from having state, an edge from an actor to itself can be used to carry state information in the form of an input). Dataflow languages and functional languages, like pure Lisp and Standard ML, belong to the class of *definitional* languages, in contrast to imperative languages like C or FORTRAN, which belong to the class of *operational* languages [Ambl92]. The difference can be succinctly stated in the following manner: in a definitional approach, the programmer just *defines* the problem she wants to solve, while in an operational approach, the programmer has to specify the precise sequence of steps needed to solve the problem. Usually, a definitional approach uses the mathematical notion of function application to inputs: the output answer is the composition of all the functions specified in the program. In an operational approach, there is a global state, consisting of variables that the program manipulates: the result of the program is the final state after all of the steps have been executed. Operational approaches may overspecify the problem because, in general, there might be more than one sequence of steps that solves the same problem (that is, results in the same final state). In such a case, there might be an advantage of choosing one particular sequence over another, for reasons of resource usage for example. But determining that some different, desired sequence of steps will lead to the same final state is a very difficult problem, and cannot usually be done. In a definitional approach, the sequencing (that is, the order in which the functions are evaluated) is left to the compiler, and often good use can be made of this freedom. In [Lee95], more connections are made between dataflow, visual languages, and various existing programming languages.

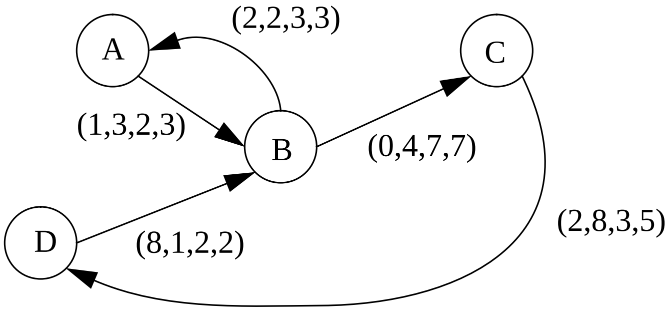


Figure 1-1 A computation graph. The 4-tuple associated with an edge represents the number of initial tokens, the number of tokens produced onto the arc, the number of tokens consumed from the arc, and the threshold parameter of the arc.

General dataflow graphs are Turing complete [Lee96b]. Of-course, this is trivially true if we do not make any restrictions on the actors: an actor could encode an entire Turing machine. However, dataflow graphs are Turing complete even if we make the restriction that all actors have finite state (or functional with self-loops allowed), and that actors produce tokens drawn from a finite alphabet. Because of these restrictions, an actor by itself cannot internally store integers of arbitrary size; hence, it cannot by itself have the expressive power of a Turing machine. However, the presence of potentially unbounded queues on the edges allows us to encode arbitrary integers using finitely-valued tokens, and this allows us to simulate a Turing machine using dataflow actors. Since key questions for Turing machines, such as the halting problem, are undecidable, such questions for dataflow programs are also undecidable.

1.4.1 Computation Graphs

Perhaps the earliest form of dataflow studied is the computation graph model proposed by Karp and Miller in their seminal 1966 paper [Karp66]. In this model, each actor consumes and produces a fixed number of tokens (known at compile time) on each invocation. In addition, each edge has zero or more initial tokens, and a threshold parameter that dictates that the sink vertex of that edge can be invoked only if the number of tokens on the edge is at least equal to the threshold (figure 1-1). Note that the threshold is at least equal to the number consumed on the arc. The primary focus in [Karp66] is in establishing the determinacy of computation graphs: Karp and Miller prove that the sequence of tokens produced on the edges does not depend on the order in which actors are invoked as long as all of the data precedences are respected. They are also interested in establishing termination conditions for these graphs. A program graph terminates if at least one actor is unable to continue firing because of a lack of input tokens. They formulate the problem as an integer linear program and show that an integer solution to these equations is necessary and sufficient to establish termination. Finally, they prove conditions under which buffer sizes on the edges remain bounded as the computation graph is executed.

1.4.2 Petri Nets

A Petri net is a directed bipartite graph consisting of two types of nodes: places and transitions (see figure 1-2). All edges in this graph are directed between places and transitions (or vice-versa), but never between two transitions or two places. Roughly speaking, places correspond to edges in a dataflow graph and hold tokens. A transition fires by removing a token from each of its input places and adding a token to each of its output places. A place cannot have a negative number of tokens; hence, a transition can fire only if each of its input places has at least one token. A marking is simply a function mapping each place to a non-negative integer representing the number of tokens in that place.

It turns out that Petri nets with the expressive power of Turing Machines result if so called inhibitor edges are allowed between places and transitions— these allow a transition to check whether a place has zero tokens [Petr81]. Equivalent to Petri nets are the vector addition systems from [Karp69]. The idea here is to look at sequences of vectors

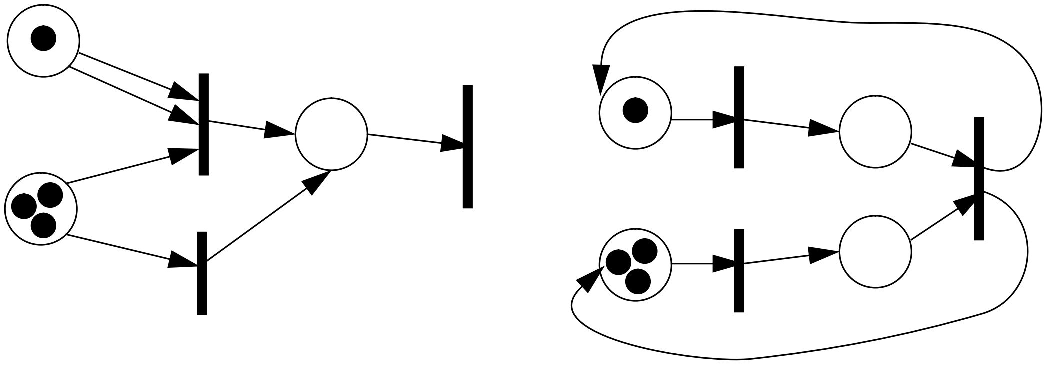


Figure 1-2 a) A Petri net. Places are the circles, and transitions are the heavy lines. The dots in the places indicate tokens. b) A Petri net equivalent to a dataflow graph.

(representing the marking of a Petri net) that result when elements of the vectors are incremented or decremented (representing the firing of transitions, or executions of actors in a computation graph). Karp and Miller's paper is mainly concerned with reachability analysis—that is, to determine whether a particular marking is reachable from some other marking.

1.4.3 Synchronous Dataflow

This model of computation was proposed by Lee in [Lee86] for use in specifying multirate DSP algorithms. SDF is a special case of Karp-Miller computation graphs in that the threshold parameter is always equal to the number of tokens consumed on the edge. In [Lee86][Lee87], algorithms are given that determine at compile time whether or not an SDF graph deadlocks, and whether or not an SDF graph has a periodic schedule that does not require unbounded memory. Algorithms are also developed for constructing static multiprocessor schedules, where the partitioning and sequencing is all done at compile time. SDF has since proved to be very suitable for describing a large class of useful DSP applications, and its strong formal properties have motivated its use (and some closely related models) in a large number of programming environments for signal processing [Lauw90, Lee89, Ohal91, Pino95a, Ritz92, Veig90]. Part of this thesis is devoted to the problem of constructing static schedules that minimize the amount of program and data memory required by the resulting software implementation.

If each actor in an SDF graph is required to have finite state and produce tokens drawn from a finite alphabet, then the SDF model is not Turing complete [Lee96b]. However, without these restrictions, the model is Turing complete. We will not be worried much about these issues in the rest of the thesis since they are mainly of theoretical importance. The presence of actors with theoretically infinite state does not affect our results since the optimizations we perform are at the SDF graph level, and at that level, we know that the buffers are of well-defined, bounded lengths provided that the SDF graph is sample-rate consistent.

The term "synchronous" in SDF is not to be confused with its usage in the context of synchronous reactive languages such as Signal, Lustre, and Esterel [Benv91]. In these languages, “synchronous” refers to the fact that there is a global clock (sometimes this is explicit as in the case of Lustre, and sometimes implicit as in the case of Esterel) that coordinates events. A variable (representing an infinite stream of tokens) is aligned with a boolean-valued clock signal, and takes on some value whenever its clock value is TRUE. In Esterel, a clock defines the instants in which a reaction takes place to external events. In Signal, a powerful algebraic methodology has been devised that allows the compiler to reason about the signals in the system and detect inconsistencies. In SDF, there is no notion of a clock: synchronous simply means that the schedules constructed for SDF graphs ensure that data is produced and consumed at the same rate over the period of the static schedule. “Synchronous” has also been used in the concurrency community to mean a particular style of synchronizing communication actions called “rendezvous”.

To avoid confusion, we emphasize that SDF is not by itself a programming language but a model on which a class of programming languages can be based. A library of predefined SDF actors together with a means of specifying how to connect a set of instances of these actors into an SDF graph constitutes a programming language. Augmenting the actor library with a means for defining new actors, perhaps in some other programming language, defines a more general SDF-based programming language. This thesis presents techniques to compile programs in any such language into efficient implementations.

1.4.4 Cyclo-static Dataflow

This is a closely related model to SDF; it has been proposed by the researchers in the GRAPE project at K. U. Leuven [Lauw94]. In this model, an actor has several phases: in each phase, it consumes a certain number of tokens and produces a certain number of tokens. The numbers of tokens produced and consumed can be different for each phase of the actor. Since the total number of phases of each actor is restricted to be finite, static periodic schedules can be constructed for cyclo-static graphs in a manner analogous to SDF graphs. The main advantage of this model appears to be a lower buffering requirement on the edges in certain types of graphs.

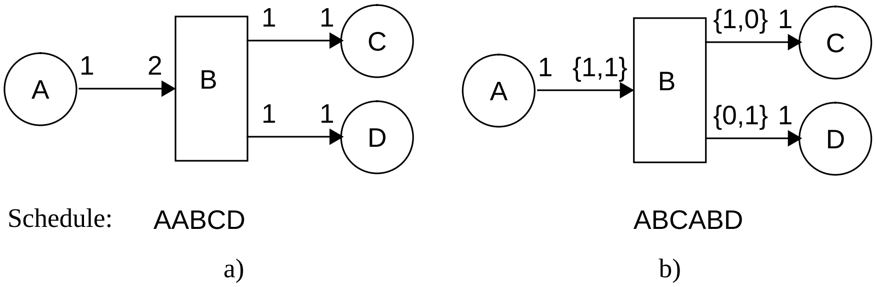


Figure 1-3 Cyclostatic dataflow compared to synchronous dataflow. Actor B is a distributor actor. a) SDF specification. b) CSDF specification.

For example, consider a *distributor* operator, which routes data received from a single input to each of two outputs in alternation (Figure 1-3). In SDF, this actor consumes two tokens and produces one token on each of its two outputs. In cyclo-static dataflow, by contrast, this operation can be represented as an actor that consumes one token on its input edge, and produces tokens according to the periodic pattern $1, 0, 1, 0, \ldots$ (one token produced on the first invocation, none on the second invocation, one on the third invocation, and so on) on the output edge corresponding to edge (B, C), and according to the complementary pattern $0, 1, 0, 1, \ldots$ on the edge corresponding to (B, D). The periodic SDF schedule in Figure 1-3(a) requires 2 units of memory for the buffer on edge (A, B), while the cyclostatic implementation in Figure 1-3(b) requires only 1 unit of memory. However, the cyclostatic schedule is larger, and it is not clear whether the reduction in buffer size is more than the increase in code size of the cyclo-static schedule (for example, the schedule $ABCABD$ is longer than $AABCD$). Further investigation is required to resolve these issues rigorously.

1.4.5 Multidimensional Synchronous Dataflow

Lee has proposed a multidimensional extension of SDF [Lee93] in which actors produce and consume *n*-dimensional rectangles of data, and each edge corresponds to a semi-infinite multidimensional sequence

$$\{x_{n_1, n_2, \ldots, n_m} | (0 \le n_1, n_2, \ldots, n_m < \infty)\}.$$

For example, an actor could produce a 2×3 grid consisting of six tokens each time it is invoked. This extension to multidimensional dataflow has several advantages over SDF, especially for multidimensional, multirate signal processing applications. One advantage is that data parallelism is exposed to a much greater degree in an MDSDF specification than is possible with SDF. This issue is crucial in image processing algorithms where there is a lot of data parallelism present, and a specification model that exposes all of it to a compiler could result in better implementations. Secondly, certain forms of control flow that cannot be expressed succinctly in SDF can be expressed succinctly in MDSDF. Thirdly, the presence of more than one dimension allows more efficient and intuitive specification of one-dimensional algorithms since certain bookkeeping activities can be done in the other dimension.

However, this model appears to be most useful for specifying multirate, multidimensional signal processing systems. When there are 2 or more dimensions, there are many possible choices for the sampling geometry, and the standard rectangular grid is not necessarily the “best” choice. The MDSDF model as specified in [Lee93] has n-dimensional rectangles as data structures, and can only model systems using the rectangular sampling grid. Part of the contribution of this thesis is to present a generalization of MDSDF to handle arbitrary sampling lattices. The key issue that arises is whether the generalized model can be scheduled statically—it turns out that our model *is* amenable to static scheduling.

1.4.6 Systolic Arrays

A systolic array is a network of processing elements (P.E’s) interconnected in a regular fashion [Kung88]. These elements are often very simple; for example, adders and multipliers. The processors compute and pass data along in a “rhythmic” fashion; the behavior of each processor, that of pumping data in and out in a regular fashion, each time performing some short computation, is akin to the behavior of the heart pumping blood through the arteries in a regular fashion (hence the moniker “systolic” arrays). A large amount of work has been done to determine techniques for optimally mapping algorithms onto systolic arrays (for example, by minimizing the number of processing elements, maximizing the throughput, or minimizing the buffers on the arcs). A general class of applications that are realizable by systolic arrays are the so-called regular iterative algorithms [Kung88], which include many problems from linear algebra like LU decomposition, and matrix multiplication, problems requiring dynamic programming such as finding the shortest path in a graph, and many signal processing problems like convolution and IIR filtering. Roughly speaking, a regular iterative algorithm has a dependence graph with a highly regular topology and connectivity that is highly localized (that is, there are no “global” communications). Given such an algorithm, it is possible to generate an optimized systolic architecture that implements the algorithm. Similarly, *wavefront* arrays are like systolic arrays but instead of being globally synchronized, they use a data-driven computing paradigm like dataflow. However, wavefront arrays also have the regular structure found in systolic arrays.

There are several differences between MDSDF and the dependence graphs amenable to implementation as systolic arrays. MDSDF is a dataflow model, and MDSDF graphs, like SDF graphs, can be of arbitrary topology. Hence, it is not necessary that MDSDF graphs have the sort of regular structure that dependence graphs of regular iterative algorithms do. Secondly, one of the possible uses of MDSDF is to provide a specification language from which implementations can be derived for general purpose multiprocessor architectures. These include architectures like the Philips Video Signal Processor (VSP), where the number of processing elements is fixed and is not scalable with the size of the algorithm, and where MIMD (multiple instructions multiple data) design is necessary due to the differing granularity of the tasks and non-regular control flow. Hence, the work on scheduling dependence graphs onto systolic and wavefront arrays has limited use in the context of MDSDF.

1.4.7 Boolean Dataflow

While the above models all have restricted expressive power, the *boolean dataflow (BDF) model,* which was defined by Lee in [Lee91] and explored further by Buck in [Buck93], has significantly more expressive power. In this model, the number of data values produced or consumed by each actor is either fixed, as in SDF, or is a function of a boolean-valued token produced or consumed by the actor. Buck addresses the problem of constructing a non-empty sequence of conditional actor invocations, where each actor is either invoked unconditionally or invoked conditionally based on the value of boolean tokens. This sequence should produce no net change in the number of tokens residing in the FIFO queue corresponding to each edge. Such an invocation sequence is referred to as a *complete cycle*, and clearly, if a finite complete cycle is found, it can be repeated indefinitely, and a finite bound on the amount of memory required (for buffering) can be determined at compile-time. It can be shown that the boolean dataflow model is Turing-complete. Thus, a number of key decision problems for BDF graphs are undecidable, including the problem of finding finite complete cycles (and thus, the problem of determining whether a graph can be implemented in bounded memory). Buck presents heuristic techniques for finding finite complete cycles for BDF graphs. Whenever his techniques fail, the graph has to be executed dynamically, although clustering techniques can often significantly reduce the number of tasks that have to be executed dynamically [Buck93].

1.4.8 Well Behaved Dataflow

Gao *et al.* have studied a programming model, called well-behaved dataflow, in which non-SDF actors are used only as parts of predefined constructs [Gao92]. Of the two non-SDF constructs provided, one is a conditional construct, and the other is a looping construct in which the number of iterations can be data-dependent. This restriction on the use of more general actors guarantees that infinite schedules can be implemented with bounded memory. Gao’s model, although more general than SDF, has significantly less expressive power than the BDF model of Buck because it is not Turing-complete (this follows from the fact that every graph in this model can be implemented in bounded memory).

1.4.9 Kahn Process Networks

The Kahn process network model is an elegant model of concurrency that was proposed by Gilles Kahn in his seminal 1974 paper [Kahn74]. In his model, there are a number of processes that communicate by sending data to each other over channels which have potentially infinite queues. The processes execute concurrently; a process may write to a channel whenever it wants, but if it wants to read from a channel and the channel is empty, then it must block until it gets data on that channel. In other words, a process cannot base an action on the presence or absence of data.

Kahn is mainly interested in defining precisely the function computed by the network. He proceeds by constructing a partial order of streams based on the prefix order. A stream is a potentially infinite sequence of tokens, and captures the movement of data along the communication channel (also called the channel's history). In the prefix partial order, a stream A is less than or equal to a stream B if A is a prefix of B. Clearly, if A is a prefix of B, then B has more information than A, and thus represents a more evolved state than A. Kahn constrains the processes that implement blocking reads and non-blocking writes to correspond to continuous functions on the prefix partial order; this simply prevents a process from waiting for an infinite amount of time before producing outputs. Thus, Kahn is able to use the Tarski fixpoint theorem to show that the function computed by the network is the least fixed point of their composition. Since the least fixed point of a continuous function on a complete partial order is always unique, it follows that Kahn process networks are determinate in the sense that there can only be one valid computation (that is, exactly one history on each channel) for a particular set of inputs. The fixed point in the network can be reached by executing the processes in the usual way since Tarski's theorem tells us that the least fixed point can be reached by starting the computation on the all-undefined state—in the process network, this is simply the state where all queues on the channels are empty. Also, the least fixed point of a Kahn process network is usually a set of infinite sequences; hence, any execution of the network for a finite amount of time represents an approximation of the final answer. Put another way, the operational behavior of the network converges to the denotational meaning only when time goes to infinity.

Kahn’s conceptualization of dataflow networks in terms of these continuous processes is a powerful one; the determinacy of models like computation graphs, SDF graphs, BDF graphs all follow from this since each of these models satisfies the assumptions made of processes in the Kahn model.

1.5 Non Dataflow MoCs

While we are only concerned with dataflow MoCs in this thesis, it is worthwhile to briefly review some non-dataflow MoCs. These include Statecharts for specifying control-dominated applications, numerous other discrete event simulators used for modeling hardware, queueing networks, communication protocols, and hybrid dynamical systems used for modeling systems with continuous time and discrete time behaviors.

1.5.1 Discrete Event Simulators

Unlike the dataflow MoCs we have discussed so far, data values in a discrete event (DE) system also have an associated time stamp. An event is a tuple $\{e_k, t_k\}$ where e_k is the kth datum and t_k its timestamp. A discrete event simulator processes events chronologically, in order of increasing time. There is typically a global event queue in which all the events in the graph are stored, and the actor that is invoked next is the one that has input events with the smallest timestamps. Typically, maintaining the global event queue is a computationally expensive procedure. The presence of simultaneous events often leads to non-determinacy, where the result of the computation depends on the order in which blocks are invoked. The simultaneity problem can be partially solved by insisting that each block add an infinitesimal delay on its output events. Networks of such Δ*-causal* systems have been shown to be determinate [Yates93]. The presence of feedback loops can also cause problems; we refer the reader to [Chan96] for a discussion of these issues. Examples of popular discrete event simulators include the VHDL language used for hardware specification and synthesis, and numerous simulation tools such as SIMAN, SIMULA, GASP, and SLAM [Cass93].

1.5.2 Statecharts

Statecharts is a visual language based on a hierarchical finite state machine model of computation. Hierarchy is a useful feature, both for designing ease, and for managing complexity. For example, a compiler can use hierarchy in the specification during scheduling and compilation intelligently and avoid the blow-up in the size of the program that can result if the hierarchy is flattened or is not there to begin with. However, the semantical issues become more challenging when hierarchy is allowed. Statecharts attempts to precisely define the syntax and semantics of hierarchical FSM systems. The original specification of Statecharts had a number of ambiguous definitions; as a result there are at least 22 different variants of the language in use today, each having slightly different semantics and formal properties [Beec95].

1.5.3 Hybrid Dynamical Systems

A hybrid dynamical system consists of some sort of FSM at the top level, while within each state, the system evolves according to a set of continuous time differential equations. Transitions between states at the top level occur when the system within the present state evolves to a point that triggers a change of state. These systems are useful for modeling scenarios where, for example, some discrete event protocol interacts with a system having continuous time dynamics.

For example, consider the problem of designing the controller for the crossing gate at a railway track intersection. This controller has three states at the top level: gate up, gate lowering, and gate down. In each state, there is a differential equation according to which the system variables evolve. The system variables are the position of the train, and the position of the gate. When the gate is up, the system evolves according to the continuous time dynamics of the train. When the train nears the gate, the controller moves into the “gate lowering” state. When the gate is being lowered, the evolution is a pair of differential equations, the second one modeling the movement of the gate.

The research in this area is still new, and the objectives are mainly formal verification: to formally prove that the system does not enter any bad states, where a bad state is defined to be a region of the continuous state space that forms the domain of the variables in the system. In the train gate controller example, this is the region where the gate is still up even though the train is passing through. There are essentially two degrees of modeling freedom in these systems: the generality of the finite state machine model at the top level, and the generality of differential equations allowed in the states. Typically, the top-level model is a basic finite state machine, and the continuous variables have piecewise linear trajectories (resulting in the so-called linear hybrid systems). Typically, one tries to prove that the verification problem is decidable, rather than undecidable. Making the differential equations more general usually results in the verification problem becoming undecidable. Even when decidable, the decision algorithms typically have exponential or super exponential lower bounds because of the state space explosion problem (formal verification methods typically explore the entire state-space in order to ensure that bad states are not reachable; however, the size of the state-space can be exponentially bigger than the size of the specification), so that heuristic techniques of state-space reduction have to be used to get reasonable running times. The papers [Alur95][Henz96] provide a starting point for exploring this research area.

1.6 A Denotational Approach for Classifying MoCs

Terms like “synchronous”, “discrete event”, “dataflow” have been used somewhat loosely in the past. For instance, one research group coined the term “asynchronous dataflow” to mean the opposite of “synchronous dataflow”; i.e, dynamic dataflow where actors do not consume and produce fixed numbers of tokens known at compile time. This use of “asynchronous” contradicts its usage in other contexts where it implies a self-timed concurrent execution policy, which can certainly be used on a “synchronous” dataflow graph. There has been some work recently that attempts to define and classify, in a mathematically rigorous manner, the various models used for specifying reactive systems [Lee96]. The approach in [Lee96] is denotational in flavor, in that it attempts to give meanings to systems constructed using different models of computation (in contrast, an operational approach would attempt to specify the precise method of execution in a model of computation). The basic result from [Lee96] is a classification of various kinds of systems in terms of signals and functions on signals. Signals are defined to be possibly-infinite sets of tag/value tuples, where tags are usually drawn from some infinite partially ordered set. The different classifications arise when the set of tags is uncountably infinite, countably infinite, when the tags can be put into one-on-one correspondence with the integers in an order preserving way, etc. Fixed-point semantics are used to give meaning to networks of processes (functions) with feedback loops.

1.7 Other Programming Languages

There are several programming languages that are also relevant to the research in this thesis, and some of these, like the synchronous reactive languages, have been mentioned already. Some of these languages have a dataflow flavor to them, like Signal, and some are based on finite state machines, like Esterel. Other particularly interesting and relevant languages are APL, Lucid, and Sisal. These languages have all been developed to bring multidimensional programming into the realm of functional, declarative languages.

As already mentioned, imperative languages overspecify the computation; this makes it difficult to compile this code onto parallel architectures. This has motivated a large body of work in developing efficient compilers that can extract parallelism from FORTRAN or C programs automatically. For example, nested loops often pose great problems for parallelism extraction because the usual dependency analysis may not be good enough to deduce what is really going on. For instance, if a statement A writes to an array X, and a statement B after statement A reads from array X, straightforward dependency analysis suggests that there is a dependency between statements A and B. However, it could be that the index into which A is writing to is never the same as the index from which B is reading from; hence, in reality, there might be no dependency at all. It is possible to systematically deduce these dependencies for particular classes of array indexing functions; for example, if the indexing function is an affine function of the loop indices. These dependency tests frequently involve solving simultaneous Diophantine equations, as in the Banerjee test [Bane88][Zima90].

The languages community, in contrast, is tackling the problem from the other end, by trying to provide languages where arrays and indexing can be expressed declaratively, or functionally, rather than imperatively. This should make the job of the compiler significantly easier, and would theoretically lead to more efficient implementations. However, current functional languages are inept at handling large multidimensional objects like arrays [Mull91]. There are several reasons for this. Firstly, allocating and filling in values into data-structures is an alien concept in declarative semantics since data-structures are results of expressions in such languages. Secondly, referential transparency requires that any array element can only be defined once; recursive definitions of arrays where this might be violated are illegal. Thirdly, operations that modify the array must first copy the array, again because once an array has been defined, it may not be written to. The cost of copying large arrays is expensive.

The first problem has been solved by so called array comprehensions. These allow various subregions in an array to be defined in a single expression. Moreover, the ordering of the definitions of the subregion does not matter; only the data dependencies are specified and from this the compiler can choose any initialization order it wants (see [Huda89] for an example).

One of the first functional languages to incorporate arrays as a basic type was Iverson's APL (A Programming Language) [Iver62]. APL permits direct manipulation of arrays as complete entities and provides functions that can be applied simultaneously over all entries in an array. However, APL has proved difficult to implement and compilers for APL do not produce code competitive to FORTRAN code (see papers 2 and 3 in [Mull91]). One of the main goals in the design of APL was succinctness; in this it succeeded rather well. Programs written in APL use a special character set, and are succinct to the point of being cryptic. APL was also the language that inspired FP [Back78], which in turn rejuvenated interest in definitional programming paradigms.

The "Concrete domains" work of Kahn and Plotkin [Kahn93], and the following[1] sequential algorithms work of Berry and Curien [Berr85] is an attempt by theoreticians to extend denotational theories to computations that operate on the so-called "deterministic concrete data-structures" (dcds); these structures encapsulate, in an abstract way, most common data structures like arrays, records, lists etc. Essentially, a dcds consists of a number of cells where each cell may be filled with a unique value. A cell has an enabling

[1] It should be noted that the material in [Kahn93] actually appeared in french as an INRIA technical report in 1978; it was published in english in 1993.

condition that dictates that it be filled with a value if and only if the enabling condition is satisfied. The enabling condition usually takes the form of specifying that some other cell or cells be filled with particular values. It can be seen that intuitively, the cells are elements of a complete partial order, induced by the enabling conditions; this gives these data-structures specific mathematical properties [Kahn93]. A sequential algorithm is roughly defined to be a so-called Kahn-Plotkin sequential function whose inputs and outputs are states of a dcds. These functions can be composed to build up an applicative program and it is shown in [Berr85] that these programs have the full abstraction property: the operational specification of sequential functions and their compositions is identical to the denotational semantics that gives meaning to such a program.

Despite the overspecification problem, imperative languages still outperform functional languages; they can be compiled down to very efficient implementations that make good use of the capabilities of the machines and run many times faster than equivalent implementations generated from functional programs. On the other hand, imperative languages are generally considered to be difficult to maintain and change since even a minor change could have all sorts of unexpected side effects. One expects that ultimately there will be good compilers for functional languages that can compete with imperative languages. Below we mention a couple of interesting functional languages suitable for multidimensional programming.

1.7.1 Lucid

Lucid is a declarative language with a dataflow flavor [Ashc95]. Declarative is similar to functional: it means that the programmer specifies the computation she wants, and not the precise operational details for executing the computation. Lucid has a notion of multidimensional indexing which makes it possible to write programs that operate on multidimensional data objects. For example, in the construct

```
seq = posint fby.a runningSum.b(drop.b(seq,n));
```

`fby.a` denotes “followed by in dimension a”, `runningSum.b` denotes “the procedure `runningSum` should operate on dimension b”, and `drop.b(seq,n)` denotes “drop every nth value from `seq` in the `b` dimension”, where these dimensions are all assumed to be orthogonal. As another example, the following construct is used as the basic computation step in a routine for solving Laplace's equation for heat transfer in three dimensions:

```
avg(M) = (left M + right M + up M + down M + front M + rear M)/6
```

where the expression `left M` denotes the value of `M` at the current (horizontal) space coordinate minus one, just as `down M` denotes the value of `M` at the current vertical space coordinate minus one.

The developers of Lucid argue that Lucid can be viewed as an application of *intensional* logic. Intensional logic is a theory invented by logicians to study natural language statements that have context sensitive meanings. However, these contexts are usually "hidden", and not explicit. Even though conventional forms of *extensional* logic, like temporal logic, can be used for reasoning about such statements, the resulting formalisms are usually unnatural and do not have the intuitive feel that the natural language statement would. In Lucid, the "contexts" are the indices of the multidimensional data-structures, and these are implicit, as the two example statements given above show. While these statements can also be thought of extensionally, where data objects are infinite sequences in the style of Haskell, and Lucid operators like `first`, `next`, and `fby` correspond to the list operations `head`, `tail`, and `cons`, the developers argue that it is more natural to interpret Lucid statements in an intensional manner.

The intensional approach in Lucid naturally leads to a demand-driven style of operational semantics (called *eduction* in [Ashc95]); that is, demands for values at given contexts generate demands for other, possibly different, values at other, possibly different contexts. The demand-driven style of evaluation is roughly similar to lazy evaluation (which is an efficient implementation of the normal-form reduction order in the lambda calculus), in that values that are not needed are never computed. Normal order reduction is guaranteed to terminate in the normal canonical form if one exists for the expression. Applicative order reduction in the lambda calculus, on the other hand, makes the fatal mistake of evaluating everything at least once, and this can result in a non-terminating series of reductions, even though there is a normal canonical form [Alli86][Stoy77].

The multidimensional contexts in Lucid can be used for programming constructs that compute things that are not inherently multidimensional. The different dimensions can be used for “temporary” storage. For example, in a matrix multiplication specification, a problem that is inherently 2-dimensional, a third dimension is used to lay out the row from one matrix and the column from the other matrix so that a dot-product may be formed. As will be shown in chapter 4, multidimensional synchronous dataflow is very similar in this regard since temporary dimensions can be used for computations that do not ordinarily require them. The use of extra dimensions can however uncover all of the parallelism that might be present in the algorithm so that a compiler could make good use of it.

However, it should be noted that Lucid is a general, possibly Turing complete programming language whereas MDSDF is not Turing complete, if MDSDF actors are restricted to have finite state. Hence, since static schedules can be constructed at compile time, we can not only construct schedules, but we can optimize them for various metrics, something that is usually not possible for more general languages. Also, it is not clear whether it would be easy to express the sort of programs we want to in Chapter 5; the extension of MDSDF given there seems to be better suited for the task.

1.7.2 SISAL

SISAL stands for Streams and Iterations in a Single Assignment Language. This language is a general purpose applicative language intended mainly for large scale scientific applications. Since arrays are an integral part of such computations, arrays and array operations are included in the language definition (see the paper by Feo in [Mull91], and also [McGr83]).

Sisal includes array comprehensions that allow arrays to be defined as expressions. Recursive definitions of arrays are not allowed; this solves the second problem mentioned before, namely, the problem of defining some elements of the array more than once. Some effort is also made to eliminate copy operations; the compiler does some node re-ordering so that read operations are scheduled before write operations, and some runtime checking to identify the last user of an array.

One attractive feature of arrays in Sisal is that arrays can be “jagged”. This is possible because an n dimensional array in Sisal is defined to be an array of $n-1$

dimensional arrays. The size and bounds of each of these arrays can be different; hence, an n dimensional array need not be a rectangular hypercube.

Sisal is also a general purpose language, and hence the emphasis is not so much on static scheduling as it is in MDSDF.

1.8 Modularity and Code Generation

Graphical programming environments for DSP normally contain palettes of graphical icons that correspond to predefined computational blocks, and the program is constructed by selecting blocks from these palettes and specifying interconnections. If some functionality is desired that is not available in the existing library, then it is usually easy to define a new function and add it to the library, upon which the new function can become available to all other users of the system. Thus, the format of graphical programming environments makes it natural and convenient to recycle software and the development effort. For example, since each function is defined only once, it becomes economical to spend a large effort to hand-optimize frequently used functions for efficiency.

1.8.1 Compiling SDF and MDSDF Graphs

The approach used in many signal processing programming environments for constructing software implementations from block diagrams is the technique of *threading* [Bier95]. In this approach, the underlying graph model, in this case SDF or MDSDF, is scheduled to generate a sequence of actor invocations. A code generator then steps through this schedule and generates the machine instructions necessary for the computation specified by each actor it encounters; these instructions are obtained from a predefined library of actor codeblocks. Finally, a memory allocator assigns variables to memory locations, and does some other bookkeeping activities. The result is the target program.

An alternative approach used by some systems is synthesis; in this approach, the graph is first translated into some intermediate language. This translation could be by threading codeblocks specified in the intermediate language. Then the language is compiled into assembly language. Since the threading technique can be used in synthesis as well, we assume that a threading model is being used. We also assume that the code-generator generates inline code; the alternative of using subroutine calls can have unacceptable overhead, especially if there are many small tasks in the graph. Inline code presents its own problems however, the chief one being code size explosion. For example, if an actor appears 20 times in the schedule, then there will be 20 codeblocks in the target code for that actor. DSPs have very tight constraints on on-chip memory; hence, the generated code might be unacceptable. However, for SDF graphs, it is usually possible to generate the so-called single appearance schedules where each actor appears only once in the schedule; for these schedules, inline code generation results in code that has the least size to a first approximation. We refer the reader to [Bhat96a] for a more thorough discussion on various issues involved in compilation from block diagrams.

1.8.1.1 Memory Usage

An important problem that arises when compiling SDF programs is the minimization of memory requirements—both for code and data (intermediate results). This is a critical problem because programmable digital signal processors have very limited amounts of on-chip memory, and the speed and financial penalties for using off-chip memory are often prohibitively high for the types of applications, typically embedded systems, where these processors are used. Moreover, off-chip memory typically needs to be static, increasing the system cost considerably. One concern in this thesis is to develop techniques to minimize the code and buffering size (for the buffers on the edges between blocks) when compiling an SDF or MDSDF program.

As will be discussed later, large sample rate changes result in an explosion of code size requirements if naive compilation techniques are used. Hence, we use a particular class of schedules that do not result in the code-size explosion problem, and we show that there is considerable opportunity for optimization for buffer memory usage within this class of schedules. We develop techniques for producing optimized schedules that minimize both the code size and the buffering requirements.

Below, we review some alternative approaches to code generation for embedded systems.

1.8.2 Compilers

There have been a number of reports on the inability of high-level language compilers to deliver satisfactory code for time-critical DSP applications [Geni89, Tow88, Yu93, Zivo95]. The throughput requirements of such applications are often severe, and designers typically resort to careful manual fine-tuning to sufficiently exploit the parallel and deeply pipelined architectures of programmable digital signal processors while meeting their stringent memory constraints. For example, a study was done by Zivojnovic *et al*. on the performance of C compilers for several popular DSPs [Zivo95], The best performance exhibited by these compilers — by the Tartan compiler for the Texas Instruments TMS320C40, and the ADI 2.0 compiler for the Analog Devices ADSP 21060 — exhibited overheads over handwritten code of 290% and 219% for execution time, 44% and 57% for program memory, and 0% and 0% for data memory respectively. These numbers were measured on an FIR filter benchmark. The worst performance exhibited by these compilers —by the ADI 5.1 for the Analog Devices 2101—had overheads of 775%, 250%, and 0% for execution time, program memory, and data memory respectively. Although there is new research that appears promising [Liao95], it is too early to determine whether compilers that come within factors of 1.2-1.5 of hand-written code can be developed for DSPs. However, it is worth repeating that even if there are good compilers available, there are still compelling reasons to use block-diagram languages because these languages can be more easily mapped to multiprocessor architectures than just straight C code, they are more intuitive to use, they are more modular, and more amenable to formal verification due to their strong formal properties. Hence, the optimizations that we develop in this thesis are global in the sense that they apply to the overall graph structure; the code within the individual actors is assumed to be optimized already, either by hand or by a good compiler.

1.8.3 Subroutine Libraries

The use of optimized subroutine libraries, as described earlier, is one approach to improving efficiency without forcing the user to write or fine-tune code at the assembly language level. A second approach is to add extensions to a high level language that facilitate the expression and optimization of common signal processing operations [Lear90]. This can be highly successful in some compilers; for example, when DSP extensions to C are used, the Tartan compiler achieves an overhead of only 5% in execution time, and has no overhead in program or data memory on the FIR filter benchmark. However, in other compilers, this may not be as useful. The afore-mentioned ADI 5.1 compiler for the 2101 actually gives worse results when compiling the FIR benchmark with DSP extensions: overheads of 885%, 283%, and 0% for execution time, program, and data memory respectively [Zivo95].

Another approach is the application of artificial intelligence techniques to confer optimization expertise to high level language compilers [Yu93]. Although it has not been extensively evaluated yet, preliminary results show promise.

1.8.4 Block Libraries

The alternative that is pursued in this work is the use of graphical or textual block diagram languages based on the SDF and MDSDF model and its extensions in conjunction with hand-optimized block libraries. As is discussed in Chapter 2, the SDF model allows us to schedule all of the computations at compile-time and thus eliminate the run-time overhead of dynamic sequencing. This increased efficiency comes at the expense of reduced expressive power: computations that include data-dependent control constructs cannot be represented in SDF. However, SDF is suitable for a large and important class of useful applications, as the large number of SDF-based signal processing design environments suggests. Benchmarks on the Gabriel design environment [Lee89] showed that compilation from SDF block diagrams produced code that was significantly more efficient than that of existing C compilers [Ho88a], although not as efficient as hand-optimized code. For a restricted model of SDF in which each computation produces only one data value on each output and consumes only one data value each input, the Comdisco

Procoder block diagram compiler produced results that were comparable to the best hand-optimized code [Powe92]. The reason for this impressive performance is that traditional compilers apply optimizations mostly within basic blocks [Aho88], while SDF compilers have more knowledge of the control structure of the program (as mentioned above, the sequencing of SDF computations can be fixed at compile time) and can thus apply optimizations globally. Hence, a hybrid compiler/SDF-compiler approach to block diagram based code synthesis should prove to be very competitive with hand-optimized code (such a test cannot be made currently since existing compilers perform rather poorly as already mentioned). Although the performance of the Comdisco Procoder is impressive, the restricted computational model to which its optimizations apply does not support systems that have multiple sample rates or decisions.

1.9 Block Diagram Languages—History

There are several graphical programming environments for DSPs available commercially. Some of these include the Signal Processing WorkSystem marketed by the Alta group at Cadence Design Systems [Barr91], the COSSAP environment [Ritz92] that grew out of a research project at the Aachen University of Technology, now being marketed by Synopsys, the DSP Station developed by Mentor Graphics, the GRAPE environment [Lauw90] developed at the Katholieke University of Leuven and now marketed by Eonic systems, and the Hyperception environment. It should be noted that not all graphical programming environments for DSPs have succeeded. The Sproc chip, a 4-processor single chip DSP made by Star Semiconductor, also had a graphical environment that came packaged with a graphical software development system. The development system included automated partitioning and scheduling tools. However, despite the adequate programming tools offered, the company and product failed. Several additional graphical programming and simulation environments for DSP are described in [Desm93, Kapl87, Karj88, Olso92, Kons94, Reek92, Shan87].A comprehensive survey of the state-of-the-art tools for DSP programming can be found in [Bier95].

The earliest block diagram language to be used for signal processing was at the Bell Laboratories in the 1960s, where a block diagram compiler was developed for acoustic research [Kell61]. A graphical programming environment for designing digital filters in presented in [Covi87] (this environment uses only two types of actors: adders and multiplication by constants). At Advanced Micro Devices, a graphical tool was developed that allows mapping of a DSP algorithm onto a two-dimensional array of processors [Ziss87]. At Carnegie Mellon University, an environment targeting the iWarp multicomputer was developed [Ohal91]. The Khoros program that grew out of research done at the University of New Mexico is used widely for image and video processing simulations [Kons94]. At the University of California at Berkeley, work in graphical environments for DSP and communication is rooted in the BLOSIM system developed by Messerschmitt in 1984 [Mess84]. This work inspired the development of the Synchronous Dataflow (SDF) model [Lee86], and provided the foundation for the Gabriel system [Lee89]. The Gabriel system had code-generation capability, with the target processor being the Motorola 56000 DSP. The Ptolemy project was started in 1990 as the successor to Gabriel [Buck94]. The Ptolemy project is an object-oriented framework for the specification, simulation, and software-synthesis of large scale heterogeneous systems. Ptolemy supports multiple models of computation, unlike Gabriel which only had one model of computation—SDF. Different models of computation can interact seamlessly in Ptolemy, and new models of computation can easily be added by an experienced user. Models of computation available in Ptolemy include SDF, discrete event, dynamic dataflow, boolean dataflow, and Kahn process networks. Ptolemy also has extensive code-generation capabilities, and some of the processors targeted have included the Sproc multiprocessor DSP from Star Semiconductor [Murt93], the Motorola DSP 56000 and DSP 96000 [Pino95a], and the Texas Instruments TMS320C50 DSP chip.

The block diagram environments mentioned above are all based on MoCs with varying degrees of formal properties. However, some researchers have proposed block diagram environments where there is essentially no MoC at all; an example of this approach is the Pope project from IMEC in Belgium [Verk96]. In the Pope environment, a module can be specified in C, VHDL, or DFL (data flow language). Multiple protocols for communication are allowed between modules, unlike most of the MoCs in the literature. The use of different protocols leads to different semantics.

1.10 Overview of the Thesis

In this chapter, we have argued in favor of using large grain dataflow languages for specifying real-time signal processing systems. We have concentrated on two particular subsets of dataflow: synchronous dataflow, which has proved to be useful for expressing signal processing algorithms, and its extension to multiple dimensions, multidimensional synchronous dataflow. We have also argued in favor of block diagram programming environments, and described a strategy for synthesizing software from SDF/MDSDF-based graphical programs. We have shown how scheduling plays an important role in the compilation process.

In the next chapter, we review the SDF model in detail, and formalize the scheduling concepts needed. We review the concept of single appearance scheduling and describe its property of requiring the least amount of program memory when software implementations are generated. We show that one particular scheduling problem is NP-complete and that we have to resort to heuristics in general. We also establish a simple lower bound on the buffer size required on any edge in a consistent SDF graph.

In Chapter 3, we identify the problem of constructing single appearance schedules that minimize the amount of buffering memory required. We describe formally the buffering model we use, and contrast our buffering models to some other possibilities. The buffering model we choose is shown to have several useful properties in contrast to other models. We then build up some machinery to describe single appearance schedules that minimize the amount of buffering memory required, and we give a polynomial time algorithm for generating an optimal loop hierarchy, thereby minimizing the buffering memory. This algorithm is shown to be optimal for a restricted set of graphs called well-ordered graphs. Several extensions are given. We then tackle general acyclic graphs and show that the problem becomes NP-complete. Two heuristics are developed, and an extensive experimental study is given that shows the efficacy of these techniques. Finally, we extend these techniques further to arbitrary SDF graphs, and conclude with some discussion of related work. Much of the work in this chapter was done in collaboration with Dr. Shuvra Bhattacharyya, a researcher at the Hitachi Systems Research Laboratory in San Jose, California.

In Chapter 4, we review the MDSDF model, and establish some results regarding self-loops. In SDF graphs, the precedence constraints imposed by self-loops are relatively easy to conceptualize; in MDSDF, it requires a bit more effort. We then extend the scheduling results from Chapter 3 to MDSDF schedules.

In Chapter 5, we identify a shortcoming of the MDSDF model, namely, its inability to specify multidimensional, multirate signal processing systems sampled on non-rectangular lattices. In a multidimensional, discrete-time signal, there can be several choices for the sampling geometry; that is, the periodic “grid” from which samples are taken. The straightforward extension of one-dimensional sampling leads to the so-called rectangular sampling structure (also called a rectangular lattice), and MDSDF is capable of expressing systems dealing with rectangularly sampled signals. However, a more general sampling structure in multiple dimension is a geometric lattice, and in general, the lattice may not be a rectangular lattice. Non-rectangular lattices can be useful in practice because the sampling density required to represent the signal might be lower than with an equivalent rectangular lattice.

We review multidimensional signal processing concepts, and present a dataflow model that is capable of expressing systems sampled on non-rectangular lattices. We concentrate on static scheduling, and show that the problem of computing the balance equations becomes fundamentally more difficult. In particular, the balance equations in this model include equations requiring the computation of so called “integer volumes” of parallelepipeds. However, we are able to develop a method that allows these equations to be solved, thereby permitting static schedules. A practical example using this model is given. The problem of computing integer volumes turns out to be an interesting number-theoretic problem and we give a partial solution to this problem at the end of Chapter 5. In the final chapter, we present our conclusions and directions for future research.

2

Synchronous Dataflow

2.1 Notation

Given a finite set P of positive integers, we denote by $gcd(P)$ the greatest common divisor of P — the largest positive integer that divides all members of P, and we denote the least common multiple of the members of P by $lcm(P)$. If $gcd(P) = 1$, we say that the members of P are **coprime**. Given a finite set R of real numbers, we denote the largest and smallest numbers in R by $max(R)$ and $min(R)$, respectively. Also, if r is a real number, we denote the largest integer that is less than or equal to r (the "floor" of r) by $\lfloor r \rfloor$, and we denote the smallest integer that is greater than or equal to r (the "ceiling" of r) by $\lceil r \rceil$. We denote the number of elements in a finite set S by $|S|$.

2.1.1 Graph Concepts

By a **directed multigraph**, we mean an ordered pair (V,E), where V and E are finite sets, and associated with each $e \in E$ there are two properties $src(e)$ and $snk(e)$ such that $src(e), snk(e) \in V$. Each member of V is called a **vertex** of the directed multigraph and each member of E is called an **edge**. If e is an edge in a directed multigraph, we say that $src(e)$ is the **source** vertex of e; $snk(e)$ is the **sink** vertex of e; e is **directed from** $src(e)$ **to** $snk(e)$; e is an **output edge** of $src(e)$; and e is an **input edge** of $snk(e)$.

Given two not necessarily distinct vertices v_1 and v_2 in a directed multigraph (V,E), we say that v_1 is a **predecessor** of v_2 if there exists $e \in E$ such that $src(e) = v_1$ and $snk(e) = v_2$; we say that v_1 is a **successor** of v_2 if v_2 is a predecessor of v_1; and we say that v_1 and v_2 are **adjacent** if v_1 is a successor or predecessor of v_2; and if v_1, v_2 are distinct, then $\{v_1, v_2\}$ is called an **adjacent pair**. Two subsets $V_1, V_2 \subseteq V$ are adjacent if there exist vertices $v_1 \in V_1$ and $v_2 \in V_2$ such that v_1 and v_2 are adjacent. By a **subgraph** of a directed multigraph $G = (V, E)$, we mean the directed multigraph formed by any $V' \subseteq V$ together with the set of edges $\{e \in E | (src(e), snk(e) \in V')\}$. We denote the subgraph associated with the vertex-subset V' by $subgraph(V', G)$; if G is understood from context, we may simply write $subgraph(V')$. A **path** in (V,E) is a nonempty sequence $e_1, e_2, e_3, \ldots \in E$ such that $snk(e_1) = src(e_2)$, $snk(e_2) = src(e_3)$, …. We say that the path $p = e_1, e_2, e_3, \ldots$ **passes through** each member of

$$Z_p = (\bigcup_i \{src(e_i)\}) \cup (\bigcup_i \{snk(e_i)\}),$$

and we refer to the graph formed by Z_p together with the set of edges in p as the **associated graph** of p. Observe that the associated graph of p is not necessarily a subgraph since it does not necessarily contain all of the edges whose source and sink vertices are members of Z_p. Given a finite path $p = e_1, e_2, \ldots, e_n$, we say that p is *directed from* $src(e_1)$ *to* $snk(e_n)$. A path that is directed from some vertex to itself is called a **cycle** or a **directed cycle**, and a **fundamental cycle** is a cycle of which no proper subsequence is a cycle. A directed multigraph is **acyclic** if it contains no cycles. Finally, if e is the only edge directed from $src(e)$ to $snk(e)$, then we occasionally denote e by $src(e) \to snk(e)$.

If $(p_1, p_2, \ldots, p_k)$ is a finite sequence of finite paths such that $p_i = (e_{i,1}, e_{i,2}, \ldots, e_{i,n_i})$, for $1 \le i \le k$, and $snk(e_{i,n_i}) = src(e_{i+1,1})$, for $1 \le i \le (k-1)$, then we define

$$\langle (p_1, p_2, \ldots, p_k) \rangle \equiv (e_{1,1}, \ldots, e_{1,n_1}, e_{2,1}, \ldots, e_{2,n_2}, \ldots, e_{k,1}, \ldots, e_{k,n_k}).$$

Clearly, $\langle (p_1, p_2, \ldots, p_k) \rangle$ is a path from $src(e_{1,1})$ to $snk(e_{k,n_k})$. If there is a path from $v_i \in V$ to $v_j \in V$, then we say that v_i is an **ancestor** of v_j, and v_j is a **descendant** of v_i. If v_i is neither a descendant nor an ancestor of v_j, we say that v_i is **independent of** v_j. Given a directed multigraph and an vertex v_j in this graph, $ancs(v_j)$ denotes the set consisting of v_j and the set of ancestors of v_j, $desc(v_j)$ denotes the set consisting of v_j and the set of descendants of v_j, and $independent(v_j)$ denotes the set of vertices independent of v_j. Finally, an edge is **transitive** if there exists another path between the source and sink actors of the transitive edge.

A sequence of vertices $(v_1, v_2 \ldots, v_k)$ is a *chain* that joins v_1 and v_k if v_{i+1} is adjacent to v_i for $i = 1, 2, \ldots, (k-1)$. We say that a directed multigraph is **connected** if for any pair of distinct members A, B of Z, there is a chain that joins A and B. Given a directed multigraph $G = (V, E)$, there is a unique partition (unique up to a reordering of the members of the partition) $V_1, V_2, \ldots, V_n$ such that for $1 \le i \le n$, $subgraph(V_i)$ is connected; and for each $e \in E$, $src(e), snk(e) \in V_j$ for some j. Thus, each V_i can be viewed as a maximal connected subset of V, and we refer to each V_i as a **connected component** of G. Depth-first search can be used to find the connected components of a directed multigraph in time that is linear in the number of vertices and edges.

A directed multigraph (V,E) is **strongly connected** if for *each* pair of distinct vertices v_1, v_2, there is a path directed from v_1 to v_2. We say that a subset V' of vertices in V is strongly connected if $subgraph(V',(V,E))$ is strongly connected. A **strongly connected component** of (V,E) is a strongly connected subset $V' \subseteq V$ such that no strongly connected subset of V properly contains V'.

Given a directed multigraph (V, E), a vertex v of (V, E) is a **root vertex** of (V, E) if there is no edge e in (V, E) such that $snk(e) = v$. A **root strongly connected component** of (V, E) is a strongly connected component V' of (V, E) such that $\{e \in E | (src(e) \notin V' \textbf{ and } snk(e) \in V')\} = \varnothing$. Finally, if V' is a connected component of (V,E), then $subgraph(V')$ is called a **connected component subgraph** of (V,E); similarly, if V' is a strongly connected component of (V,E), then $subgraph(V')$ is a **strongly connected component subgraph** of (V,E).

A **topological sort** of an acyclic directed multigraph (V,E) is an ordering $v_1, v_2, \ldots, v_{|V|}$ of the members of V such that for each $e \in E$, $((src(e) = v_i) \textbf{ and } (snk(e) = v_j) \Rightarrow (i < j))$; that is, the source vertex of each edge occurs earlier in the ordering than the sink vertex. An acyclic directed multigraph is said to be **well-ordered** if it has only one topological sort, and we say that an n-vertex well-ordered directed multigraph is **chain-structured** if it has $(n-1)$ edges. Thus, for a chain-structured directed multigraph, there are orderings $v_1, v_2, \ldots, v_n$, and $e_1, e_2, \ldots, e_{n-1}$ of the vertices and edges, respectively, such that each e_i is directed from v_i to v_{i+1}.

In the remainder of this thesis, by a "graph" or a "directed graph", we mean a directed multigraph, unless otherwise stated.

2.1.2 Computational Complexity

When discussing the complexity of algorithms, we will use the standard O, Ω, and, Θ notation. A function $f(x)$ is $O(g(x))$ if for sufficiently large x, $f(x)$ is bounded above by a positive real multiple of $g(x)$. Similarly, $f(x)$ is $\Omega(g(x))$ if $f(x)$ is bounded below by a positive real multiple of $g(x)$ for sufficiently large x. Finally, $f(x)$ is $\Theta(g(x))$ if it is both $O(g(x))$ and $\Omega(g(x))$.

Let x represent the size of an input instance to some algorithm A. Suppose that the running time of this algorithm, $f(x)$, is represented as $\Theta(g(x))$. If the function $g(x)$ is a polynomial function of x, then the algorithm is said to run in **polynomial time**. If $g(x)$ is an exponential function of x, then the algorithm is said to run in **exponential time**. Given a problem instance, the solution returned by the algorithm is called the **certificate**.

Not all combinatorial problems have polynomial time algorithms. The class of problems that can be solved in polynomial time is called the class **P**. A more general class of problems, the class **NP**, are those that have the following property: a certificate (solution) to the problem can be *verified* in polynomial time. There are a great many combinatorial problems that we can easily show to be in **NP** but cannot show to be in **P** because we are unable to come up with algorithms that solve the problem in polynomial time. All known algorithms for these problems take exponential time. For reasons too technical to go into here, if **NP** $\neq$ **P**, then there is a subset of **NP** disjoint from **P** called the set of **NP-complete problems**. This class of problems has the property that if any problem in this class is shown to have a polynomial time algorithm, then **NP** $=$ **P**. We use standard techniques of polynomial-time reductions to establish NP-completeness of certain problems.

2.2 Synchronous Dataflow

Recall that in a dataflow graph, the data values on the edges are referred to as **tokens**. Formally, an SDF graph is a directed multigraph in which each edge α has three properties in addition to $src(\alpha)$ and $snk(\alpha)$:

- $del(\alpha)$, a nonnegative integer that gives the number of initial tokens associated with α. The number of initial tokens $del(\alpha)$ is also called the **delay** of an edge because it can induce an offset in the execution of the sink actor in relation to the source actor.

- $prd(\alpha)$, a positive integer that indicates the number of tokens produced onto the channel corresponding to α by each execution of the computation corresponding to $src(\alpha)$.

- $cns(\alpha)$, a positive integer that represents the number of tokens consumed from α by each execution of $snk(\alpha)$.

We refer to a vertex of an SDF graph as an **actor**, and given an SDF graph G, we represent the set of actors and the set of edges in G by $actors(G)$ and $edges(G)$, respectively. If for each $\alpha \in edges(G)$, $prd(\alpha) = cns(\alpha) = 1$, then we say that G is a **homogeneous** SDF (HSDF) graph.

Conceptually, each edge in G, corresponds to a FIFO (first-in, first-out) queue that buffers the tokens that pass through the edge. The FIFO queue associated with an edge is called a **buffer** for that edge, and the process of maintaining the queue of tokens on a buffer is referred to as **buffering**. Each buffer contains an initial number of tokens equal to the delay on the associated edge. A **firing** of an actor in G corresponds to removing $cns(\alpha)$ tokens from the head of the buffer for each input edge α, and appending $prd(\beta)$ tokens to the buffer for each output edge β. Thus, a firing is only possible if for each input edge α, there are at least $cns(\alpha)$ tokens on the corresponding buffer. After a sequence of 0 or more firings, we say that an actor is **fireable** if there are enough tokens on each input buffer to fire the actor. A **schedule** for G is a sequence $S = f_1 f_2 f_3 \ldots$, which can be finite or infinite, of actors in G. Each term f_i of this sequence is called an **invocation** of the corresponding actor in the schedule; and for each actor N, we denote the jth invocation of N in the schedule by N_j, and we call j the **invocation number** of N_j. The schedule that consists of no invocations — the empty sequence — is called the **null schedule**. For each i, f_i is said to be an **admissible firing** if it is fireable immediately after $f_1, \ldots, f_{i-1}$ have fired in succession. The schedule $S = f_1 f_2 f_3 \ldots$ is an **admissible schedule** for G if f_i is an admissible firing for each i. The process of successively firing the invocations in an admissible schedule is called **executing** the schedule, and if a schedule is executed repeatedly, each repetition of the schedule is called a **schedule period** of the execution.

If e is an SDF edge, then the **delayless version** of e is an edge e' such that $e' = e$ if $del(e) = 0$, and if $del(e) \neq 0$, then e' is the edge defined by $src(e') = src(e)$, $snk(e') = snk(e)$, $cns(e') = cns(e)$, $prd(e') = prd(e)$, and $del(e') = 0$. If $G = (V, E)$ is an SDF graph, then G is **delayless** if $del(e) = 0$ for all $e \in E$, and the **delayless version** of G is the SDF graph defined by (V, E'), where $E' = \{\text{the delayless version of } e | e \in E\}$. In words, the delayless version of G is the graph that results from setting the delays on all edges to zero.

Consider the simple SDF graph in Figure 2-1. Each edge is annotated with the number of tokens produced by its source actor and the number of tokens consumed by its sink — for example, actor A produces three tokens per firing on its output edge and B consumes two tokens from its input edge. The "2D" next to the edge directed from A to B indicates that this edge has a delay of 2. Now consider the schedule $BACBA$ for this example. As we fire the invocations in the schedule, we can represent the **state** of the system — the number of tokens queued on the buffers — with an ordered pair whose first and second members are, respectively, the number of tokens on the edge $A \to B$ and the number of tokens on the edge $B \to C$. Then, since there is a delay of 2 on the left side edge, the initial state of the system is $(2,0)$. Thus, at the start, B is fireable, and as it fires, two tokens are removed from the left edge and one token is appended to the right edge, so the state becomes $(0,1)$. Since A has no input edges, it can be fired at any time. Hence the second invocation of the schedule is an admissible firing and its firing leads to the state

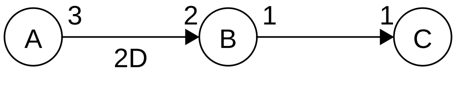


Figure 2-1 A simple SDF graph.

$(3,1)$. It is easily verified that the remaining three invocations in the schedule are admissible firings, and the sequence of buffer states that results from these remaining firings is $(3,0), (1,1), (4,1)$. Thus, $BACBA$ is an admissible schedule for the SDF graph in Figure 2-1. In contrast, the slightly different schedule $BACBB$ is not admissible, since only one token resides on the input edge of B prior to the third invocation of B, so B is not fireable.

If $S = A_1A_2A_3\ldots$ is not an admissible schedule, then some A_i is not fireable immediately after its antecedents have fired. Thus, there is at least one edge α such that (1) $snk(\alpha) = A_i$ and (2) the buffer associated with $snk(\alpha)$ contains less than $cns(\alpha)$ tokens just prior to the ith firing in S. For each such α, we say that S **terminates on α at invocation A_i**. Clearly then, a schedule is admissible if and only if it does not terminate on any edge.

Given a schedule S and an actor A, we define $inv(A, S)$ to be the number of times that S invokes A. For example, $inv(A, BACBA) = 2$.

We say that a finite schedule S is a **periodic schedule** if it invokes each actor at least once and produces no net change in the system state (i.e, the number of tokens on each edge is the same after S has executed as before) — for each edge α, $(inv(src(\alpha), S)) \times prd(\alpha) = (inv(snk(\alpha), S)) \times cns(\alpha)$. For example for the SDF graph in Figure 2-1, we saw that if the initial state is $(2,0)$, the state after executing the schedule $BACBA$ is $(4,1)$. Thus this schedule produces a net change of $+2$ tokens on edge $A \to B$ and $+1$ token on $B \to C$, so this schedule is not periodic.

Suppose that $\mathbf{b}$ is a vector of positive integers indexed by the actors in a connected SDF graph G. Non-connected SDF graphs can be analyzed by examining each connected component separately. By definition, a schedule that invokes each actor A $\mathbf{b}(A)$ times is a periodic schedule if and only if for each edge α in G,

$$\mathbf{b}(src(\alpha)) \times prd(\alpha) = \mathbf{b}(snk(\alpha)) \times cns(\alpha). \tag{2.1}$$

This system of equations in the set of variables $\{\mathbf{b}(A) | (A \in actors(G))\}$ — consisting of one equation for each edge in G — is known as the system of **balance equations** for G. Clearly, a periodic schedule exists for G if and only if the balance equations have a solution whose components are all positive integers[1]. The balance equations can be expressed more compactly in matrix-vector form as

$$\Gamma\mathbf{b} = \mathbf{0}, \tag{2.2}$$

where Γ, called the **topology matrix** of G, is a matrix whose rows are indexed by the edges in G and whose columns are indexed by the actors in G, and whose entries are defined by

$$\Gamma(\alpha,A) = \begin{cases} prd(\alpha), \text{ if } A = src(\alpha) \\ -cns(\alpha), \text{ if } A = snk(\alpha) \\ 0, \text{otherwise} \end{cases} \tag{2.3}$$

This formulation assumes that G does not contain any **self-loops**, edges whose source and sink vertices are identical. In such a case, there is a vertex α such that $src(\alpha) = snk(\alpha)$, and therefore equation 2.3 is self contradictory. In an SDF graph, a self-loop edge α precludes the existence a periodic schedule if $prd(\alpha) \neq cns(\alpha)$; otherwise it has no effect on the existence of a periodic schedule, and thus it can be ignored in this analysis.

A periodic schedule exists if and only if equation 2.2 has a solution $\mathbf{b}$ where each element of the vector $\mathbf{b}$ is a positive integer. From equation 2.2, it is evident that this requires that the topology matrix Γ not have full rank.

We have the following theorem from [Lee86].

Theorem 2-1: A connected SDF graph with n actors has a periodic schedule if and only if its topology matrix Γ has rank $n-1$. Moreover, if its topology matrix Γ has rank $n-1$, then there exists a unique smallest integer solution $\mathbf{q}$ to the balance equations $\Gamma\mathbf{q} = \mathbf{0}$. Moreover, the entries in the vector $\mathbf{q}$ are coprime.

The reason that the rank of the topology matrix of a connected SDF graph with n actors cannot be less than $n-1$ is that such a graph must have at least $n-1$ edges. Each of these edges contributes a linearly independent row to the topology matrix. Adding more

[1] Recall that in our definition of *periodic schedule*, we do not require admissibility — a periodic schedule need not be admissible.

edges to the graph adds rows to the topology matrix; however, adding rows to a matrix cannot decrease its rank. Hence, the rank is at least $n - 1$.

Notice that the graph may not have an *admissible* schedule. This will depend on the number of delays on edges in a directed cycle. The topology matrix, and hence the existence of a periodic schedule, does not depend on the delays in an SDF graph. Facts 2-1-2-3 summarize some key properties that follow immediately from theorem 2-1.

Fact 2-1: A positive-integer vector $\mathbf{q}$ is the repetitions vector of a connected SDF graph if and only if its components are coprime and it satisfies the balance equations, $\Gamma\mathbf{q} = \mathbf{0}$.

Fact 2-2: Any positive-integer vector that satisfies the balance equations is a positive-integer multiple of the repetitions vector.

Fact 2-3: A schedule S for a connected SDF graph G is periodic if and only if the repetitions vector $\mathbf{q}_G$ exists and there exists a positive integer J_0 such that S invokes each actor A exactly $J_0\mathbf{q}_G(A)$ times.

The positive integer J_0 in Fact 2-3 is called the **blocking factor** of the associated schedule. If S is a periodic schedule, we denote the blocking factor of S by $J(S)$, and if $J(S) = 1$ we say that S is a **minimal** periodic schedule.

An example of a connected SDF graph that does not have a periodic schedule is shown in Figure 2-2(a). The topology matrix for this SDF graph is

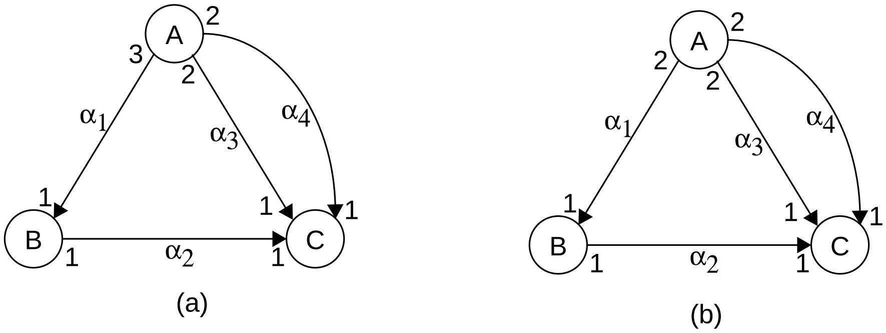


Figure 2-2 (a) An SDF graph that does not have a periodic schedule.
(b) A slightly modified version that has a periodic schedule.

$$\Gamma = \begin{bmatrix} 3 & -1 & 0 \\ 0 & 1 & -1 \\ 2 & 0 & -1 \\ 2 & 0 & -1 \end{bmatrix}, \qquad (2.4)$$

where each α_i corresponds to the ith row and the ith vertex (in alphabetical order) corresponds to the ith column. Observe that the bottom two rows of Γ are identical, and the top three rows form a square matrix whose determinant is nonzero. Thus, the matrix contains three linearly independent rows, so it has full rank, and there is no nontrivial solution to 2.2.

To understand what is "defective" about this graph, observe that for each firing of A, three firings of B are required to return edge α_1 to its initial state of having no tokens queued in its buffer, and then three firings of C are required to return α_2 to its initial state. However, since α_3 is an input edge of C and A produces only two tokens per firing on α_3, only two firings of C are possible for each firing of A. Thus, any infinite admissible sequence of firings for this graph will produce an unbounded token accumulation on α_1, α_2, or both.

If we change $prd(\alpha_1)$ to 2, the resulting SDF graph, shown in Figure 2-2(b), has a periodic schedule. The topology matrix of this new SDF graph is

$$\Gamma' = \begin{bmatrix} 2 & -1 & 0 \\ 0 & 1 & -1 \\ 2 & 0 & -1 \\ 2 & 0 & -1 \end{bmatrix}. \qquad (2.5)$$

It is easily verified that the first two rows of Γ' are linearly independent, and each of the third and fourth rows is the sum of the first two rows. Thus, the rank of Γ' is 2, one less than the number of actors, so positive-integer solutions to (2.2) exist, and thus the repetitions vector exists. The repetitions vector for Figure 2-2(b) is given by

$$\mathbf{q}(A, B, C) = (1, 2, 2)^T. \qquad (2.6)$$

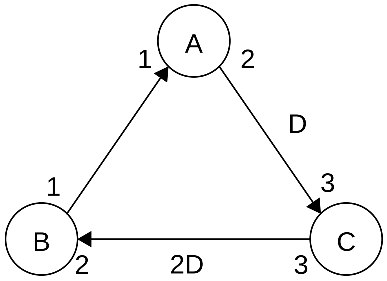


Figure 2-3 An SDF graph that has a repetitions vector but does not have an admissible schedule.

From (2.6), we see that $ABCBC$ and $ABBCC$ are minimal periodic schedules, and $ABABBCBCCC$ is a periodic schedule having blocking factor 2. All three of these schedules are admissible.

This thesis is primarily concerned with schedules that are both periodic and admissible, and we refer to such schedules as **valid** schedules. An SDF graph is **consistent** if and only if it has a valid schedule, and we say that an SDF graph is **sample rate consistent** if it has a periodic schedule. Thus, for SDF graphs, consistency implies sample rate consistency, but the converse is not true: a sample rate consistent SDF graph that is deadlocked is not consistent.

Clearly, an SDF graph is consistent if and only if each connected component subgraph is consistent, and a necessary condition for a connected SDF graph to be consistent is that the topology matrix does not have full rank. However, for an admissible periodic schedule to exist, an SDF graph must also have a sufficient amount of delay in each fundamental cycle. For example, consider the SDF graph in Figure 2-3. The repetitions vector for this graph is given by

$$\mathbf{q}(A, B, C) = (3, 3, 2)^T, \tag{2.7}$$

and thus periodic schedules exists. However, one can easily verify that there are only five possible non-null admissible schedules for this SDF graph — B, BA, BAC, $BACB$, and $BACBA$. Since none of these five schedules contains enough invocations for a periodic schedule, we see that a valid schedule does not exist. If we increase delay on the output edge of A from one to two, valid schedules, such as the periodic schedule $BACBACBA$, exist.

Associated with any connected, consistent SDF graph G, given a positive integer blocking factor J, there is a unique directed graph, called an **acyclic precedence graph (APG)**, that specifies the precedence relationships between actor invocations throughout J successive minimal schedule periods for G [Lee87]. Each vertex of the APG corresponds to an actor invocation. There is an edge directed from the vertex corresponding to invocation A_i to the vertex corresponding to B_j if and only if at least one token produced by A_i is consumed by B_j. As a simple example, Figure 2-4 below shows the APG for Figure 2-1 and blocking factor 1. See [Sih91] for an efficient algorithm that systematically constructs the APG.

Finally, given a sample rate consistent, connected SDF graph G and an edge α in G, we denote the total number of tokens consumed by $snk(\alpha)$ in a minimal schedule period by $TNSE(\alpha, G)$; that is

$$TNSE(\alpha, G) = \mathbf{q}_G(snk(\alpha)) \times cns(\alpha).$$

Since in a periodic schedule, the number of tokens produced on an edge equals the number of tokens consumed, we also have that $TNSE(\alpha, G) = \mathbf{q}_G(src(\alpha)) \times prd(\alpha)$. If G is understood from context, then we may suppress the second argument and write $TNSE(\alpha)$ in place of $TNSE(\alpha, G)$.

2.3 Computing the Repetitions Vector

The repetitions vector can be computed efficiently by applying depth-first search; see [Bhat96a] for an algorithm. Assuming the production and consumption parameters on

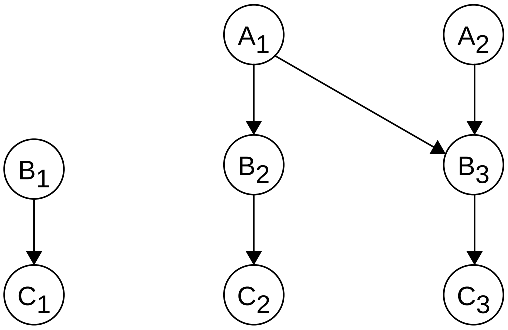


Figure 2-4 The acyclic precedence graph for Figure 2-1 and unity blocking factor.

the edges are bounded — so that computing the least common multiple of two numbers is an elementary operation — the algorithm in [Bhat96a] has time complexity that is linear in the number of actors and edges in the input SDF graph (that is, its running time is given by $\Theta(|edges(G)| + |actors(G)|)$.

2.4 Constructing a Valid Schedule

If a connected SDF graph is consistent and the repetitions vector is computed, a valid schedule can be constructed. In [Lee87], Lee and Messerschmitt define a class of scheduling algorithms, called *class-S algorithms*, that construct valid schedules given a positive integer multiple of the repetitions vector $\mathbf{r}$ ($\mathbf{r} = k\mathbf{q}$ for some positive integer k). A class-S algorithm maintains the state of the system as a vector $\mathbf{b}$ that is indexed by the edges in the input SDF graph. A class-S algorithm is any algorithm that repeatedly schedules fireable actors, updating $\mathbf{b}$ as each actor is fired, until either no actor is fireable or until all actors have been scheduled exactly the number of times specified by the corresponding component of $\mathbf{r}$. Thus, once an actor A has been scheduled $\mathbf{r}(A)$ times, a class-S algorithm does not schedule A again. Lee and Messerschmitt show that a class-S algorithm constructs a valid schedule if and only if the SDF graph in question is consistent [Lee87]. A simple implementation of the above idea of constructing a valid schedule has a running time given by $O(I f_i f_o + |E|)$, where $G = (V, E)$ is the input SDF graph,

$$I = \sum_{A \in actors(G)} \mathbf{r}(A),$$

and f_i (f_o) is the maximum over all actors of the number of input (output) edges that are incident to any actor.

2.5 Determining Whether an SDF Graph Deadlocks

While deadlock in an SDF graph can be detected by simulation, the number of steps required in the simulation can be as high as $\sum \mathbf{q}(v)$. This number is not a polynomial function of the size of the input graph; hence, detecting deadlock in this manner is not an efficient method. However, we can derive a sufficient condition from the Karp-Miller integer programming formulation in [Karp66] for a simple SDF cycle in the following manner. Firstly, we state the theorem from [Karp66] as it applies to SDF graphs (recall that SDF graphs are a special case of computation graphs). Consider the SDF cycle in figure 2-5.

Theorem 2-2: [Karp66] The cycle in figure 2-5 deadlocks if and only if the following integer program has a non-negative integer solution for the $x(i)$, $1 \le i \le n$:

$$x(1) \ge \frac{D_1}{W_1} + \frac{(1 - W_1)}{W_1} + \frac{U_n}{W_1}x(n)$$

$$x(2) \ge \frac{D_2}{W_2} + \frac{(1 - W_2)}{W_2} + \frac{U_1}{W_2}x(1) \tag{2.8}$$

$$\dots$$

$$x(n) \ge \frac{D_n}{W_n} + \frac{(1 - W_n)}{W_n} + \frac{U_{n-1}}{W_n}x(n-1)$$

From the above, we can derive the following corollary by observing that SDF graphs of interest to us are always sample rate consistent. For any cycle, this means that the product of the numbers consumed around the cycle has to be equal to the product of the numbers produced around the cycle.

Corollary 2-1: If the inequality

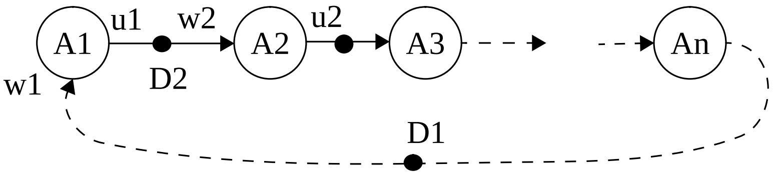


Figure 2-5 A simple SDF cycle.

$$\frac{U_1U_2\dots U_{n-1}}{W_2\dots W_n}\left(\frac{D_1}{W_1}+\frac{1-W_1}{W_1}\right)+\frac{U_2\dots U_{n-1}}{W_3\dots W_n}\left(\frac{D_2}{W_2}+\frac{1-W_2}{W_2}\right)+\dots$$
$$\dots+\left(\frac{D_n}{W_n}+\frac{1-W_n}{W_n}\right)>0 \tag{2.9}$$

holds, then the cycle does not deadlock.

Proof: Suppose that the cycle does deadlock. Then there is a solution to the system of inequalities 2.8; the left hand side of 2.9 is derived from some simple algebraic manipulation of 2.8 and the manipulation shows that it is less than or equal to 0. Hence, the contrapositive of this implies that there is no solution if it is greater than 0, and this implies that the cycle does not deadlock by Theorem 2-2. **QED**.

The corollary provides a quick algebraic test to see if the cycle does not deadlock; if the test is negative, then we have to resort to simulation to detect deadlock. In graphs that have few cycles, checking each cycle by this method might be much quicker than simulation. The Karp-Miller theorem also establishes that the problem of detecting deadlock is in NP, since a non-deterministic algorithm would simply exhibit the solution to the integer program if the graph deadlocks. The theorem guarantees that if the solution is valid, then the graph does deadlock.

2.6 Scheduling to Minimize Buffer Usage

One desirable optimization criterion is the amount of buffer space that is required by the schedule. If we assume a model of buffering where there is one buffer on every edge, then the amount of buffer space required on each edge is given by the maximum number of tokens queued on that edge during one period of the schedule. It is easy to see that different valid schedules can have vastly different buffering requirements; for the graph in Figure 2-1, the schedule $AABBBCCC$ requires 11 spaces in all, while the schedule $BCABCABC$ requires 5 spaces. This section develops a heuristic that attempts to generate a schedule that minimizes buffering requirements.

Since an SDF schedule in general can have up to

$$I \equiv \sum_{A \in \ actors(G)} \mathbf{r}(A)$$

actor appearances, algorithms that attempt to construct minimum-buffer schedules can take $O(I)$ steps or more to execute. Hence, these algorithms do not run in time that is polynomial *in the size of input SDF graph* (an algorithm would run in time polynomial in the size of the SDF graph if its running time were a polynomial function of N, the number of vertices in the graph, and $N \cdot \log(P)$ where P is the maximum of all of the produced/consumed parameters in the SDF graph). We will therefore require our algorithms for this purpose to run in time that is a polynomial function of $O(I)$, preferably a linear function. However, as the theorem below shows, the problem of computing a minimum buffer schedule for even an arbitrary homogenous SDF graph is NP-complete; hence, we have to rely on heuristics.

2.6.1 NP Completeness

Formally, the problem HSDF-MIN-BUFFER is defined as follows: Given an arbitrary homogenous SDF graph $G = (V, E)$, with arbitrary numbers of delays on each edge, and a positive integer K, is there a valid schedule for G that has a total buffering requirement of K or less, assuming that there is a buffer on every edge?

Definition 2-1: Let $D_T = \sum_{(u, v) \in E} del(u, v)$.

Observation 2-1: Any valid, minimal schedule for an HSDF graph has a buffering requirement B that satisfies $D_T \le B \le D_T + |E|$.

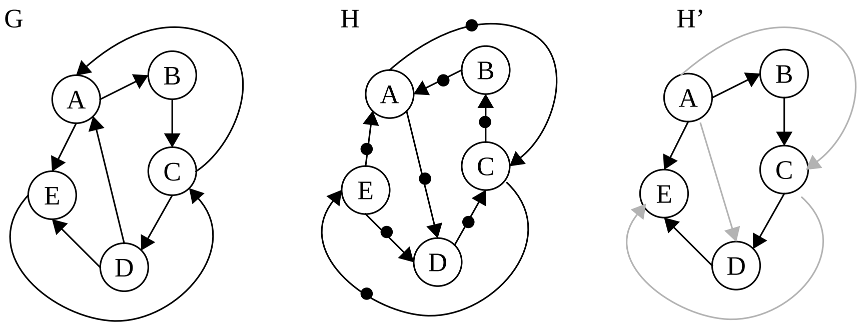
Proof: In an HSDF graph, each actor needs to be fired only once in any periodic schedule. If on any edge (u, v), $src((u, v))$ fires before $snk((u, v))$, then the maximum number of tokens queued is $del((u, v)) + 1$. If $snk((u, v))$ fires before $src((u, v))$ (provided $del((u, v)) > 0$) then the maximum number of tokens queued remains $del((u, v))$. In the worst schedule, the source actor fires before the sink actor on every edge; this increases the total amount of buffering by $|E|$. In the best conceivable schedule, the sink actor fires before the source on every edge (again, provided there is at least one delay on each edge), resulting in the lower bound of D_T.

Definition 2-2: The FEEDBACK ARC SET (FAS) problem is the following: given a directed graph $G = (V, A)$, and a positive integer K, does there exist a subset $B \subseteq A$ with $|B| \le K$, such that B contains at least one edge from every directed cycle in G? This problem is known to be NP-complete [Gare79].

Theorem 2-3: The HSDF-MIN-BUFFER problem is NP-complete.

Proof: The fact that HSDF-MIN-BUFFER is in NP is obvious. We reduce FEEDBACK ARC SET to this problem to show completeness. Let $G = (V, A)$, K be an arbitrary instance of the FAS problem. Let H be the HSDF graph constructed from G by reversing each edge in G and setting $del(e) = 1$ for each edge. We show that H contains a schedule having a buffering requirement $B \le K + |A| \Leftrightarrow$ there is a feedback arc set $A' \subseteq A, |A'| \le K$ in G. Observe that any permutation of the actors constitutes a valid schedule for H since there is a delay on every edge.

Assume that H contains such a schedule. This means that there are at most K edges in H where the source actor of the edge fires before the sink actor; on every other edge, the sink actor of the edge fires before the source actor. Construct a new delayless graph $H' = (V, E)$ in which an edge (u, v) from G is in E if and only if u fires before v in the schedule. An edge (v, u) is in E if (u, v) is an edge in G and v fires before u in the schedule. Hence, H is the same graph as G except that it has no delays and has reversed edges wherever the sink actor fires before the source actor in the schedule for H. These edges that are reversed in H' with respect to H are the same orientation as the corresponding edges in G. Hence, the set of edges in H' that have the reverse orientation with respect to edges in G, has cardinality at most K. By construction, it is clear that the schedule for H is also a valid schedule for H' since the schedule respects all of the precedence constraints in H'. Since H' is delayless, and has a valid schedule, it must be acyclic. Hence, reversing a subset of edges, with respect to G, where this subset has cardinality at most K results in an acyclic graph; this subset must be a solution to the FAS instance since the set has to include at least one edge from each directed cycle from G. Figure 2-6 shows an example.



Schedule: ABCDE

Figure 2-6 Example to illustrate proof of Theorem 2-3. A filled dot on an edge indicates a delay on that edge. The shaded edges in the H' graph show the edges that are reversed w.r.t G

Now suppose that G has a feedback arc set A' of cardinality less than or equal to K. In H, remove all of the edges corresponding to A' to form a new graph H'. The graph H' is now acyclic since at least one edge from every cycle has been removed. Construct a schedule for H' in reverse topological order; this ensures that sink actors on all of the edges fire before the source actors. This schedule is also valid for H. Moreover, in H, the schedule is such that on at most K edges can we have the situation that the source actor fires before the sink; hence, this schedule has a buffering requirement of at most $K + |A|$. ■

2.6.2 Heuristic for Minimum Buffer Scheduling

Figure 2-7 shows a simple algorithm that constructs schedules that attempt to minimize buffer usage. A simpler variant of this algorithm has been used in both Gabriel and Ptolemy programming environments for a number of years. A similar algorithm is also given by Cubric and Panangaden in [Cubr93] where they establish its optimality for a restricted set of acyclic, delayless graphs. An actor in the algorithm is **deferrable** if any one of its output edges that is not a transitive edge has at least as many tokens as consumed by the sink actor on that edge in one firing.

Examples can easily be constructed where the algorithm fails to achieve the minimum buffer usage. However, in practice, the algorithm does quite well and produces schedules that are close to optimal even if not optimal. The algorithm can be made more sophisticated by making the actor selection procedure in the `else` condition less greedy;

for example, by firing only actors that are in undirected cycles. However, it is an open problem as to whether such modifications will give an optimal algorithm for a broad class of graphs.

We close this section by proving a simple theorem about the lower bound on the amount of buffering required on any edge in an SDF graph. This result is a generalization of a result proved by researchers in the GRAPE project in [Ade94].

Theorem 2-4: For the 2-actor SDF graph depicted in Figure 2-8, the minimum amount of buffering required on the edge over all possible valid schedules is given by $a + b - c + d \bmod c$, where $c = gcd(a, b)$, if $0 \le d \le a + b - c$, and by d otherwise.

Proof: Clearly, the minimum buffer schedule for this graph is such that B is invoked whenever there are enough tokens on the edge to enable it. Also, the maximum number of tokens will be reached after a firing of A and before a firing of B. The theorem claims that $a + b - c + d \bmod c$ is the maximum number of tokens queued. To see this, assume that it is not; that in fact it is greater than this amount. Then it must be the case that the number of tokens is at least equal to $d \bmod c + ic$ where $ic > a + b - c$ (the change in the number of tokens on the edge is a multiple of c since if A has been fired x times and B has been fired y times, then the number of tokens on the edge is given by $xa - yb + d$ and this can be written as $ic + d \bmod c$.) Since $a + b - c$ is divisible by c, we get that $ic - a \ge b$. The

```
Procedure min-buf-schedule (SDF Graph G=(V,E)):
$F$ = {fireable actors};
$D$ = {deferrable($F$)};
while ($F \neq \phi$)
        if ($F \backslash D \neq \phi$)
                // fire an actor from $F \backslash D$
        else
                // fire an actor that increases total
                // number of tokens the least.
        end if
end while
```

Figure 2-7 A heuristic for generating a minimum buffer schedule.

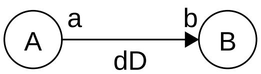


Figure 2-8 A simple 2-actor SDF graph.

maximum number of tokens always occurs after a firing of A and before a firing of B; hence, just before the maximum is reached, the number of tokens was $d \bmod c + ic - a \geq d \bmod c + b$ since $ic - a \geq b$. This shows that B was fireable at the previous step and contradicts the fact that we always fire it as soon as it becomes fireable.

It remains to show that this bound is tight and is in fact reached during the schedule. This can be seen by a state enumeration argument. Since A is fired b/c times and B is fired a/c times, the total number of states on the edge is $(a + b)/c$. Since any state that is reached is of the form $ic + d \bmod c$, and the number of states in $[0, a + b - c + d \bmod c]$ of this form is precisely $(a + b)/c$, we conclude that the state $a + b - c + d \bmod c$ is reachable and that the bound is tight (no state can be repeated since by definition a valid schedule for an SDF graph is one that returns the buffers to the initial state).

Suppose $d > a + b - c$. Initially, we can fire B enough times to get $d' = d \bmod b$ tokens on the arc. Now suppose that we were to exceed d tokens on the arc during the minimum buffer schedule. Then we would have $ic + d'$ tokens for some i, and $ic + d' > d$ or $ic > d - d'$ at some stage in the schedule. Since $d - d'$ is a multiple of b, and thus c, we have $ic + d' \geq d + c$. This number of tokens would have resulted after a firing of A; hence, before A was fired, we would have had $ic + d' - a \geq d + c - a$ tokens. Since $d > a + b - c$, $ic + d' - a > b$ and B was fireable when A was fired, contradicting the scheduling strategy. Hence, we cannot exceed d tokens under the minimum-buffer scheduling strategy. ■

2.7 Related Work: Multiprocessor Scheduling

SDF graphs can be scheduled onto multiple processors easily. In this section, we assume that the SDF graph is in fact an HSDF graph; this is not a restriction since any SDF graph can be converted[1] to an equivalent HSDF graph [Lee86]. Most forms of resource-constrained (meaning we have some fixed number of processors for example) multiprocessor scheduling (MS) problems are NP-hard; hence, we must resort to heuristics

[1] However, the size of the HSDF graph is exponentially bigger than the SDF graph. Some recent work has been done on scheduling the SDF graph onto a multiprocessor directly [Pino95b].

in general. However, there are several issues that must be dealt with when we generate multiprocessor schedules for DSP applications. The traditional metric that one tries to minimize is the makespan; this is the time by which all tasks have finished executing. But DSP applications are non-terminating; hence, we would like to generate schedules that maximize the throughput (or minimize the iteration period[1]). These two metrics give rise to two different forms of scheduling: Non-overlapped scheduling and overlapped scheduling.

It is well known that a fundamental upper bound on the throughput achievable in an HSDF graph is given by the inverse of the *maximum cycle mean* [Reit68]:

$$\lambda = MAX_{l \in \Lambda}\left\{\frac{T(l)}{D(l)}\right\} \tag{2.10}$$

where Λ is the set of all circuits in the graph, $T(l)$ is the total computation time of circuit l, and $D(l)$ is the delay count of the circuit l. A loop that achieves this maximum is called a *critical cycle*. A schedule for an HSDF graph is *rate-optimal* if the iteration-period for the schedule is equal to the maximum cycle mean (also called the *iteration-period bound*).

In non-overlapped scheduling, one only considers one iteration of the graph and applies classical MS heuristics to schedule the precedence graph. This schedule is infinitely repeated. However, since the algorithm overlooks the potential parallelism between the iterations, schedules obtained by this method will not be throughput optimal in general. There are a couple of ways in which the performance of a non-overlapped scheduler can be improved: by increasing the **blocking factor**, and by **retiming** [Leis91]. Retiming is a way of moving the delays around in the graph so that the critical path can be lowered; this can result in better schedules.

Increasing the blocking factor allows one to consider more than just one iteration of the graph while scheduling, and gives improved performance. However, an example graph is given in [Lee86a] for which no non-overlapped schedule of any finite blocking factor can achieve rate-optimal throughput. Even then, the question of choosing the “best” blocking factor was posed and left as an open problem [Lee86]. Using max-algebra techniques

[1] By an iteration, we mean the execution of one schedule period of the SDF graph.

[Bacc93], this problem is analyzed thoroughly in [Murt94b], where the asymptotic behavior of the critical path in the precedence graph of blocking factor J, as J is increased, is shown to be cyclic in the following sense: there exist constants T and ρ such that the critical path in the precedence graph of blocking factor $J + \rho$ has weight given by

$$CP(J + \rho) = CP(J) + \rho\lambda \qquad \forall J > T$$

where λ is the maximum cycle mean in the original graph, and ρ is an integer computable from the graph in the following way: a particular type of transitive graph is derived from the original HSDF graph. In this graph, the cyclicity of a maximal strongly connected component is the greatest common divisor of the lengths of all the critical cycles. The cyclicity of the graph is the least common multiple of the cyclicities of all the maximal strongly connected components. The result allows us to conclude that the problem of computing the “best” blocking factor is decidable, even though the algorithm for finding it does not run in polynomial time.

In overlapped scheduling, the inter-iteration parallelism is fully taken into account, and in the absence of resource constraints, always leads to rate-optimal schedules [Reit68]. In [Murt94b] and [Parh91], an example graph is given for which no retiming or increase in blocking factor can achieve a non-overlapped, rate-optimal schedule. Hence, overlapped scheduling is fundamentally more powerful than non-overlapped scheduling. However, much more work has been done in non-overlapped scheduling to devise heuristics based on critical path methods for many resource constrained problems; for example, minimizing the interprocessor communication [Sih91], scheduling for a fixed number of processors, and scheduling for fault tolerance. In contrast, there are few heuristics that generate overlapped schedules for these various practical scenarios; this is an interesting direction for future research.

3

Looped Schedules

In this chapter, we develop scheduling techniques for SDF graphs that jointly optimize for code-size compactness and for buffer memory usage as a secondary goal. Code-size compactness is achieved by the organization of loops in the target code. Since single appearance schedules have the maximum degree of code-compactness, the algorithms and heuristics we present generate single appearance schedules that minimize the amount of buffer memory required by the schedule. We present an extensive experimental study to demonstrate the usefulness of these techniques.

3.1 Looped Schedule Terminology and Notation

We first recall some basic concepts and terminology pertaining to uniprocessor scheduling from [Bhat94b].

Definition 3-1: Given an SDF graph G, a **schedule loop** is a parenthesized term of the form $(nT_1T_2\ldots T_m)$, where n is a positive integer, and each T_i is either an actor in G or another schedule loop. The parenthesized term $(nT_1T_2\ldots T_m)$ represents the successive repetition n times of the invocation sequence $T_1T_2\ldots T_m$. If $L = (nT_1T_2\ldots T_m)$ is a schedule loop, we say that n is the **iteration count** of L, each T_i is an **iterand** of L, and $T_1T_2\ldots T_m$ constitutes the **body** of L**.** If the body of L is empty, that is if $m = 0$, we say that L is a **null** schedule loop; except where otherwise stated, we assume that all schedule loops are non-null. A **looped schedule** is a sequence $V_1V_2\ldots V_k$, where each V_i is either an actor or a schedule loop. Since a looped schedule is usually executed repeatedly, we refer to each V_i as an **iterand** of the associated looped schedule.

When referring to a looped schedule, we often omit the "looped" qualification if it is understood from context; similarly, we may refer to a schedule loop simply as a **loop**. Given a looped schedule S, we refer to any contiguous sequence of actors and schedule loops in S (at any nesting depth) as a **subschedule** of S. For example, the schedules $(3AB)C$ and $(2B(3AB)C)A$ are both subschedules of $A(2B(3AB)C)A(2B)$, whereas $(3AB)CA$ is not. By this definition, S is a subschedule of itself, and every schedule loop in S is a subschedule of S. If the same invocation sequence appears in more than one place in a looped schedule, we distinguish each instance as a separate subschedule. For example, in $(3A(2BC)D(2BC))$, there are two appearances of $(2BC)$, and these correspond to two distinct subschedules. In this case, the content of a subschedule is not sufficient to specify it — we must also specify the lexical position, as in "the second appearance of $(2BC)$." If S_0 is a subschedule of S, we say that S_0 is **contained in** S, and we say that S_0 is **nested in** S if S_0 is contained in S and $S_0 \neq S$.

We denote the set of actors that appear in a looped schedule S by $actors(S)$, and we denote the number of times that an actor A appears in S by $appearances(A, S)$; thus,

$$actors((2(2B)(5A))) = \{A, B\},$$

$$actors(X(2Y(3Z)X)) = \{X, Y, Z\},$$

$$appearances(C, (3CA)(4BC)) = 2,$$

$$appearances(A, (2ABAC)(3A)) = 3.$$

Formally, S is a **single appearance schedule** if

$$appearances(A, S) = 1 \ \forall A.$$

As argued in [Bhat94b], if we neglect the code-size overhead associated with the loops, any single appearance schedule yields the smallest inline implementation of an SDF graph with regard to code size. Most programmable DSPs have provisions to efficiently manage loop indices and perform the loop test in hardware, without explicit software control; hence, the loop overhead is typically small. It is almost negligible compared to the alternative of code-size explosion if a non-looped schedule is used.

Given a looped schedule S and an actor A, we define $\mathit{inv}(A, S)$ to be the number of times that S invokes A. Similarly, if S_0 is a subschedule, we define $\mathit{inv}(S_0, S)$ to be the number of times that S invokes S_0. For example, if $S = A(2(3BA)C)BA(2B)$, then $\mathit{inv}(B, S) = 9$, $\mathit{inv}((3BA), S) = 2$, and $\mathit{inv}(\text{first appearance of } BA, S) = 6$. Also, we refer to the invocation sequence that a looped schedule S represents as the invocation sequence generated by S. For example, the invocation sequence generated by $S = A(2(3BA)C)BA(2B)$ is $ABABABACBABABACBABB$. When there is no ambiguity, we occasionally do not distinguish between a looped schedule and the invocation sequence that it generates.

A schedule loop is a **one-iteration** loop if its iteration count is 1. Although such loops are usually useless in the implementation of a schedule, they are useful for analyzing schedules, as will be apparent, for example, in Section 3.3. Since a one-iteration schedule loop generates the same invocation sequence as its body, replacing the loop by its body does not change the invocation sequence of an enclosing schedule. Thus, given an arbitrary looped schedule S, if we select a one-iteration loop and replace it with its body, select a one-iteration loop in the resulting schedule and replace it with its body, and repeat this process until there are no one-iteration loops remaining, we will arrive at a new schedule S' that generates the same invocation sequence as S and contains no one-iteration loops. Thus, the following fact is obvious.

Fact 3-1: Given a looped schedule S, there exists a looped schedule S' that generates the same invocation sequence as S such that S' contains no one-iteration schedule loops, and

$$\forall(A \in \mathit{actors}(S)), \mathit{appearances}(A, S') = \mathit{appearances}(A, S).$$

Given a schedule S, an invocation I is said to be **part of** a subschedule S_0 if I occurs in an invocation of S_0. For example, in the schedule $AA(2AB)BB$, invocations A_3, A_4, B_1, and B_2 are part of the subschedule $(2AB)$, whereas A_1, A_2, B_3, and B_4 are not. Given an SDF graph G, an edge α in G, a looped schedule S for G, and a nonnegative integer i, we define $P(\alpha, i, S)$ to denote the number of invocations of $\mathit{src}(\alpha)$ that precede the ith invocation of $\mathit{snk}(\alpha)$ in S; and we define $T(\alpha, i, S)$ to denote the number of tokens on α just prior to the ith invocation of $snk(\alpha)$ in an execution of S. For example, consider the SDF graph in Figure 3.1 and let α denote the edge directed from B to C. Then $P(\alpha, 2, BC(2ABC)) = 2$, the number of invocations of B that precede invocation C_2 in the invocation sequence $BCABCABC$, and $T(\alpha, 2, BC(2ABC)) = 1$.

We will occasionally need to refer to the relative lexical positions of actors in a single appearance schedule. For this purpose, we define $position(X, S)$ to be the number of actors that lexically precede X in the single appearance schedule S. Observe that no ambiguity arises in this definition since we apply it only to single appearance schedules. For example, if $S = (2(3B)(5C))(7A)$, then $position(A, S) = 2$, $position(B, S) = 0$, and $position(C, S) = 1$. Formally, we define the **lexical ordering** of a single appearance schedule S, denoted $lexorder(S)$, to be the sequence of actors $(A_1, A_2, \ldots, A_n)$ where $\{A_1, A_2, \ldots, A_n\} = actors(S)$ and $position(A_i, S) = i - 1$ for each i. Thus, $lexorder((2(3B)(5C))(7A)) = (B, C, A)$. We will apply the following obvious fact about lexical orderings.

Fact 3-2: If S is a valid single appearance schedule for a delayless SDF graph[1], then whenever X is an ancestor of Y, we have

$$position(X, S) < position(Y, S).$$

3.2 Buffering Model

Given a looped schedule S for an SDF graph G, we define the **buffer memory** required by S, denoted $buffer\_memory(S)$, to be the number of storage units required to implement the buffering for S if each buffer is mapped to a separate contiguous block of memory. We assume that every token occupies one storage unit. Quantitatively, if $max\_tokens(\alpha, S)$ denotes the maximum number of tokens that are simultaneously queued on edge α during an execution of the schedule S, we have that

$$buffer\_memory(S) = \sum_{\alpha \in edges(G)} max\_tokens(\alpha, S).$$

[1] Note that a consistent, delayless SDF graph is necessarily acyclic.

We define the **buffer memory requirement** of a schedule S, as $buffer\_memory(S)$. In Figure 3-1, if $S = BC(2ABC)$, then $max\_tokens(A \to B, S) = 4$, $max\_tokens(B \to C, S) = 1$, and $buffer\_memory(S) = 4 + 1 = 5$.

The amount of memory required for buffering may vary greatly between different schedules. For example, the schedule $(9A)(12B)(12C)(8D)$ has a buffer memory requirement of $36 + 12 + 24 = 72$, and the schedule $(3(3A)(4B))(4(3C)(2D))$ has a buffer memory requirement of $12 + 12 + 6 = 30$ for the graph in Figure 3-1.

In the model of buffering implied by the “buffer memory requirement” measure, each buffer is mapped to a contiguous and independent block of memory. This model is convenient and natural for code generation, and it is the model used, for example, in the SDF-based code generation environments described in [Ho88b, Pino95a, Ritz92]. However, perfectly valid target programs can be generated without these restrictions. For example, another model of buffering (for a chain-structured graph) is to use a shared buffer of size

$$max(\{\mathbf{q}(N_i) \times prd(N_i) \,|\, 1 \le i < K\})$$

which gives the maximum amount of data transferred on any edge in one execution of the single appearance schedule $(\mathbf{q}(N_1)N_1)(\mathbf{q}(N_2)N_2)\ldots(\mathbf{q}(N_K)N_K)$, where K is the number of vertices in the graph. Assuming that there are no delays on the graph edges, it can be shown that via proper management of pointers, such a buffer suffices. For the example graph 3-1, this would imply a buffering requirement of 36 since on edge AB, 36 samples are exchanged in the schedule $(9A)(12B)(12C)(8D)$, and this is the maximum over all edges. Moreover, the implementation of this schedule using a shared buffer would be much simpler than the implementation of a more complicated nested schedule.

But there are two problems with buffer-sharing that prevent its use as the model for evaluating the buffering cost of single appearance schedules (even for chain-structured

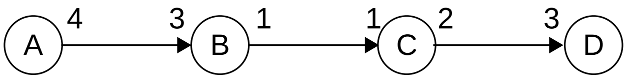


Figure 3-1 A chain-structured SDF graph.

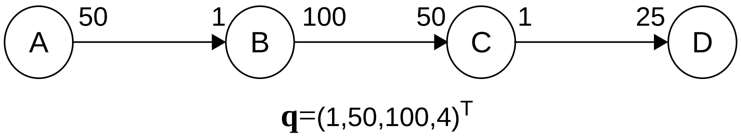


Figure 3-2 Example to illustrate the inefficiency of using shared buffers.

graphs; the general case is more complicated). Consider the graph in Figure 3-2. The shared-buffer cost for the schedule $S' = (A)(50B)(100C)(4D)$ for this graph is given by

$$max(\{1 \times 50, 50 \times 100, 100 \times 50, 4 \times 25\}) = 5000.$$

However, with a buffering model where we have a buffer on each edge, the schedule $A(50B(2C))(4D)$ requires total buffering of only 250 units. Of-course, we could attempt sharing buffers in this nested looped schedule as well, but the implementation of such sharing could be awkward.

Consider also the effect of having delays on the edges. In the model where we have a buffer on every edge, having delays does not affect the ease of implementation. For example, if we introduce d delays on edge BC in the graph in Figure 3-2, then we merely augment the amount of buffering required on that edge by d. This is fairly straightforward to implement. On the other hand, having delays in the shared buffer model causes complications because there is often no logical place in the buffer to store the initial samples since the entire buffer might be written over by the time we reach the actor that consumes the delays. For instance, consider the graph in Figure 3-3. This graph is an SDF abstraction of a possible system to do sample rate conversion between CD rates and DAT rates; see Section 3.10 for more details. The repetitions vector for this graph is given by $(147, 49, 28, 32, 160)^T$. Suppose that we were to use the shared-buffer implementation for schedule S'. We find that we need a buffer of size 224. After all of the invocations of A have been fired, the first 147 locations of the buffer are filled. Since B writes more samples than it reads, it starts writing at location 148 and writes 196 samples. When C begins execution, it starts reading from location 148 and starts writing from location 120 (120 = (148+196) mod 224). Actor C then writes 224 samples into the buffer. When D is invoked,

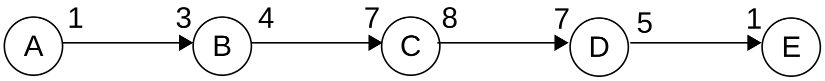


Figure 3-3 Example to illustrate the difficulty of using shared buffers with delays.

it starts reading from location 120. Hence, if there were a delay on edge CD for instance, the logical thing to do would be to have a buffer of size 225 (meaning that D would start reading from location 119) and place the delay in location 119. However, location 119 would have been written over by A; hence, it is not a safe location. This shows that handling delays in the shared buffer model can be quite awkward, and would probably involve copying over data from a “delay” buffer of some sort.

For these reasons, this chapter focuses on the buffering model associated with the “buffer memory requirement” measure. However, a simple extension of these techniques to combine the above simple model of buffer sharing with the non-shared model is presented later on for a more restricted class of graphs.

3.3 Factoring Schedule Loops

This section shows that in a single appearance schedule, common terms from the iteration counts of inner loops can be “factored” into the iteration counts of the enclosing loops. An important practical advantage of factoring is that it may significantly reduce the buffer memory requirement.

For example, consider the SDF graph in Figure 3-4. Here, $\mathbf{q}(A, B, C, D) = (100, 100, 10, 1)^T$, and one valid single appearance schedule for this graph is $(100A)(100B)(10C)D$. With this schedule, prior to each invocation of C, 100 tokens are queued on each of the input edges of C, and a maximum of 10 tokens are queued on the input edge of D. Thus 210 units of storage are required to implement the buffering of tokens for this schedule

Now observe that this schedule generates the same invocation sequence as $(1(100A)(100B)(10C))D$. The main result developed in this section allows us to factor the common divisor of 10 in the iteration counts of the three inner loops into the iteration

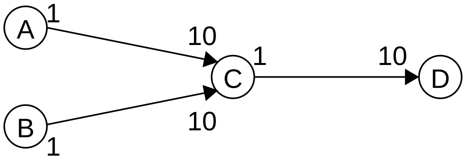


Figure 3-4 An SDF graph used to illustrate the factoring

count of the outer loop. This yields the new single appearance schedule $(10(10A)(10B)C)D$, for which at most 10 tokens simultaneously reside on each edge. Thus, this factoring application has reduced the buffer memory requirement by a factor of 7.

There is, however, a trade-off involved in factoring. For example, the schedule $(100A)(100B)(10C)D$ requires 3 loop initiations per schedule period, while the factored schedule $(10(10A)(10B)C)D$ requires 21. Thus, the run-time cost of starting loops — usually, initializing the loop indices — has increased by the same factor by which the buffer memory requirement has decreased. However, for programmable digital signal processors, the loop-start-up overhead is normally much smaller than the penalty that is paid when the memory requirement exceeds the on-chip limits. Unfortunately, we cannot in general perform the reverse of the factoring transformation; that is, moving a factor from the iteration count of an outer loop to the iteration counts of the inner loops. This *reverse factoring* transformation might be desirable in situations where minimizing the buffer memory requirement is not critical.

Figure 3-5 shows a simple SDF graph that can be used to demonstrate that unlike the factoring transformation, reverse factoring does not necessarily preserve the admissibility of a valid single appearance schedule. It is easily verified that $(10AB)$ is a valid single appearance schedule (with blocking factor 10) for this graph, while the reverse-factored derivative $(10A)(10B)$ terminates on the edge $B \rightarrow A$ at the second invocation of A.

The following theorem establishes a sufficient condition for valid application of the factoring transformation. The condition is that the sets of actors invoked by the factored loops are all mutually disjoint. Clearly, this condition is always satisfied when working with single appearance schedules, and thus a major consequence of Theorem 3-1 is that

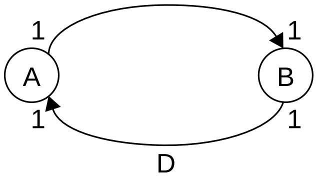


Figure 3-5 An example used to illustrate that reverse factoring is not always valid for single appearance schedules.

factoring cannot convert a valid single appearance schedule into a schedule that is not valid. The theorem also establishes that the buffering requirement cannot be increased on any edge by the factoring transformation. This is intuitively clear since, for any edge (A, B), factoring reduces the number of times A is invoked before B is invoked, thus reducing the number of tokens queued on the edge.

Theorem 3-1: Suppose that S is a valid schedule for an SDF graph G, and suppose that $L = (m(n_1S_1)(n_2S_2)\dots(n_kS_k))$ is a schedule loop in S of any nesting depth such that $(1 \le i < j \le k) \Rightarrow \mathit{actors}(S_i) \cap \mathit{actors}(S_j) = \varnothing$. Suppose also that γ is any positive integer that divides $n_1, n_2, \dots, n_k$; let L' denote the schedule loop $(\gamma m(\gamma^{-1}n_1S_1)(\gamma^{-1}n_2S_2)\dots(\gamma^{-1}n_kS_k))$; and let S' denote the schedule that results from replacing L with L' in S. Then S' is a valid schedule for G, and $\mathit{max\_tokens}(e, S') \le \mathit{max\_tokens}(e, S)\ \forall e$.

Proof: See [Bhat96a].■

Recall that the definition of *buffer memory requirement* assumes that each buffer is implemented as a separate, contiguous block of storage, and thus Theorem 3-1 does not necessarily apply under more flexible buffer implementations — such as when storage is shared between multiple buffers that are active (contain unread data) in mutually disjoint segments of time. In [Bhat94a], shared buffers and buffers that do not necessarily reside in contiguous memory locations are discussed.

3.4 Reduced Single Appearance Schedules

Definition 3-2: If Λ is either a schedule loop or a looped schedule, we say that Λ is **coprime** if not all iterands of Λ are schedule loops, or if all iterands of Λ are schedule loops, and there does not exist an integer $j > 1$ that divides all of the iteration counts of the iterands of Λ.

For example, the schedule loops $(3(4A)(2B))$ and $(10(7C))$ are both non-coprime, while the loops $(5(3A)(7B))$ and $(70C)$ are coprime. Similarly, the looped schedules $(4AB)$ and $(6AB)(3C)$ are both non-coprime, while the schedules $A(7B)(7C)$ and $(2A)(3B)$ are coprime. From the discussion in the previous section, we know that non-coprime schedules or loops may result in much higher buffer memory requirements than their factored counterparts.

Definition 3-3: Given a single appearance schedule S, we say that S is **fully reduced** if S is coprime and every schedule loop contained in S is coprime.

It can be shown that any fully reduced schedule has unit blocking factor [Bhat96a]. This implies that any schedule that has blocking factor greater than one is not fully reduced. Thus, if we decide to implement a schedule that has nonunity blocking factor, then we risk introducing a higher buffer memory requirement. The following theorem shows that we can always convert a valid single appearance schedule that is not fully reduced into a valid fully reduced schedule, and thus, we can always avoid the potential overhead associated with using non-coprime schedule loops over their corresponding factored forms.

Theorem 3-2: Suppose that $G = (V, E)$ is a consistent, connected SDF graph, and S is a single appearance schedule for G. Then there exists a valid, fully reduced schedule S' such that $lexorder(S') = lexorder(S)$, and $max\_tokens(e, S') \le max\_tokens(e, S)$, for each $e \in E$.

Proof: See [Bhat96a]. ■

3.5 Algorithms for Joint Code and Data Minimization

We want to compute a single appearance schedule that minimizes the buffer memory requirement over all valid single appearance schedules. Thus, given the model of buffer implementation defined in Section 3.2, we wish to construct a software implementation that minimizes the data memory requirement over all minimum code-size implementations. As we show later, even for chain-structured SDF graphs, the number of distinct valid single appearance schedules increases combinatorially with the number of actors, and thus exhaustive evaluation is not, in a general, a feasible means to find the single appearance schedule that minimizes the buffer memory requirement. Section 3.9 develops an efficient dynamic programming algorithm that computes an optimal hierarchy of loops given a lexical ordering of the actors. For well-ordered graphs, where there is only one topological sorting of the vertices, the schedule that results from applying the dynamic programming algorithm is guaranteed to be the optimal one. For graphs that have more than one topological sort, we develop heuristics in Section 3.16 and Section 3.19 for generating suitable topological sorts. These are nested optimally using the dynamic programming algorithm.

3.6 R-Schedules

If Λ is either a schedule loop or a looped schedule, we say that Λ satisfies the **R-condition** if one of the following two conditions holds.

(a) Λ has a single iterand, and this single iterand is an actor, *or*

(b) Λ has exactly two iterands, and these two iterands are schedule loops
having coprime iteration counts.

We call a valid single appearance schedule S an **R-schedule** if S satisfies the R-condition, and every schedule loop contained in S satisfies the R-condition.

In [Murt94a] it is shown that in a delayless chain-structured SDF graph, whenever a valid single appearance schedule exists, an R-schedule can be derived whose buffer memory requirement is no greater than that of the original schedule. This result is easily generalized to give the following theorem for arbitrary consistent SDF graphs.

Theorem 3-3: Suppose that $G = (V, E)$ is a consistent SDF graph and S is a valid single appearance schedule for G. Then there exists an R-schedule S_R for G such that $max\_tokens(e, S_R) \le max\_tokens(e, S)$ for all $e \in E$, and $lexorder(S_R) = lexorder(S)$.

Proof: See [Bhat96a] or [Murt94c].

3.7 The Buffer Memory Lower Bound for Single Appearance Schedules

Given a consistent SDF graph G, there is an efficiently computable upper and lower bound on the buffer memory requirement over all valid single appearance schedules. The lower bound can be derived easily by examining a generic two-actor SDF graph, as shown in Figure 3-6(a). From the balance equations it is easily verified that the repetitions vector for this graph is given by $\mathbf{q}(A, B) = (q/g, p/g)$, where $g \equiv gcd(\{p, q\})$, and that if $d < (pq)/g$, then the only R-schedule for this graph is $S_1 = ((q/g)A)((p/g)B)$. From Theorem 3-3 it follows that if $d < (pq)/g$, then $max\_tokens((A, B), S_1) = ((pq)/g + d)$ is a lower bound for the buffer memory requirement of the graph in Figure 3-6(a). Similarly, if $d \geq (pq)/g$, then there are exactly two R-schedules — S_1 and $S_2 = ((p/g)B)((q/g)A)$. Since $max\_tokens((A, B), S_2) = d$, we obtain d as a lower bound for the buffer memory requirement. Thus, given a valid single appearance schedule S for Figure 3-6(a), we have that

$$\left(d < \frac{pq}{g}\right) \Rightarrow \left(max\_tokens((A, B), S) \geq \left(\frac{pq}{g} + d\right)\right), \text{ and}$$

$$\left(d \geq \frac{pq}{g}\right) \Rightarrow (max\_tokens((A, B), S) \geq d). \qquad (3.1)$$

Furthermore, if (A, B) is an edge in a general SDF graph, it is easy to show that the projection of a valid schedule S onto $\{A, B\}$, which is a valid schedule for $subgraph(\{A, B\})$, always satisfies

$$max\_tokens((A, B), projection(S, \{A, B\})) = max\_tokens((A, B), S). \qquad (3.2)$$

It follows that the lower bound defined by (3.1) holds whenever (A, B) is an edge in a consistent SDF graph G, S is a valid single appearance schedule for G, $prd((A, B)) = p, cns((A, B)) = q$, and $g = gcd(\{p, q\})$. We have motivated the following definition.

Definition 3-4: Given an SDF edge e, we define the **buffer memory lower bound**

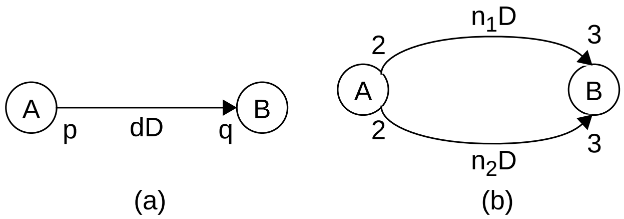


Figure 3-6 Examples used to develop the buffer memory lower bound.

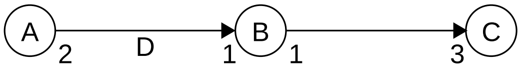


Figure 3-7 An SDF graph that does not have a BMLB schedule.

(BMLB) of e, denoted $BMLB(e)$, by

$$BMLB(e) = \begin{cases} (\eta(e) + del(e)) \text{ if } (del(e) < \eta(e)) \\ (del(e)) \text{ if } (del(e) \geq \eta(e)) \end{cases}, \text{ where}$$

$$\eta(e) = \frac{prd(e)cns(e)}{gcd(\{prd(e), cns(e)\})}$$

If $G = (V, E)$ is an SDF graph, then

$$\left(\sum_{e \in E} BMLB(e)\right)$$

is called the BMLB of G, and a valid single appearance schedule S for G that satisfies $max\_tokens(e, S) = BMLB(e)$ for all $e \in E$ is called a **BMLB schedule** for G.

In Figure 3-7, we see that $BMLB((A, B)) = 3$, and $BMLB((B, C)) = 3$. Thus, to implement any single appearance schedule for this graph, at least three memory words will be required to implement the edge (A, B), and at least three words will be required for (B, C). Furthermore, a valid single appearance schedule for Figure 3-7 is a BMLB schedule if and only if its buffer memory requirement equals 6. It is easily verified that only two R-schedules for Figure 3-7 exist — $(3A(2B))(2C)$, and $(3A)(2(3B)C)$; the associated buffer memory requirements are $3 + 6 = 9$ and $7 + 3 = 10$, respectively. Thus, a BMLB schedule does not exist for Figure 3-7.

In contrast, the SDF graph shown in Figure 3-8 has a BMLB schedule. This graph results from simply interchanging the production and consumption parameters of edge (B, C) in Figure 3-7. Here, $\mathbf{q}(A, B, C) = (1, 2, 6)$, the BMLB values for both edges are

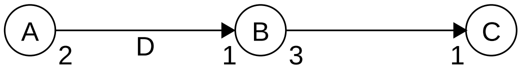


Figure 3-8 An SDF graph that has a BMLB schedule.

again identically equal to 3, and $A(2B(3C))$ is a valid single appearance schedule whose buffer memory requirement achieves the sum of these BMLB values.

The following fact is a straightforward extension of the development of the BMLB.

Fact 3-3: Suppose that G is an SDF graph that consists of two vertices A, B and $n \geq 1$ edges $e_1, e_2, \ldots, e_n$ directed from A to B. Then (a) if $del(e_i) \geq \eta(e_i)$ for all $i \in \{1, 2, \ldots, n\}$, then $(\mathbf{q}_G(B)B)(\mathbf{q}_G(A)A)$ is a BMLB schedule for G; (b) otherwise, $(\mathbf{q}_G(A)A)(\mathbf{q}_G(B)B)$ is an optimal schedule — that is, it minimizes the buffer memory requirement over the two valid minimal single appearance schedules — for G, and it is a BMLB schedule if and only if $del(e_i) < \eta(e_i)$ for $1 \leq i \leq n$[1].

For example, in Figure 3-6(b), let e_1 denote the upper edge, and let e_2 denote the lower edge. Then $\eta(e_1) = \eta(e_2) = 6$, and $(2B)(3A)$ is a BMLB schedule if $del(e_1), del(e_2) \geq 6$. Similarly, if $del(e_1), del(e_2) < 6$, then it is easily verified that $(3A)(2B)$ is a BMLB schedule. However, if $del(e_1) < 6$ and $del(e_2) \geq 6$, then $(3A)(2B)$ is optimal, but is not a BMLB schedule since in this case $max\_tokens(e_2, (3A)(2B)) = (del(e_2) + 6)$, while $BMLB(e_2) = del(e_2)$.

Fact 3-4: If $G = (V, E)$ is a connected, consistent, acyclic SDF graph, $del(e) < \eta(e)$ for all $e \in E$, and S is a BMLB schedule for the delayless version of G, then S is a BMLB schedule for G.

Proof: Let G' denote the delayless version of G. If S is a BMLB schedule for G', then S is a valid schedule for G that satisfies $max\_tokens(e, S, G) = max\_tokens(e, S, G') + del(e)$ for all $e \in E$. It follows from Definition 3-4 that S is BMLB schedule for G. ■

Fact 3-5: If G is a connected, consistent SDF graph and e is an edge in G, then

$$\eta(e) = \frac{TNSE(e, G)}{\rho_G(src(e), snk(e))}.$$

[1] Note that in this case, $(\mathbf{q}_G(A)A)(\mathbf{q}_G(B)B)$ is the only valid minimal single appearance schedule.

Recall that $TNSE(e, G)$ is the total number of samples exchanged in one complete cycle of a minimal schedule on edge e in graph G, and $\rho_G(src(e), snk(e))$ is the gcd of the repetitions of $src(e)$ and $snk(e)$.

Proof: From the balance equations,

$$\frac{TNSE(e, G)}{\rho_G(src(e), snk(e))} = \frac{\mathbf{q}_G(src(e))prd(e)}{gcd(\{\mathbf{q}_G(src(e)), \mathbf{q}_G(snk(e))\})}$$

$$= \frac{\mathbf{q}_G(src(e))prd(e)}{gcd(\{\mathbf{q}_G(src(e)), \mathbf{q}_G(src(e))(prd(e)/cns(e))\})}.$$

Multiplying the numerator and denominator of this last quotient by $cns(e)$, and recalling that $gcd(ka, kb) = kgcd(a, b)$, we obtain the desired result. ■

We conclude this section by defining an obvious, efficiently computable upper bound for single appearance schedules that have unit blocking factor. Clearly, if $G = (V, E)$ is a connected, consistent SDF graph, and S is a unit blocking factor single appearance schedule for G, we have

$$buffer\_memory(S) \le \sum_{e \in E} (TNSE(e) + del(e)).$$

We refer the RHS of this inequality as the **buffer memory upper bound (BMUB)** for G.

In Figure 3-8, $\mathbf{q}(A, B, C) = (1, 2, 6)$, and the BMUB for this graph is 9.

3.8 The Number of R-Schedules

Now let ε_n denote the number of R-schedules for an n-actor chain-structured SDF graph. Trivially, for a 1-actor graph there is only one schedule obtainable by the recursive scheduling process, so $\varepsilon_1 = 1$. For a 2-actor graph, there is only one edge, and thus only one choice for i, $i = 1$. Since for a 2-actor graph, $left(1)$ and $right(1)$ both contain only one actor, we have $\varepsilon_2 = \varepsilon_1 \times \varepsilon_1 = 1$. For a 3-actor graph, $left(1)$ contains 1 actor and $right(1)$ contains 2 actors, while $left(2)$ contains 2 actors and $right(2)$ contains a single actor. Thus,

$$\begin{aligned}
\varepsilon_3 &= (\text{the number of R-schedules when } (i = 1)) \\
&\quad + (\text{the number of R-schedules when } (i = 2)) \\
&= (\text{the number of R-schedules for } \mathit{left}(1)) \\
&\quad \times (\text{the number of R-schedules for } \mathit{right}(2)) \\
&\quad + (\text{the number of R-schedules for } \mathit{left}(2)) \\
&\quad \times (\text{the number of R-schedules for } \mathit{right}(1)) \\
&= (\varepsilon_1 \times \varepsilon_2) + (\varepsilon_2 \times \varepsilon_1) = 2\varepsilon_1\varepsilon_2 .
\end{aligned}$$

Continuing in this manner, we see that for each positive integer $n > 1$,

$$\varepsilon_n = \sum_{k=1}^{n-1} (\text{the number of R-schedules when } (i = k)) = \sum_{k=1}^{n-1} (\varepsilon_k \times \varepsilon_{n-k}) . \qquad (3.3)$$

The sequence of positive integers generated by (3.3) with $\varepsilon_1 = 1$ is known as the set of **Catalan numbers**, and each ε_i is known as the $(i-1)$th Catalan number. Catalan numbers arise in many problems in combinatorics; for example, the number of different binary trees with n vertices is given by the nth Catalan number, ε_n. It can be shown that the sequence generated by (3.3) is given by

$$\varepsilon_n = \frac{1}{n}\binom{2n-2}{n-1}, \text{ for } n = 1, 2, 3, \ldots , \qquad (3.4)$$

where $\binom{a}{b} \equiv \frac{a(a-1)\ldots(a-b+1)}{a!}$, and it can be shown that the expression on the right hand side of (3.4) is $\Omega(4^n / n)$ [Corm90].

For example, the chain-structured SDF graph in Figure 3-1 consists of four actors, so (3.4) indicates that this graph has $\frac{1}{4}\binom{6}{3} = 5$ R-schedules. The R-schedules for Figure 3-1 are $(3(3A)(4B))(4(3C)(2D))$, $(3(3A)(4(1B)(1C)))(8D)$, $(3(1(3A)(4B))(4C))(8D)$, $(9A)(4(3(1B)(1C))(2D))$, and $(9A)(4(3B)(1(3C)(2D)))$; and the corresponding buffer memory requirements are, respectively, 30, 37, 40, 43, and 45.

3.9 Dynamic Programming Algorithm

The problem of determining the R-schedule that minimizes the buffer memory requirement for a chain-structured SDF graph can be formulated as an optimal parenthesization problem. A familiar example of an optimal parenthesization problem is matrix chain multiplication [Corm90]. In matrix chain multiplication, we must compute the matrix product $M_1 M_2 \ldots M_n$, assuming that the dimensions of the matrices are compatible with one another for the specified multiplication. There are several ways in which the product can be computed. For example, with $n = 4$, one way of computing the product is $(M_1(M_2 M_3))M_4$, where the parenthesizations indicate the order in which the multiplies occur. Suppose that M_1, M_2, M_3, M_4 have dimensions $10 \times 1, 1 \times 10, 10 \times 3, 3 \times 2$, respectively. It is easily verified that computing the matrix chain product as $((M_1 M_2)M_3)M_4$ requires 460 scalar multiplications, whereas computing it as $(M_1(M_2 M_3))M_4$ requires only 120 multiplications (assuming that we use the standard algorithm for multiplying two matrices).

Thus, we would like to determine an optimal way of placing the parentheses so that the total number of scalar multiplications is minimized. This can be achieved using a dynamic programming approach. The key observation is that any optimal parenthesization splits the product $M_1 M_2 \ldots M_n$ between M_k and M_{k+1} for some k in the range $1 \le k \le (n-1)$, and thus the cost of this optimal parenthesization is the cost of computing the product $M_1 M_2 \ldots M_k$, plus the cost of computing $M_{k+1} M_{k+2} \ldots M_n$, plus the cost of multiplying these two products together. In an optimal parenthesization, the subchains $M_1 M_2 \ldots M_k$ and $M_{k+1} M_{k+2} \ldots M_n$ must themselves be parenthesized optimally. Hence this problem has the optimal substructure property and is thus amenable to a dynamic programming solution.

Determining the optimal R-schedule for a chain-structured SDF graph is similar to the matrix chain multiplication problem. Recall the example of Figure 3-1. Here $\mathbf{q}(A, B, C, D) = (9, 12, 12, 8)^T$; an optimal R-schedule is $(3(3A)(4B))(4(3C)(2D))$; and the associated buffer memory requirement is 30. Therefore, as in the matrix chain multiplication case, the optimal parenthesization (of the schedule body) contains a break in the chain at some $k \in \{1, 2, \ldots, (n-1)\}$. Because the parenthesization is optimal, the chains to the left of k and to the right of k must both be parenthesized optimally. Thus, we have the optimal substructure property.

Now given a chain-structured SDF graph G consisting of actors $A_1, A_2, \ldots, A_n$ and edges $\alpha_1, \alpha_2, \ldots, \alpha_{n-1}$, such that each α_k is directed from A_k to A_{k+1}, given integers i, j in the range $1 \le i \le j \le n$, denote by $b[i, j]$ the minimum buffer memory requirement over all R-schedules for $\mathit{subgraph}(\{A_i, A_{i+1}, \ldots, A_j\}, G)$. Then, the minimum buffer memory requirement over all R-schedules for G is $b[1, n]$. If $1 \le i < j \le n$, then,

$$b[i, j] = \mathit{min}(\{(b[i, k] + b[k+1, j] + c_{i,j}[k]) | (i \le k < j)\}), \tag{3.5}$$

where $b[i, i] = 0$ for all i,and $c_{i,j}[k]$ is the memory cost at the split if we split the chain at A_k. It is given by

$$c_{i,j}[k] = \frac{\mathbf{q}_G(A_k)\mathit{prd}(\alpha_k)}{\mathit{gcd}(\{\mathbf{q}_G(A_m) | (i \le m \le j)\})}. \tag{3.6}$$

The gcd term in the denominator arises because, the repetitions vector $\mathbf{q}'$ of $\mathit{subgraph}(\{A_i, A_{i+1}, \ldots, A_j\}, G)$ satisfies $\mathbf{q}'(A_p) = \frac{\mathbf{q}_G(A_p)}{\mathit{gcd}(\{\mathbf{q}_G(A_m) | (i \le m \le j)\})}$, for all $p \in \{i, i+1, \ldots, j\}$.

A dynamic programming algorithm derived from the above formulation is specified in Figure 3-9. In this algorithm, first the quantity $\mathit{gcd}(\{\mathbf{q}_G(A_m) | (i \le m \le j)\})$ is computed for each subchain $A_i, A_{i+1}, \ldots, A_j$. Then the two-actor subchains are examined, and the buffer memory requirements for these subchains are recorded. This information is then used to determine the minimum buffer memory requirement and the location of the split that achieves this minimum for each three-actor subchain. The minimum buffer memory requirement for each three-actor subchain A_i, A_{i+1}, A_{i+2} is stored in entry $[i, i+2]$ of the array Subcosts, and the index of the edge corresponding to the split is stored in entry $[i, i+2]$ of the SplitPositions array. This data is then examined to determine the minimum buffer memory requirement for each four-actor subchain, and so on, until the minimum buffer memory requirement for the n-actor subchain, which is the original graph G, is determined. At this point, procedure ConvertSplits is called to recursively construct an

```
procedure ScheduleChainGraph
input: a chain-structured SDF graph $G$ consisting of actors $A_1, A_2, \ldots, A_n$
and edges $\alpha_1, \alpha_2, \ldots, \alpha_{n-1}$ such that each $\alpha_i$ is directed from $A_i$ to $A_{i+1}$.
output: an R-schedule body for $G$ that minimizes the buffer memory requirement.

for $i = 1, 2, \ldots, n$                /* Compute the gcd's of all subchains */
    $\mathrm{GCD}[i, i] = \mathbf{q}_G(A_i)$
    for $j = (i+1), (i+2), \ldots, n$
        $\mathrm{GCD}[i, j] = gcd(\{\mathrm{GCD}[i, j-1], \mathbf{q}_G(A_j)\})$

for $i = 1, 2, \ldots, n$ $\mathrm{Subcosts}[i, i] = 0$;
for $\mathrm{chain\_size} = 2, 3, \ldots, n$
    for $\mathrm{right} = \mathrm{chain\_size}, \mathrm{chain\_size} + 1, \ldots, n$
        $\mathrm{left} = \mathrm{right} - \mathrm{chain\_size} + 1$;
        $\mathrm{min\_cost} = \infty$;
        for $i = 0, 1, \ldots, \mathrm{chain\_size} - 2$
            $\mathrm{split\_cost} = (\mathbf{q}_G(A_{\mathrm{left}+i}) / \mathrm{GCD}[\mathrm{left}, \mathrm{right}]) \times prd(\alpha_{\mathrm{left}+i})$;

$\mathrm{total\_cost} = \mathrm{split\_cost} + \mathrm{Subcosts}[\mathrm{left}, \mathrm{left} + i] + \mathrm{Subcosts}[\mathrm{left} + i + 1, \mathrm{right}]$;
            if $(\mathrm{total\_cost} < \mathrm{min\_cost})$
                $\mathrm{split} = i$; $\mathrm{min\_cost} = \mathrm{total\_cost}$
        $\mathrm{Subcosts}[\mathrm{left}, \mathrm{right}] = \mathrm{min\_cost}$; $\mathrm{SplitPositions}[\mathrm{left}, \mathrm{right}] = \mathrm{split}$;
output $\mathrm{ConvertSplits}(1, n)$; /* Convert the $\mathrm{SplitPositions}$ array into an R-schedule */


procedure ConvertSplits($L, R$)
implicit inputs: the SDF graph $G$ and the $\mathrm{GCD}$ and $\mathrm{SplitPositions}$ arrays
of procedure ScheduleChainGraph.
explicit inputs: positive integers $L$ and $R$ such that $1 \le L \le R \le n = |actors(G)|$.
output: An R-schedule body for $subgraph(\{A_L, A_{L+1}, \ldots, A_R\}, G)$ that minimizes
the buffer memory requirement.

if $(L = R)$ output $A_L$
else
    $s = \mathrm{SplitPositions}[L, R]$; $i_L = \mathrm{GCD}[L, L+s] / \mathrm{GCD}[L, R]$;
    $i_R = \mathrm{GCD}[L+s+1, R] / \mathrm{GCD}[L, R]$;
    output $(i_L \mathrm{ConvertSplits}(L, L+s))(i_R \mathrm{ConvertSplits}(L+s+1, R))$;
```

Figure 3-9 Pseudo-code for the dynamic programming algorithm to schedule a chain-structured graph.

optimal R-schedule from a top-down traversal of the optimal split positions stored in the SplitPositions array.

Assuming that the components of $\mathbf{q}_G$ are bounded, which makes the *gcd* computations elementary operations, it is easily verified that the time complexity of ScheduleChainGraph is dominated by the time required for the innermost **for** loop — the (**for** $i = 0, 1, \ldots,$ chain_size – 2) loop — and the running time of one iteration of this loop is bounded by a constant that is independent of n. Thus, the following theorem guarantees that under our assumptions, the running time of ScheduleChainGraph is $O(n^3)$ and $\Omega(n^3)$.

Theorem 3-4: The total number of iterations of the (**for** $i = 0, 1, \ldots,$ chain_size – 2) loop that are carried out in ScheduleChainGraph is $O(n^3)$ and $\Omega(n^3)$.

Proof: This is straightforward.

3.10 Example: Sample Rate Conversion

Digital audio tape (DAT) technology operates at a sampling rate of 48 kHz, while compact disk (CD) players operate at a sampling rate of 44.1 kHz. Interfacing the two, for example, to record a CD onto a digital tape, requires a sample rate conversion.

The naive way to do this is shown in Figure 3-10(a). It is more efficient to perform the rate change in stages. Rate conversion ratios are chosen by examining the prime factors

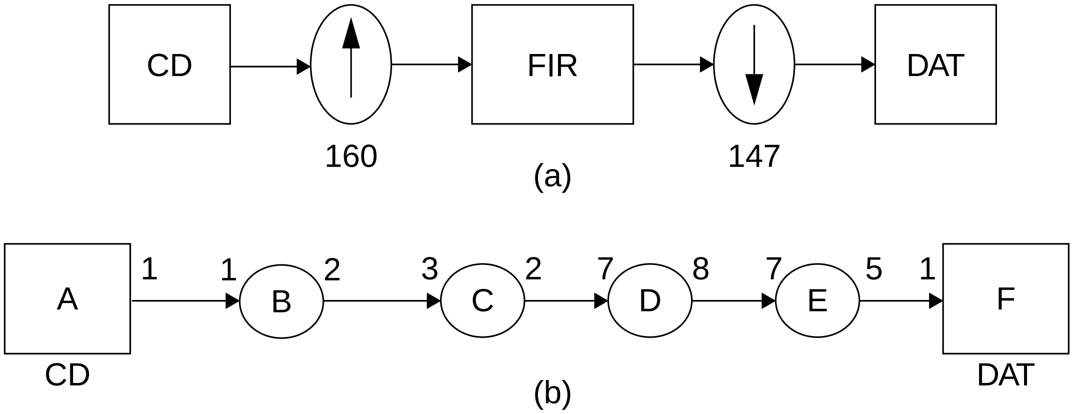


Figure 3-10 (a). CD to DAT sample rate change system.
(b). Multi-stage implementation of a CD to DAT sample rate system.

of the two sampling rates. The prime factors of 44,100 and 48,000 are $2^2 3^2 5^2 7^2$ and $2^7 3^1 5^3$, respectively. Thus, the ratio 44,100 : 48,000 is $3^1 7^2 : 2^5 5^1$, or 147 : 160. One way to perform this conversion in four stages is 2:1, 4 : 3, 4 : 7, and 5 : 7. Figure 3-10(b) shows the multistage implementation. Explicit upsamplers and downsamplers are omitted, and it is assumed that the FIR filters are general polyphase filters [Buck91].

Here $\mathbf{q}(A, B, C, D, E, F) = (147, 147, 98, 28, 32, 160)^T$; the optimal looped schedule given by our dynamic programming approach is $(7(7(3AB)(2C))(4D))(32E(5F))$; and the associated buffer memory requirement is 264. In contrast, the alternative schedule $(147A)(147B)(98C)(28D)(32E)(160F)$ has a buffer memory requirement of 1021 if a separate buffer is used for each edge and a buffer-memory requirement of 294 if one shared buffer is used. This is an important savings with regard to current technology: a buffer memory requirement of 264 will fit in the on-chip memory of most existing programmable digital signal processors, while a buffer memory requirement of 1021 is too high for all programmable digital signal processors, except for a small number of the most expensive ones. The savings of 30 (10 %) over using a single shared buffer can also be significant on chips that only have on the order of 1000 words of memory. It can be verified that the latency of the optimally nested schedule is given by $146T + E_A + E_B + 2E_C + 4E_D + E_E$, as opposed to $146T + E_A + 147E_B + 98E_C + 28E_D + 32E_E$ for the naive schedule. If we take $T = 500$ (for example, a 22.05Mhz chip has 22.05Mhz/44.1khz = 500 instruction cycles in one sample period of the CD actor), and $E_A = 10, E_B = E_C = E_D = E_E = 100$, then the two latencies are 73810 and 103510 instruction cycles; the nested schedule has 29% less latency.

One more advantage that a nested schedule can have over the naive schedule with shared buffering is in the amount of *input buffering* required. Some DSP chips have a feature where a dedicated I/O manager can write incoming samples to a buffer in on-chip memory, the size of which can be programmed by the user. If the single appearance schedule spans more than one sample period, then input buffering is a useful feature since it avoids the need for interrupts. Chips that have input buffering include the Analog Devices ADSP 2100. If we compute the amount of input buffering required by the naive schedule, we find that it is $((147 + 98 + 28 + 32)100 + 160 \times 10) / 500 \cong 65$, whereas for the optimally nested schedule, it is given by $(100 + 200 + 400 + 3200 + 1600) / 500 \cong 11$.

3.11 An Efficient Heuristic

Our dynamic programming solution for chain-structured graphs runs in $O(m^3)$ time, where m is the number of actors. As a quicker alternative solution, we developed a more time-efficient heuristic approach. The heuristic is simply to introduce the parenthesization on the edge where the minimum amount of data is transferred. This is done recursively for each of the two halves that result. The running time of the heuristic is given by the recurrence

$$T(n) = T(n-k) + T(k) + O(n), \tag{3.7}$$

where k is the actor at which the split occurs. This is because we must compute the *gcd* of the repetitions vector components of the k actors to the left of the split, and the *gcd* of the repetitions of the $n-k$ actors to the right. This takes $O(n)$ time assuming that the repetitions vector components are bounded. Computing the minimum of the data transfers takes a further $O(n)$ time since there are $O(n)$ edges to consider. The worst case solution to this recurrence is $O(n^2)$, but the average case running time is $O(n \bullet Log\ (n))$ if $k = \Omega(n)$.

We have evaluated the heuristic on $10,000$ randomly generated 50-actor chain-structured SDF graphs, and we have found that on average, it yields a buffer memory requirement that is within 60% of the optimal cost. For each random graph, we also compared the heuristic's solution to the worst-case schedule and to a randomly-generated R-schedule. On average, the worst-case schedule had over 9000 times higher cost than the heuristic's solution, and the random schedule had 225 times higher cost. Furthermore, the heuristic outperformed the random schedule on 97.8 percent of the trials. We also note that in over 99% of the randomly generated 50-actor chain-structured SDF graphs, the shared-buffer cost for the naive single appearance schedule was worse than the cost of the nested schedule given by the heuristic. Unfortunately, the heuristic does not perform well on the example of Figure 3-10 — it achieves a buffer memory requirement of 565, which is over double of what is required by an optimum R-schedule. In comparison, the worst R-schedule for Figure 3-10 has a buffer memory requirement of 755.

3.12 Extensions

In this section we present three useful extensions of the dynamic programming solution developed in Section 3.9. First, the algorithm can easily be adapted to optimally handle chain-structured graphs that have delays on one or more of the edges. This requires that we modify the computation of $c_{i,j}[k]$, the amount of memory required to split the subchain $A_i, A_{i+1}, \ldots, A_j$ between the actors A_k and A_{k+1}. This cost now gets computed as

$$c_{i,j}[k] = \frac{1}{r}\mathbf{q}_G(A_k)prd(\alpha_k) + del(\alpha_k),$$

where $r = gcd(\{\mathbf{q}_G(A_m) | (i \le m \le j)\})$, if

$$del(\alpha_k) < \frac{1}{r}\mathbf{q}_G(A_k)prd(\alpha_k);$$

otherwise (if $del(\alpha_k) \ge \frac{1}{r}\mathbf{q}_G(A_k)prd(\alpha_k)$), $c_{i,j}[k]$ gets computed as $c_{i,j}[k] = del(\alpha_k)$. Accordingly, if the optimum split extracted in a given invocation of *ConvertSplits* (Figure 3-9) corresponds to a split in which the latter condition applied in the computation of $c_{i,j}[k]$, then *ConvertSplits* returns $(i_R\text{ConvertSplits}(L + s + 1, R))(i_L\text{ConvertSplits}(L, L + s))$; otherwise, *ConvertSplits* returns $(i_L\text{ConvertSplits}(L, L + s))(i_R\text{ConvertSplits}(L + s + 1, R))$, as in the original version. This requires a method for keeping track of which condition applies to each of the optimum subchain splits, which can easily be incorporated, for example, by varying the sign of the associated entry in the *SplitPositions* array.

The technique applies to the more general class of well-ordered SDF graphs. A well-ordered graph is one where the partial order is a total order; chain-structured graphs are a special case of these. Again, this requires modifying the computation of $c_{i,j}[k]$. Here, this cost gets computed as

$$c_{i,\,j}[k] \;=\; \frac{\sum\limits_{\alpha \in E_s} \mathbf{q}_G(A_k)\,prd(\alpha_k)}{gcd(\{\mathbf{q}_G(A_m) \,|\, (i \le m \le j)\})}\,, \tag{3.8}$$

where

$$\begin{aligned} E_s \equiv \{\beta \,|\, (src(\beta) \in \{A_i, A_{i+1}, \ldots, A_k\}) \quad &; \\ \textbf{and } (snk(\beta) \in \{A_{k+1}, A_{k+2}, \ldots, A_j\})\} & \end{aligned} \tag{3.9}$$

that is, E_s is the set of edges directed from one side of the split to the other side.

The dynamic programming technique of Section 3.9 can also be applied to reducing the buffer memory requirement of a given single appearance schedule for an arbitrary acyclic SDF graph (not necessarily chain-structured or well-ordered).

Suppose we are given a valid single appearance schedule S for an acyclic SDF graph and again for simplicity, assume that the edges in the graph contain no delay. Let $\Psi = B_1, B_2, \ldots, B_m$ denote the sequence of lexical actor appearances in S (for example, for the schedule $(4A(2FD))C$, $\Psi = A, F, D, C$). Thus, since S is a single appearance schedule, Ψ must be a topological sort of the associated acyclic SDF graph. The technique of Section 3.9 can easily be modified to optimally “re-parenthesize” S into the optimal single appearance schedule (with regard to buffer memory requirement) associated with the topological sort Ψ. The technique is applied to the sequence Ψ, with $c_{i,\,j}[k]$ computed as in (3.8). It can be shown that the algorithm runs in time $O(|V|^3)$, where $|V|$ is the number of vertices in the graph.

Thus, given any topological sort Ψ^* for a consistent acyclic SDF graph, we can efficiently determine the single appearance schedule that minimizes the buffer memory requirement over all valid single appearance schedules for which the sequence of lexical actor appearances is Ψ^*.

Another extension applies when we relax the assumption that each edge is mapped to a separate block of memory, and allow buffers to be overlaid in the same block of memory. There are several ways in which buffers can be overlaid; the simplest is to have one memory segment of size

$$CS_{i,j} \equiv \frac{max(\{prd(\alpha_k) \times \mathbf{q}(A_k) | (i \le k < j)\})}{gcd(\{\mathbf{q}(A_i), \mathbf{q}(A_{i+1}), \ldots, \mathbf{q}(A_j)\})} \tag{3.10}$$

for the subchain $A_i, A_{i+1}, \ldots, A_j$ (as explained in Section 3.2). We follow this computation with

$$b'[i, j] = min(\{b[i, j], CS_{i,j}\}), \tag{3.11}$$

to determine amount of memory to use for buffering in the subchain $A_i, A_{i+1}, \ldots, A_j$. In general, this gives us a combination of overlaid and non-overlaid buffers for different subchains. Incorporating the techniques of this section with more general overlaying schemes is a topic for future work.

3.13 Extension to Arbitrary Topologies

The material reported in this section and the next is taken from [Bhat95]. DPPO can be extended to efficiently handle graphs that are not necessarily delayless, although a few additional considerations arise. We refer to this extension as **Generalized DPPO (GDPPO)**. First, if delays are present, then $lexorder(S)$, the lexical ordering of the input schedule, is not necessarily a topological sort. As a consequence, generally not all parenthesizations of the input schedule will be valid. For example, suppose that we are given the valid schedule $S = (6A)(5(2C)(3B))$ for Figure 3-11. Then $lexorder(S) = (A, C, B)$ clearly is not a topological sort, and it is easily verified that the schedule that corresponds to splitting the outermost parenthesization between C and B — $(2(3A)(5C))(15B)$ — is not a valid schedule since there is not sufficient delay on the edge (B, C) to fire 10 invocations of C before a single invocation of B.

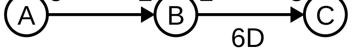


Figure 3-11 An SDF graph used to illustrate GDPPO applied to SDF graphs that have nonzero delay on one or more edges. Here $\mathbf{q}(A, B, C) = (6, 15, 10)$.

Thus, we see that when delays are present, the set E_s defined in equation (3.9) no longer generally gives all of the edges that cross the parenthesization split. We must also examine the set of *back edges*
$E_b = \{e | (snk(e) \in \{A_i, A_{i+1}, \ldots, A_k\}$ **and** $src(e) \in \{A_{k+1}, A_{k+2}, \ldots, A_j\})\}$
Each $e \in E_b$ must satisfy

$$del(e) \geq (TNSE(e))/(gcd(\{\mathbf{q}_G(A_x) | (i \leq x \leq j)\})); \qquad (3.12)$$

otherwise, the given parenthesization split will give a schedule that is not valid. To take into account any nonzero delays on members in E_s, and the memory cost of each of the back edges, the cost expression of equation (3.8) for the given split gets replaced with the *forward split cost* defined by

$$b[i, k] + b[k+1, j] + \sum_{e \in E_s} TNSE(e)/(gcd(\{\mathbf{q}_G(A_x) | (i \leq x \leq j)\})) + \sum_{e \in E_s \cup E_b} delay(e)$$

This expression gives the cost of splitting the subsequence $(A_i, A_{i+1}, \ldots, A_j)$ between A_k and A_{k+1} assuming that the subsequence $(A_i, A_{i+1}, \ldots, A_k)$ precedes $(A_{k+1}, A_{k+2}, \ldots, A_j)$ in the lexical order of the schedule that will be implemented. However, if (3.12) is satisfied *for all* "forward edges" $e \in E_s$, it may be advantageous to interchange the lexical order of $(A_i, A_{i+1}, \ldots, A_k)$ and $(A_{k+1}, A_{k+2}, \ldots, A_j)$. Such a reversal will be advantageous whenever the *reverse split cost* defined by

$$b[i, k] + b[k+1, j] + \sum_{e \in E_b} TNSE(e)/gcd(\{\mathbf{q}_G(A_x) | (i \leq x \leq j)\}) + \sum_{e \in E_s \cup E_b} del(e)$$

is less than the forward split cost — that is, whenever

$$\sum_{e \in E_b} TNSE(e) < \sum_{e \in E_s} TNSE(e). \qquad (3.13)$$

The possibility for reverse splits introduces a fundamental difference between GDPPO and DPPO: if one or more reverse splits are found to be advantageous, then GDPPO does not preserve the lexical ordering of the original schedule. If GDPPO changes the lexical ordering, then the result computed by GDPPO will necessarily have a buffer memory requirement that is less than that of an order-optimal schedule for $lexorder(S)$. In such cases, GDPPO may be applied multiple times in succession to possibly yield more benefit than a single application — that is, GDPPO can in general be applied iteratively, where the iterative application terminates when the schedule produced by GDPPO produces no improvement over the schedule computed in the previous iteration.

Although the iterative application of GDPPO is conceptually interesting, we have found that for all of the practical SDF graphs that we have applied it to, termination occurred after only 2 iterations, which means that no further improvement was ever generated by a second application of GDPPO. This suggests that when compile-time efficiency is a significant issue, it may be preferable to bypass iterative application of GDPPO, and immediately accept the schedule produced by the first application.

GDPPO can be implemented efficiently by updating forward and reverse costs incrementally. If we are examining the splits of the subsequence $(A_i, A_{i+1}, \ldots, A_j)$, and we have computed the forward and reverse split costs F_k and R_k associated with the split between A_k and A_{k+1}, $i \le k < (j-1)$, then the splits costs F_{k+1} and R_{k+1} associated with the split between A_{k+1} and A_{k+2} can easily be derived by examining the output and input edges of A_{k+1}. To ensure that we ignore reverse splits (forward splits) that fail to satisfy (3.12) for all $e \in E_s$ $(e \in E_b)$ a cost of

$$M \equiv 1 + \sum_{e \in E} (TNSE(e) + del(e))$$

is added to the reverse (forward) split cost for any input edge (output edge) e of A_{k+1} whose source (sink) is a member of $(A_{k+2}, A_{k+3}, \ldots, A_j)$, and that does not satisfy (3.12). Similarly, for each output (input) edge e of A_{k+1} whose sink (source) is contained in $(A_i, A_{i+1}, \ldots, A_k)$, and that does not satisfy (3.12), M is subtracted from R_{k+1} (F_{k+1}) since such an edge no longer prevents the split from being valid. Choosing M so large has the effect of “invalidating” any cost C_M that has M added to it (without a corresponding subtraction) since any minimal valid schedule has a buffer memory requirement less than M, and thus, any valid split will be chosen over a split that has cost C_M.

If forward and reverse costs are updated in this incremental fashion, then GDPPO attains a time complexity of $O(n_v^3)$ where n_v is the number of actors, if we can assume that the number of input and output edges of each actor is always bounded by some constant α. In the absence of such a bound, GDPPO has time complexity that is $O(n_e n_v^3)$, where n_e is the number of edges in the input graph.

3.14 Adaptation to Minimize Code Size for Arbitrary Schedules

We have applied the basic concept behind DPPO to derive an algorithm that computes an optimally compact looped schedule (minimum code size) for an arbitrary sequence of actor firings. For example, consider the SDF graph in Figure 3-12, and suppose that we are given the valid firing sequence[1] $\sigma = ABABCBC$ (this firing sequence minimizes the buffer memory requirement over all valid schedules). If the code size cost (number of program memory words required) for a code block for A is greater than the code size cost for C, then the optimally compact looped schedule for σ is $(2AB)CBC$, whereas $ABA(2BC)$ is optimal if C has a greater code size cost than A.

To understand how dynamic programming can be used to compute an optimally compact loop structure, suppose that $\sigma = A_1 A_2 \ldots A_n$ is the given firing sequence, and let $\sigma' = A_i A_{i+1} \ldots A_j$ be any subsequence of σ ($1 \le i < j \le n$). If the optimal loop structures for all $(j-i)$-length subsequences of σ are available, then we determine the optimal loop structure for σ' by first computing $C_k \equiv Z_{i,k} + Z_{k+1,j}$ for $k \in \{i, i+1, \ldots, j-1\}$, where $Z_{x,y}$ denotes the minimum code size cost for the subsequence $A_x, A_{x+1}, \ldots, A_y$. The value of k that minimizes C_k gives an optimum point at which to "split" the subsequence *if* $A_i A_{i+1} \ldots A_j$ *are not to be executed through a single loop.*

To compute the minimum cost attainable for σ' if $A_i A_{i+1} \ldots A_j$ are to be executed through a single loop, we first determine whether or not $\sigma' = A_i A_i \ldots A_i$. If this holds, then σ' can be executed through a single loop $((j-i+1)A_i)$, and the code size cost is taken to

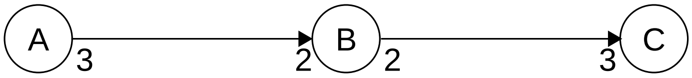


Figure 3-12 An SDF graph used to illustrate the problem of finding an optimally compact loop structure for an arbitrary firing sequence.

[1] By a *firing sequence*, we simply mean a schedule that contains no loops.

be the code size cost of A_i plus the code size overhead C_L of a loop. If $\sigma' \neq A_i A_i \ldots A_i$, we determine whether or not $\sigma' = A_i A_{i+1} A_i A_{i+1} \ldots A_i A_{i+1}$, and if so, then σ' can be implemented as $(((j - i + 1)/2) A_i A_{i+1})$, and the code size cost is taken to be the sum of C_L and the costs of A_i and A_{i+1}. Next, if $\sigma' \neq A_i A_i \ldots A_i$ and $\sigma' \neq A_i A_{i+1} A_i A_{i+1} \ldots A_i A_{i+1}$, then we determine whether or not $\sigma' = A_i A_{i+1} A_{i+2} \ldots A_i A_{i+1} A_{i+2}$. If this holds then σ' can be implemented as $(((j - i + 1)/3) S_L(A_i A_{i+1} A_{i+2}))$, where $S_L(A_i A_{i+1} A_{i+2})$ is an optimal loop structure for $A_i A_{i+1} A_{i+2}$. It is easily seen that an optimal loop structure L for executing $A_i A_{i+1} \ldots A_j$ through a single loop can be determined (if one exists) by iterating this procedure $\lfloor (j - i + 1)/2 \rfloor$ times, where $\lfloor * \rfloor$ denotes the *floor* operator. If one or more loop structures exist for executing $A_i A_{i+1} \ldots A_j$ through a single loop, then the code size cost of the optimal loop structure L is compared to the minimum value of C_k, $i \leq k < j$, to determine an optimal loop structure for $A_i A_{i+1} \ldots A_j$; otherwise the optimal loop structure for $A_i A_{i+1} \ldots A_j$ is taken to be that corresponding to the minimum value of C_k.

The time complexity of this technique for finding an optimal loop structure for the subsequence $A_i A_{i+1} \ldots A_j$, given optimal looping structures for all $(j - i)$-length subsequences, is $O((j - i + 1)^2)$, and time complexity of the overall algorithm is $O(n^4)$, where n is the number of firings in the input firing sequence. Thus, the problem of determining an optimal looping structure is of polynomial complexity when the size of a problem instance is taken to be the number of firings in the given firing sequence. However, it should be noted that there is no polynomial function $P(m)$ such that the number of firings in a valid schedule (defined as the sum of the entries in the repetitions vector) is guaranteed to be less than $P(m)$ for an arbitrary m-actor SDF graph. Thus, unlike DPPO and GDPPO, the algorithm developed in this section is not of polynomial complexity in the size of the given SDF graph.

3.15 Complexity of the Buffer Minimization Problem

Here we show that the problem of constructing buffer-optimal single appearance schedules for acyclic graphs with delays is NP-complete in general. In fact, we prove the result for HSDF graphs. Since any schedule for an HSDF graph is a single appearance schedule, it follows that the problem for general acyclic SDF graphs, with delays allowed on edges, is also NP-complete. It also follows that computing a minimum buffer schedule for an arbitrary acyclic SDF graph with delays allowed, without the single appearance restriction is also NP-complete. Finally, cyclic graphs are an even more general case, so both the single appearance and non-single appearance, buffer minimal scheduling problems for HSDF and SDF graphs are NP-complete (this was already shown in Chapter 2, but that proof is redundant in light of this stronger proof). The only remaining interesting class of graphs is the set of delayless acyclic (but not well-ordered) SDF graphs (not homogenous). For this class, the complexity of the minimum buffer scheduling problems remains open.

To gain some intuition about why this problem might be difficult, note that if some edges have delays, then in an HSDF graph, they do not impose any precedence constraints since either the source actor of that edge or the sink actor of that edge can be fired before the other. However, if the sink actor fires before the source actor (for an edge that has 1 delay), then the buffer size on that edge can be 1; otherwise, it has to be 2. This suggests the following simple technique for getting a minimum buffer schedule: reverse all the edges that have delays and remove the delays. If we can schedule this graph, then we will have a minimum buffer schedule since on all reversed edge, the sink actor will fire before the source actor. We will be able to schedule this new graph if and only if it does not have any cycles. If it has cycles, then the problem is to determine a minimal set of edges that we reversed, to remove, so that the resulting graph becomes acyclic and a schedule may be found. The buffering requirement in that case will be increased by the size of the set of edges removed. However, as the result below shows, finding a minimal set of edges from the set of edges that have delays is not possible to do in polynomial time unless **P** = **NP**.

Definition 3-5: The AHSDF MIN BUFFER problem is the following

Instance: An acyclic, directed graph $G = (V, A)$ where every edge has 0 or 1 delays, and an integer K.

Question: Is there a schedule for G that has a total buffering requirement of $|A| + K$ or less?

Remark: Note that since we have a buffer on every arc, the buffering requirement has to be at least $|A|$.

Definition 3-6: The vertex cover (VC) problem is the following:

Instance: An undirected graph $G' = (V', A')$, and integer k.
Question: Is there a subset $V'' \subset V'$, with $|V''| \le k$, such that V'' covers every edge; that is, for every edge $(u, v) \in A$, at least one of u, v is in V''?
Remark: For an undirected graph, if (u, v) is an edge, so is (v, u).

Theorem 3-5: VC is NP-complete [Karp72].

Theorem 3-6: AHSDF MIN BUFFER is NP-complete.

Proof: Membership in NP is easy to see since we just have to simulate the schedule to see if the buffering requirement is met; this can be done in linear time since the schedule has length $|V|$. Completeness follows from a reduction of vertex cover. From an arbitrary instance $G' = (V', A')$, k, of the VC problem, we construct the instance $G = (V, A)$ of AHSDF MIN BUFFER as follows. Let $V = \{v_0, v_1 : v \in V'\}$. Let $A_1 = \{(v_1, v_0) : v \in V'\}$, $A_0 = \{(v_1, w_0) : (v, w) \in A'\}$, $A = A_0 \cup A_1$, and $K = k$. Each edge in A_1 has one delay, and each edge in A_0 has 0 delays. We refer to a vertex of the form v_0 as a "0" vertex and to a vertex of the form v_1 as a "1" vertex. Clearly, this is an instance of AHSDF MIN BUFFER; the graph is acyclic because all edges are directed from a "1" vertex to a "0" vertex. We claim that this instance of AHSDF MIN BUFFER has a solution iff the VC instance has a solution.

Suppose that there is a solution U to the VC instance. Let W be the set of edges defined as $W = \{(v_1, v_0) : v \in U\}$. Note that $W \subset A_1$. Delete these edges from G, reverse the rest of the edges in A_1, and remove the delays from them to get the graph G''. Clearly G'' is delayless. We claim that it is also acyclic. Suppose that it were not acyclic. Then there would be a directed cycle of the form $u^1 \to u^2 \to \ldots \to u^m \to u^1$ in G''. Without loss in generality, assume that $u^1 = v_0$ for some v_0. A "0" vertex of this type can only have an outgoing edge directed to the vertex v_1 in G''; hence, $u^2 = v_1$. A "1" vertex can only have an outgoing edge to some "0" vertex; hence, $u^3 = w_0$ for some w_0. Continuing this argument, it can be seen that the length of the cycle has to be even, and that there are $m/2$ “0” vertices and for each such vertex v_0, v_1 is also in the cycle. None of these vertices v can be in U since all edges of the form (u_1, u_0) were deleted for u in U, and only the remaining edges (from A_1) were reversed to yield edges of the form (v_0, v_1). But since (v_1, w_0) is an edge in the above cycle, it follows that (v, w) is an edge in G', but it is not covered by U. Hence, U cannot be a solution to the VC instance, giving us a contradiction. Now, since G'' is acyclic and delayless, it has a valid schedule. This schedule is also a valid schedule for G since it respects all the precedence constraints of the delayless arcs in G. On all arcs that were reversed, the sink actor in the original graph G is a source actor in G''; hence, on all these arcs, the buffer size is 1 in G. For the deleted arcs, we could have the source actor firing before the sink actor, and on these arcs the buffer size would be 2. Since there are at most K deleted arcs, the total buffering requirement is at most $|A| + K$.

Now suppose that the AHSDF MIN BUFFER instance has a schedule with buffering requirement of at most $|A| + K$. This means that there are at most K arcs that have delays where the source actor of the arc is fired before the sink actor in the schedule; denote this set of arcs by W. For all other arcs that have delays, the sink actor fires before the source actor. Since any arc with a delay in G is of the form (v_1, v_0), let the set U be defined as $U = \{v : (v_1, v_0) \in W\}$. Clearly, $|U| = |W| \le K$. We claim that U is a vertex cover for G'. Indeed, suppose it were not. Then there would be an edge (v, w) in G' where neither v, w is in U. This means that neither of $(v_1, v_0), (w_1, w_0)$ is in W. This means that in the schedule for G, v_0 fires before v_1, and w_0 fires before w_1. But since (v, w) is an edge in G', (v_1, w_0) and (w_1, v_0) are delayless edges in G, meaning that v_1 must fire before w_0, and w_1 must fire before v_0 in any valid schedule. Putting this together, we see that we have a cyclic dependency $v_0 \to v_1 \to w_0 \to w_1 \to v_0$ that cannot possibly be respected by the schedule, thereby contradicting our assumption that the set U is not a vertex cover. ■

Note that if buffer sharing is allowed, then the problem of determining a schedule for an HSDF graph that minimizes the total number of “live” tokens at any point in the schedule is exactly the REGISTER SUFFICIENCY problem proved to be NP-complete in by Sethi [Seth75].

In the next few sections, we develop heuristics for generating single appearance schedules that attempt to minimize the amount of buffering memory required for acyclic SDF graphs.

3.16 Recursive Partitioning by Minimum Cuts (RPMC)

The number of topological sorts in an acyclic graph can be an exponential function of the size of the graph; for example, a complete bipartite graph with $2n$ vertices has $(n!)^2$ possible topological sorts. Each topological sort gives a valid flat single appearance schedule. An optimal reparenthesization of this schedule is then computed by applying the dynamic programming algorithm. The problem is therefore to determine the topological sort that will give the lowest buffer memory requirement when nested optimally. For example, the graph in Figure 3-13 shows a bipartite graph with 4 vertices. The repetitions vector for the graph is given by $(12, 36, 9, 16)^T$, and there are 4 possible topological sorts for the graph. The flat schedule corresponding to the topological sort $ABCD$ is given by $(12A)(36B)(9C)(16D)$. This can be parenthesized as $(3(4A)(3(4B)C))(16D)$, and this schedule has a buffer memory requirement of 208. The flat schedule corresponding to the topological sort $ABDC$, when parenthesized optimally, gives the schedule $(4(3A)(9B)(4D))(9C)$, with a buffer memory requirement of 120.

A heuristic solution for this problem can be formulated as follows: find the *cut* (a partition of the set of actors) of the graph across which the minimum amount of data is transferred and schedule the resulting halves recursively. The cut that is produced must have the property that all edges that cross the cut have the same direction. This is to ensure that all the vertices on the left side of the partition can be scheduled before scheduling any

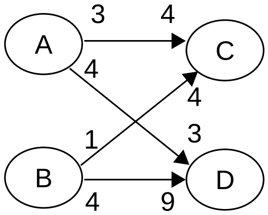


Figure 3-13 A bipartite SDF graph to illustrate the different buffer memory requirements possible with different topological sorts.

on the right side. In addition, we would like to impose the constraint that the partition that results be fairly evenly sized. This is to increase the possibility of having gcds that are greater than unity for the repetitions of the vertices in the subsets produced by the partition, thus reducing the buffer memory requirement. To see that having gcds greater than one for the subsets produced is beneficial to memory reduction, consider Figure 3-13. If the partition that had actor B on one side of the cut and actors A, C, D on the other side of the cut were formed, then we would get the loop bodies $(36B)$ and $((12A)(9C)(16D))$ and would not immediately see a reduction in buffering requirements since the repetitions of A, C, D are co-prime. However, a partition with A, B, C on the same side of the cut immediately gives us a reduction since the schedule body $((12A)(36B)(9C))$ can be factored as $(3(4A)(12B)(3C))$, and this reduces the memory for the subgraph consisting of actors A, B, C. In general, by constraining the sizes of the partition, the probability of being able to factor schedule bodies so that a reduction in memory is obtained in each stage of the recursion is increased. Needless to say, this is a greedy approach which is likely to fail sometimes but has proved to be a good rule of thumb for most instances.

Suppose that G is an SDF graph, and let $V = actors(G)$ and $E = edges(G)$. A **cut** of G is a partition of the vertex set V into two disjoint sets V_L and V_R. Define $G_L = subgraph(V_L)$ and $G_R = subgraph(V_R)$ to be the subgraphs produced by the cut. The cut is **legal** if for all edges e *crossing* the cut (that is all edges that are not contained in $subgraph(V_L)$ nor $subgraph(V_R)$), we have $src(e) \in V_L$ and $snk(e) \in V_R$. Given a *bounding constant* $K \le |V|$, the cut results in bounded sets if it satisfies

$$|V_R| \le K\,,\ |V_L| \le K\,. \tag{3.14}$$

The weight of an edge e is defined as

$$w(e) = \mathbf{q}_G(src(e)) \times prd(e)\,. \tag{3.15}$$

The weight of the cut is the total weight of all the edges crossing the cut. The problem then is to find the minimum weight legal cut into bounded sets for the graph with the weights defined as in (3.15). Since the related problem of finding a minimum cut (not necessarily legal) into bounded sets is NP-complete [Gare79], and the problem of finding an acyclic partition of a graph is NP-complete [Gare79], we believe this problem to be NP-complete as well even though we have not discovered a proof. Kernighan and Lin [Kern70] devised a heuristic procedure for computing cuts into bounded sets but they considered only undirected graphs. Methods based on network flows [Corm90] do not work because the minimum cut given by the max-flow-min-cut theorem may not be legal and may not be bounded. The graph in Figure 3-14, where a weight on an edge denotes the capacity of that edge, illustrates this. The maximum flow into vertex t is seen to be 3 (1 unit of flow along the path $sBCt$, 1 unit along $sADt$ and 1 unit along $sBDt$) and this corresponds to the cut where $V_L = \{s, B, C\}$ and $V_R = \{A, D, t\}$. The value of the cut is given by $1 + 1 + 1 = 3$ (note that the definition of the value of a cut in network flow theory is defined as sum of the capacities of the edges crossing the cut in the s to t direction only) but the cut is not legal because of the reverse edge from A to C. Indeed, the minimum weight legal cut for this graph has a value of 11, corresponding to the cut where $V_L = \{s\}$.

Therefore, a heuristic solution for finding legal minimum cuts into bounded sets is given. The heuristic is to examine the set of cuts produced by taking a vertex and all of its *descendants* as the vertex set V_R and the set of cuts produced by taking a vertex and all of its *ancestors* as the set V_L. For each such cut, an optimization step is applied that attempts to improve the cost of the cut.

Consider a cut produced by setting $V_L = ancs(v), V_R = V \setminus V_L$ for some vertex v. Consider the set $T_R(v)$ of independent, *boundary* vertices of v in V_R. A **boundary vertex** in V_R is a vertex that is not the predecessor of any other vertex in V_R. Following Kernighan and Lin [Kern70], for each of these vertices, the cost difference that results if the vertex is moved into V_L can be computed efficiently. This cost difference for a vertex

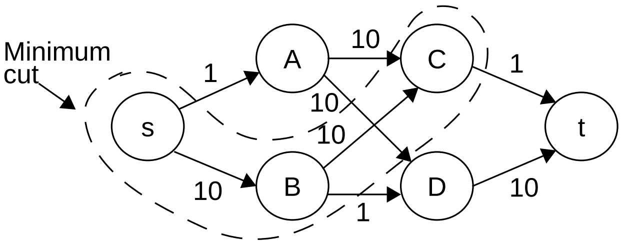


Figure 3-14 The min-cut given by the max-flow-min-cut theorem is not equal to the min-legal cut for this graph.

a in $T_R(v)$ is defined to be the difference between the total weight of all the edges out of a and the total weight of all edges into a. We then move those vertices across that reduce the cost. We apply this optimization step for all cuts of the form $ancs(v)$ and $desc(v)$ for each vertex v in the graph and take the best one as the minimum cut. The algorithm is shown in Figure 3-15. Since a greedy strategy is being used to move vertices across, and only the boundary vertices are considered, examples can be constructed where the heuristic will not give optimal cuts. Since there are $|V|$ vertices in the graph, $2|V|$ cuts are examined. Moreover, the cut produced will have bounded sets since cuts that produce unbounded sets are discarded. For example, one of the cuts examined by the heuristic for the graph in Figure 3-14, with bounding constant $K = |V| - 1$, is $ancs(A) = \{s, A\}$. This cut has a value of 30. The set of independent, boundary vertices of A in V_R is $\{B\}$, and the cost difference for B is given by $11 - 10 = 1$. Hence, B will not be moved over. The cut produced by considering $ancs(C) = \{s, A, B, C\}$ has a value of 12. The cost difference for the independent vertex D is given by $10 - 11 = -1$; hence, D is moved into V_L to yield a cut of value 11, and thus, in this example, the heuristic finds the minimum weight legal cut.

Delays on edges are handled as follows. If the number of delays on some edge e satisfies

$$del(e) \geq TNSE(e), \tag{3.16}$$

then the size of the buffer on this edge need not be any greater than $del(e)$. However, if e crosses the cut, then the size of the buffer will become $del(e) + TNSE(e)$. Hence, an edge that satisfies equation (3.16), is *tagged*; a tagged edge does not affect the legality of the cut (in other words, the heuristic ignores tagged edges when it constructs the legal cut) but affects the cost of the cut: if a tagged edge crosses the cut in the reverse direction, the cost of the edge is given by $del(e)$, and if the tagged edge crosses the cut in the forward direction, the cost is given by $del(e) + TNSE(e)$. This will discourage the heuristic from choosing partitions where tagged edges cross the cut in the forward direction.

The running time of the heuristic for computing the legal minimum cut into bounded sets can be determined as follows. Computing the descendents or ancestors of a vertex can be done by using breadth-first-search; this takes time $\Theta(|V| + |E|)$. The breadth-

```
procedure MinimumLegalCutIntoBoundedSets
input: weighted directed graph $G = (V, E)$, and a bound b. output: $V_R, V_L$.
for each $u \in V$

          /* Start with descendents */

          $S = desc(u)$, $\bar{S} = V \backslash S$
          $cutVal = cut(\bar{S}, S)$
          $T_L(u) \leftarrow independent(u) \bigcap boundary(\bar{S})$
          for each $a \in T_L(u)$
               $E(a) = \sum_{x \in S} w(a, x)$, $I(a) = \sum_{x \in \bar{S}} w(x, a)$
               $D(a) = I(a) - E(a)$ /* Cost difference if this vertex is moved over */
          end for
          $[D, Idx] \leftarrow sort(D)$
          $k \leftarrow 1$
          while $(|S| < b \ \& \ D(k) < 0 \ \& \ k < |T_L(u)|)$
               $S \leftarrow S \bigcup \{Idx(k)\}$
               $\bar{S} \leftarrow \bar{S} \backslash \{Idx(k)\}$
               $cutVal \leftarrow cutVal + D(k)$
               $k \leftarrow k + 1$
          end while
          $minCutVal \leftarrow min(minCutVal, cutVal)$
          if $(mincutVal \equiv cutVal)$, $V_L \leftarrow \bar{S}$, $V_R \leftarrow S$, end if

          /* Start with ancestors */

          $P = ancs(u)$, $\bar{P} = V \backslash P$

          /* Carry out the same type of steps as above to determine
          the partition for the set starting from the ancestors */
end for
/* $minCutVal, V_L, V_R$ correspond to the minimum legal cut. */
```

Figure 3-15 Procedure for computing legal minimum cuts.

first-search will also give us the independent vertices in the complement set. Finding and computing the cost difference for each of the boundary vertices in the set of independent vertices takes at most $O(|E|)$ steps. Sorting the cost differences takes $O(|V| \bullet \log(|V|))$ steps, and moving the vertices that reduce the cost takes $O(|V|)$ time. Since a cut is determined for every vertex twice, the total running time is $O(|V||E| + |V|^2 \bullet \log(|V|))$.

The heuristic for generating a schedule for the acyclic graph now proceeds by partitioning the graph by computing the legal minimum cut and forming the schedule body $(r_L S_L)(r_R S_R)$ where $r_L = gcd(\{\mathbf{q}(v)|v \in V_L\})$, $r_R = gcd(\{\mathbf{q}(v)|v \in V_R\})$ and S_L, S_R are schedule bodies for G_L and G_R respectively. The schedule bodies S_L, S_R are obtained recursively by partitioning G_L and G_R. Once the entire schedule body has been constructed, the dynamic programming algorithm is run to re-parenthesize the schedule to possibly give a better nesting. Letting $n = |V|$, the running time for this heuristic can be determined by solving the recurrence

$$T(n) = T(n-k) + T(k) + O(n|E| + n^2 \bullet \log(n)),$$

where $k = \left|V_L\right|$ and $n - k = \left|V_R\right|$. If the bound K in (3.14) is chosen to be a constant factor of the graph size, for example, 3/4, then it can be shown easily that $T(n) = O(|V||E| + |V|^2 \bullet \log(|V|))$. If the size of the sets is not bounded to be a constant factor of the graph size, then the worst case running time is $O(|V|^2|E| + |V|^3 \bullet \log(|V|))$. The reparenthesizing step that is run at the end uses the dynamic programming algorithm and requires $O(|V|^3)$ running time. Thus the overall running time is given by $O(|V|^3)$ assuming that K is a constant factor of the graph size.

3.17 More Analysis of legalCutIntoBddSets

In this section, we describe some of the reasons why the problem of computing legal minimum cuts into bounded sets optimally appears to be difficult. The type of graph that presents problems is one where there are several independent vertices that can be moved across. For example, consider the graph shown in Figure 3-16. It is clear that if the vertex ij is chosen as the starting point for the cut, then there $k-1$ rows of independent vertices, any of which can be chosen to be moved across. Another way to think about this is to consider a matrix filled with non-negative integers. The problem is to choose an element from each row such that the sum of the elements is minimized and that the number of elements to the right of the partition line that results is equal to the number of elements to the left of the partition line. This corresponds to the special case of the bound on the sets being exactly $|V|/2$. This problem seems to be NP-complete but we have not discovered a reduction that proves it.

3.18 Non-uniform Filterbank Example

Figure 3-17 shows the SDF graph abstraction of a non-uniform, near-perfect reconstruction filterbank [Naye93]. The lowpass filters (vertices c, f etc.) retain 2/3 of the spectrum while the highpass filters (vertices d, g etc.) retain 1/3 (instead of the customary 1/2,1/2 for the octave QMF). Rate changes in the graph are annotated wherever the number produced or consumed is different from unity. The repetitions vector of this graph is given by

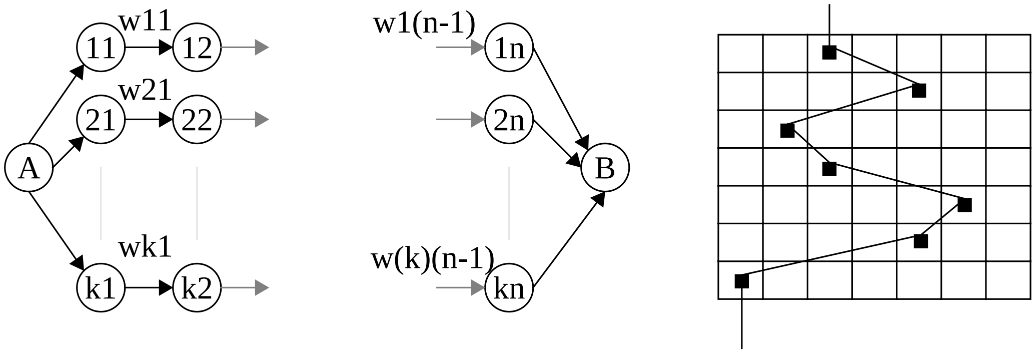


Figure 3-16 a) An example of the type of graph for which computing the legal minimum cut into bounded sets is challenging. b) The matrix interpretation of the problem.

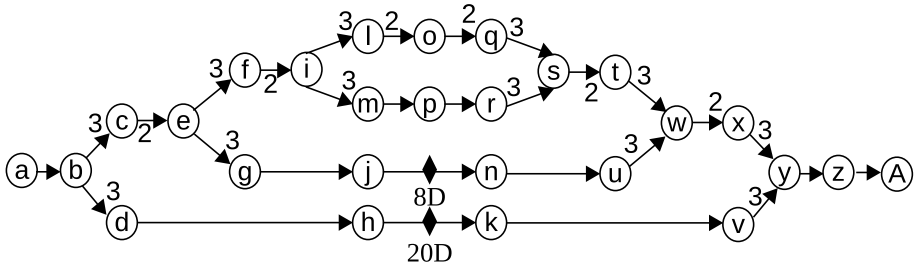


Figure 3-17 SDF graph for a non-uniform filterbank. The highpass channel retains 1/3 of the spectrum and the lowpass channel retains 2/3 of the spectrum.

$$\mathbf{q}(a, \ldots, A) = [27, 27, 9, 9, 18, 6, 6, 9, 12, 6, 9, 4, 4, 6, 8, 4, 4, 4, 12, 6, 6, 9, 18, 9, 27, 27, 27]$$

The single appearance schedule obtained by the APGAN scheduling heuristic (discussed in Section 3.19) on this system is

$$(3(3k(3ab)cd(2e)h)(2nugj)(2f(2i)))(4l(2o)q)(4mpr)(3(2(2s)t)(3(2w)xv(3yzA))),$$

and the resulting buffer memory requirement is 153. After post-processing the schedule with GDPPO, we obtain the schedule

$$(3(3(3ab)cd(2e)kh)(2gnufj))(4(3i)l(2o)qmpr(3s))(3(2t)(3(2w)xv(3yzA))),$$

which has a buffer memory requirement of 137 — a 10.5% improvement. Notice that in this example, GDPPO has changed the lexical ordering, and thus, one or more reverse splits were found to be beneficial.

The schedule returned by the RPMC scheduling heuristic has a buffer memory requirement of 131, which is lower than that obtained by the combination of APGAN and GDPPO. However, GDPPO is able to improve the schedule obtained by RPMC even further. The result computed by GDPPO when applied to the schedule derived by RPMC is

$$(3(3(3ab)dc)(2(3e)fgnj)(3kh))(4(3i)mpl(2o)qr(3s))(3(2tu(3w))(3xv(3yzA))),$$

which has a buffer memory requirement of 128.

For the same graph, if the delays are removed from the edges and instead implemented as vertices, then the RPMC heuristic obtains a schedule with a buffering cost of 100; the worst case flat schedule (for any topological sort) would have a buffering cost of 438. The best of the various PGAN heuristics (discussed in Section 3.19) found a schedule of cost 117. This shows that RPMC can be effective on graphs with irregular rate-changes in practice. More extensive experimental results on several other practical examples, and numerous random examples, are reported in Section 3.20 and 3.21.

3.19 Pairwise Grouping of Adjacent Nodes for Acyclic Graphs

In the original *Pairwise Grouping of Adjacent Nodes (PGAN)* technique for joint code and data minimization, a cluster hierarchy is constructed by clustering exactly two adjacent vertices at each step [Bhat93]. At each clustering step, a pair of adjacent actors is chosen that maximizes $\rho(\{A, B\})$, the repetition count of the adjacent pair, over all clusterable adjacent pairs $\{A, B\}$. Recall that $\rho(Z)$ can be viewed as the number of times a minimal periodic schedule for the subset of actors Z is invoked in the given SDF graph, and thus, we see that the PGAN technique repeatedly clusters adjacent pairs whose associated subgraphs are invoked most frequently in a valid schedule.

The PGAN technique verifies whether or not an adjacent pair is clusterable by checking whether or not its consolidation introduces a cycle in the Acyclic Precedence Graph (APG). It is shown that this check can be performed quickly by applying a *reachability matrix*, which indicates for any two APG vertices x and y whether or not there is a path from x to y.

Unfortunately, the cost to compute and store the reachability matrix can be prohibitively high for multirate applications that involve large changes in sample rate. Since the number of vertices in the APG of an SDF graph (V, E) is $J \times I$, where J is the desired blocking factor, I is defined as

$$I \equiv \sum_{X \in V} \mathbf{q}(X),$$

and the number of entries in the reachability matrix is quadratic in the number of APG vertices, it is easily seen that the time and space required to maintain the APG can grow exponentially with the number of actors in the given SDF graph. Although this is not a problem for the large class of practical SDF graphs for which I is not much larger than the number of elements in V, practical examples can easily be constructed where the technique consumes enormous amounts of resources relative to the size of the input SDF graph. For example, for the 6-vertex SDF representation of a multi-stage sample-rate conversion system between a compact disc player and a digital audio tape player discussed in Section 3.10, $I > 600$, which means that over $360,000$ units of storage are required to implement the reachability matrix for this 6-actor SDF graph.

Since a large proportion of DSP applications that are amenable to the SDF model can be represented as acyclic SDF graphs, a simple adaptation of PGAN has been proposed for acyclic graphs that maintains the cluster hierarchy and reachability matrix directly on the input SDF graph rather than on the APG, and thus allows us to efficiently exploit the advantages of the bottom-up clustering approach of the original PGAN technique. We refer

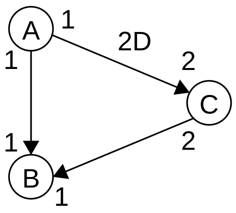


Figure 3-18 An example of how a clusterization operation that introduces a cycle can lead to a BMLB schedule. Here $(A, B, C) = (2, 2, 1$.

to this adaptation of PGAN as **Acyclic PGAN (APGAN)**. APGAN is exactly the original PGAN technique specified in [Bhat93] with the exception that the input SDF graph is assumed to be acyclic, and the cluster hierarchy and reachability matrix are maintained for the input SDF graph rather than for the APG.

In an acyclic SDF graph G, it is easily verified that if a subset Z of actors is not clusterable then $clust(Z, G, \Omega)$ contains a cycle. Whether Z introduces a cycle is easily checked given a reachability matrix for G by examining each successor of a member of Z: $clust(Z, G, \Omega)$ contains a cycle if and only if there is an $X \notin Z$ such that X is a successor of some member of Z, *and* there is a path from X to some member of Z.

Since the existence of a cycle in $clust(Z, G, \Omega)$ is only a necessary — but not sufficient — condition for Z not to be clusterable, the clusterability test that is applied in APGAN is not *exact*; it must be viewed as a conservative test. It is even inexact if we restrict ourselves to single appearance schedules. That is, it is possible for $clust(Z, G, \Omega)$ to contain a cycle, and still have a valid single appearance schedule. A simple example is shown in Figure 3-18. Here, the BMLB schedule $C(2AB)$ results if we first cluster $\{A, B\}$. However in APGAN, clustering $\{A, B\}$ is not permitted since the resulting graph contains a cycle. Instead, APGAN generates the schedule $(2A)C(2B)$ or the schedule $C(2A)(2B)$, neither of which is a BMLB schedule. Thus, in this example, we see that the inexact clusterization test prevents us from obtaining an optimal schedule.

In exchange for some degree of suboptimality in certain examples, the clusterization test attains a large computational savings over the exact test based on the reachability matrix of the APG, and this is the main reason for adopting it.

Figure 3-19 illustrates the operation of APGAN. Figure 3-19(a) shows the input SDF graph. Here $\mathbf{q}(A, B, C, D, E) = (6, 2, 4, 5, 1)$, and for $i = 1, 2, 3, 4$, Ω_i represents the ith hierarchical actor instantiated by APGAN. Each edge corresponds to a different adjacent pair; the repetition counts of the adjacent pairs are given by

$$\rho(\{A, B\}) = \rho(\{A, C\}) = \rho(\{B, C\}) = 2,$$

and

$$\rho(\{C, D\}) = \rho(\{E, D\}) = \rho(\{B, E\}) = 1.$$

Thus, APGAN will select the one of the three adjacent pairs $\{A, B\}$, $\{A, C\}$, or $\{B, C\}$ for its first clusterization step. Examination of the reachability matrix yields that $\{A, C\}$ introduces a cycle due to the path $((A, B), (B, C))$, while the other two adjacent pairs do not introduce cycles. Thus, APGAN chooses arbitrarily between $\{A, B\}$ and $\{B, C\}$ as the first adjacent pair to cluster.

Figure 3-19(b) shows the graph that results from clustering $\{A, B\}$ into the hierarchical actor Ω_1. In this graph, $\mathbf{q}(\Omega_1, C, D, E) = (2, 4, 5, 1)$, and it is easily verified that $\{\Omega_1, C\}$ uniquely maximizes ρ over all adjacent pairs. Since $\{\Omega_1, C\}$ does not introduce a cycle, APGAN selects this adjacent pair for its second clusterization step. Figure 3-19(c) shows the resulting graph.

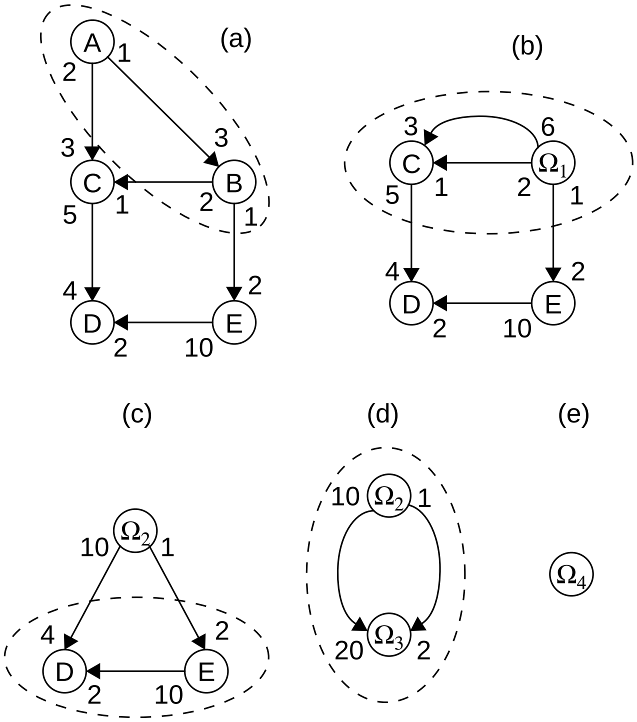
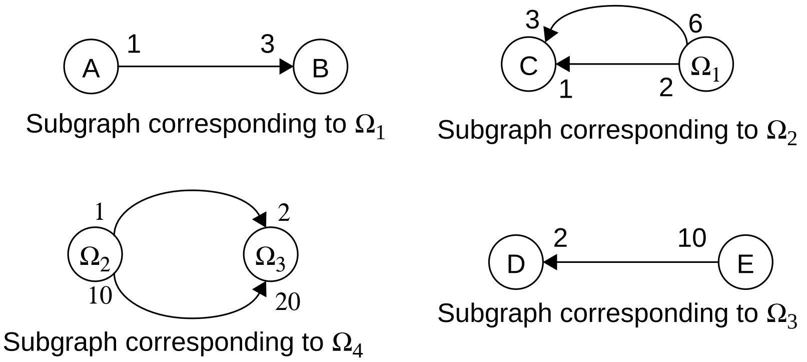


Figure 3-19 An illustration of APGAN.

In Figure 3-19(c), we have $\mathbf{q}(\Omega_2, D, E) = (2, 5, 1)$, and thus all three adjacent pairs have $\rho = 1$. Among these, clearly, only $\{\Omega_2, E\}$ and $\{E, D\}$ do not introduce cycles, so APGAN arbitrarily selects among these two to determine the third clusterization pair. Figure 3-19(d) shows the graph that results when $\{E, D\}$ is chosen. This graph contains only one adjacent pair $\{\Omega_2, \Omega_3\}$, and APGAN will consolidate this pair in its final clusterization step to obtain the single-vertex graph in Figure 3-19(e).

Figures 3-19(a-e) specify the sequence of clusterizations performed by APGAN when applied to the graph of Figure 3-19(a). A more compact representation of this sequence is shown in Figure 3-20. A valid single appearance schedule for Figure 3-19(a) can easily be constructed by recursively traversing the hierarchy induced by this sequence. We start by constructing a schedule for the top-level subgraph, the subgraph corresponding to Ω_4. The subgraph G_i corresponding to each Ω_i consists of only two actors X_i and Y_i, such that all edges in G_i are directed from X_i to Y_i. Thus, it is clear how an optimal schedule can easily be constructed for the subgraph corresponding to each Ω_i: if each edge e in G_i satisfies $del(e) \ge \eta(e)$, then we construct the schedule $(\mathbf{q}_{G_i}(Y_i)Y_i)(\mathbf{q}_{G_i}(X_i)X_i)$, and otherwise we construct $(\mathbf{q}_{G_i}(X_i)X_i)(\mathbf{q}_{G_i}(Y_i)Y_i)$. In Figure 3-19, This yields the "top-level" schedule $(2\Omega_2)\Omega_3$ (we suppress loops that have an iteration count of one) for the subgraph corresponding to Ω_4.

Next, we recursively descend one level in the cluster hierarchy to the subgraph corresponding to Ω_3, and we obtain the schedule $E(5D)$. Since this subgraph contains no hierarchical actors, $E(5D)$ is immediately returned as the "flattened" schedule for the



Subgraph corresponding to Ω_1

Subgraph corresponding to Ω_2

Subgraph corresponding to Ω_4

Subgraph corresponding to Ω_3

Figure 3-20 A compact representation for the clustering sequence shown in Figure 3-19

subgraph corresponding to Ω_3. This flattened schedule then replaces its corresponding hierarchical actor in the top-level schedule, and the top-level schedule becomes $(2\Omega_2)E(5D)$.

Next, descending to Ω_2, we construct the schedule $\Omega_1(2C)$ for the corresponding subgraph. We then examine the subgraph corresponding to Ω_1 to obtain the schedule $(3A)B$. Substituting this for Ω_1, the schedule for the subgraph corresponding to Ω_2 becomes $(3A)B(2C)$. Finally, this schedule gets substituted for Ω_2 in the top-level schedule to yield the valid single appearance schedule $S_p \equiv (2((3A)B(2C)))E(5D)$ for Figure 3-19(a).

From S_p and Figure 3-19(a) it is easily verified that $\mathit{buffer\_memory}(S_p)$ and

$$\left(\sum_{e \in E} \mathit{BMLB}(e)\right),$$

where E is the set of edges in Figure 3-19(a), are equal to 43, and thus in the execution of APGAN illustrated in Figure 3-19, a BMLB schedule is constructed.

As seen in the above example, the APGAN approach, as has been defined here, does not uniquely specify the sequence of clusterizations that will be performed, and it does not in general, result in a unique schedule for a given SDF graph. The APGAN technique together with an unambiguous protocol for deciding between adjacent pairs that are tied for the highest repetition count form an **APGAN instance**, which generates a unique schedule for a given graph. For example, one tie-breaking protocol that can be used when actors are labelled alphabetically, as in Figure 3-19, is to choose that adjacent pair that maximizes the sum of the “distances” of the actor labels from the letter “A”. If this protocol is used to break the tie between $\{A, B\}$ (“distance sum” is $0 + 1 = 1$) and $\{B, C\}$ (distance sum is $1 + 2 = 3$) in the first clusterization of step of Figure 3-19, then $\{B, C\}$ is chosen.

We say that an adjacent pair is an **APGAN candidate** if it does not introduce a cycle, and its repetition count is greater than or equal to all other adjacent pairs that do not introduce cycles. Thus, an APGAN instance is any algorithm that takes a consistent, acyclic SDF graph as input, repeatedly clusters APGAN candidates, and then outputs the schedule corresponding to a recursive traversal of the resulting cluster hierarchy.

In [Bhat96a], the following result is proven:

Theorem 3-7: Suppose that $G = (V, E)$ is a connected, consistent, acyclic SDF graph that has a BMLB schedule; $del(e) < \eta(e)$ for all $e \in E$; P is an APGAN instance; and $S_P(G)$ is the schedule obtained by applying P to G. Then $S_P(G)$ is a BMLB schedule for G.

As a consequence, all graphs, such as that shown in Figure 3-19(a), for which BMLB schedules exist, are handled optimally by any APGAN instance. For example, even if in a certain APGAN instance, $\{B, C\}$ is clustered instead of $\{A, B\}$ in the first clusterization step of Figure 3-19, we are still guaranteed that the final result achieved by that APGAN instance will be a BMLB schedule. If the graph does not have a BMLB schedule, then it is not necessary that APGAN will find the best possible schedule. In other words, APGAN finds the best schedule whenever the achievable lower bound for the SDF graph (that is, the smallest buffer-memory requirement that can be met by some valid single appearance schedule) is equal to the BMLB. While the above result has considerable intellectual interest by itself, we also demonstrate the practical relevance of this optimality result in Section 3.20 by giving practical applications to which the result applies. Also, experimental data will be presented that suggests that a particular APGAN instance frequently produces excellent results even for applications that do not have BMLB schedules, and it will be shown that it has exhibited encouraging performance on a large collection of complex randomly-generated SDF graphs.

3.20 Examples

3.20.1 Tree-structured Filter Bank

Figure 3-17 shows an SDF graph abstraction of a uniform-tree structured QMF filterbank [Vaid93]. This type of filterbank is commonly used in practice for audio coding applications. Filterbanks like this, even with arbitrary “depth”, where depth is defined to be the logarithm of the number of channels in the middle of the graph, fall into the class of SDF graphs that have BMLB schedules; hence, APGAN will always return an optimal buffer schedule for these graphs.

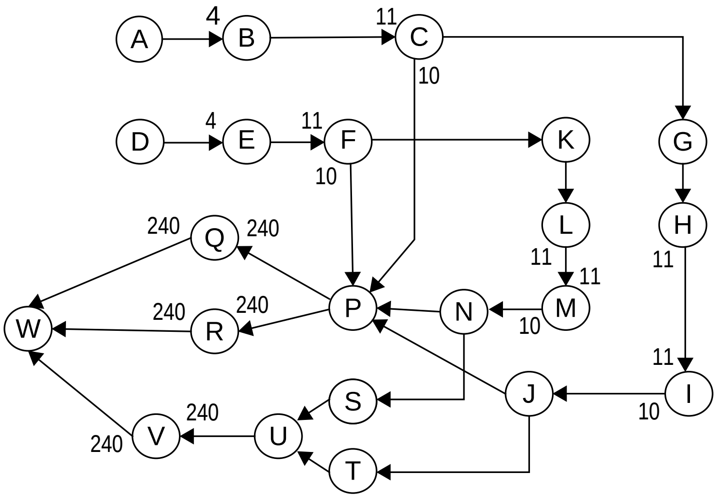
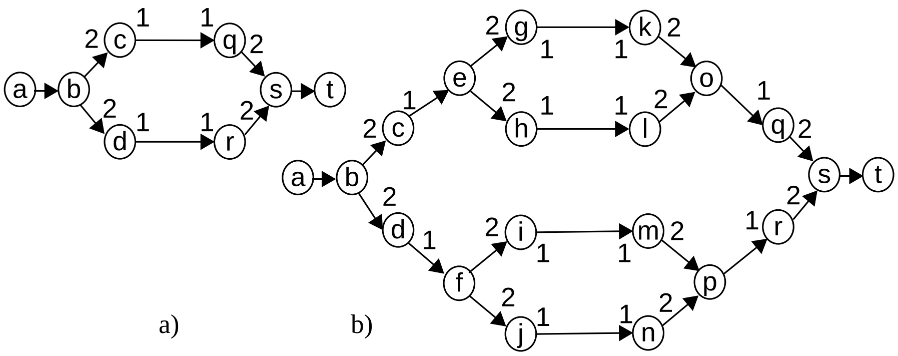


Figure 3-22 SDF abstraction for satellite receiver application from [Ritz95]

3.20.2 Satellite Receiver

In the example above, the graph had a very symmetric and regular topology. Figure 3-22 shows an example where the topology is not as regular, but is still in the class of SDF graphs that have a BMLB schedule. The graph is an abstraction for a satellite receiver implementation and is taken from [Ritz95]. The graph is annotated with the produced/ consumed numbers wherever they are different from unity. It is interesting to note that a shared-buffer implementation of an optimal flat single appearance schedule for this graph would require a buffer of size 2040 [Ritz95] while APGAN generates a BMLB schedule having a total buffering requirement of 1540 (using a seperate buffer on every edge of-course).



a) b)

Figure 3-21 SDF graph for a uniform-tree filterbank. a) Depth 1 filterbank, b) Depth 2 filterbank. The produced/consumed numbers not specified are all unity.

3.20.3 Chain Structured Graph with Irregular Rate Changes

When sample rates are irregular, APGAN can do poorly even if the topology is very simple. For the graph shown in Figure 3-23, it can be verified that APGAN will construct the schedule $(9A)(4(3BC)2D)$, which has a buffering cost of 43, while the optimal schedule, returned by GDPPO is $(3(3A)(4B))(4(3C)(2D))$, which has a cost of 30. The optimal schedule would also be returned by RPMC (without GDPPO) in this case since edge BC is where the minimum amount of data is transferred in a complete period of the schedule.

The tables in Section 3.21 have more examples illustrating the performance of the APGAN algorithm. The tables have both the performance on practical examples and performance on random graphs. As noted there, APGAN generally performs well when the topology or rate changes are fairly regular. Such regularity arises frequently in practical multirate SDF graphs. In graphs that contain significant irregularity, such as random graphs and the non-uniform filterbanks, RPMC usually performs much better than APGAN. Hence, these two heuristics complement each other well.

3.21 Experiments

Table 3-1 shows the results of applying GDPPO to the schedules generated by APGAN and RPMC on several practical SDF systems. The columns labeled “% Impr.” show the percentage of buffer memory reduction obtained by GDPPO. The QMF tree filter banks fall into a class of graphs for which APGAN is guaranteed to produce optimal results, and thus there is no room for GDPPO to produce improvement when APGAN is applied to these two examples. Overall, GDPPO produces an improvement in 11 out of the 14 heuristic/application combinations. A “significant” (greater than 5%) improvement is obtained in 9 of the 14 combinations; the mean improvement over all 14 combinations is

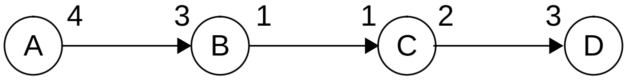


Figure 3-23 A chain-structured SDF graph.

Table 3-1: Performance of GDPPO on several practical SDF systems.

Application	Apgan only	Apgan + Gdppo	% Impr.	Rpmc only	Rpmc + Gdppo	% Impr.
Nonuniform filter bank (1/3, 2/3 splits, 4 channels)	153	137	10.5	131	128	2.34
Nonuniform filter bank (1/3, 2/3 splits, 6 channels)	856	756	11.7	690	589	14.6
QMF tree filter bank (8 channels)	78	78	0	92	87	5.43
QMF tree filter bank (16 channels)	166	166	0	218	200	8.26
Two-stage fractional decimation system	140	119	15.0	133	133	0
CD-DAT sample rate conversion	396	382	3.54	535	400	25.2
DAT-CD sample rate conversion	205	182	11.2	275	191	30.5

9.9%; and from the CD-DAT and DAT-CD examples, we see that it is possible to obtain very large reductions in the buffer memory requirement with GDPPO.

Table 3-2 shows experimental results on the performance of APGAN and RPMC for several practical examples of acyclic, multirate SDF graphs. The column titled “average random” represents the average buffer memory requirement obtained by considering 100 **random schedules**. A random schedule is generated by generating a random topological sort. Corresponding to this topological sort is a flat single appearance schedule; this is a BMUB schedule. Then GDPPO is applied to this schedule to get a single appearance schedule whose buffer memory requirement is less than or equal to that of all single appearance schedules having this particular topological ordering of the actors. All of the systems shown in the table are acyclic graphs. The data for APGAN and RPMC also includes the effect of GDPPO. As can be seen, APGAN achieves the BMLB on 5 of the 9 examples, outperforming RPMC in these cases. Particularly interesting are the last three examples in the table, which illustrate the performance of the two heuristics as the graph sizes are increased. The graphs represent a symmetric tree-structured QMF filterbank with differing depths. APGAN constructs a BMLB schedule for each of these systems while

Table 3-2: Performance of the two heuristics on various acyclic graphs.

System	BMUB	BMLB	Apgan	Rpmc	Avg. Rand.	# vert./ # edges
Fractional decimation	61	47	47	52	52	26/30
Laplacian pyramid	115	95	99	99	102	12/13
Nonuniform filterbank (1/3,2/3 splits) (4 channels)	466	85	137	128	172	27/29
Nonuniform filterbank (1/3,2/3 splits) (6 channels)	4853	224	756	589	1025	43/47
QMF nonuniform-tree filterbank	284	154	160	171	177	42/45
QMF filterbank (one-sided tree)	162	102	108	110	112	20/22
QMF analysis only	248	35	35	35	43	26/25
QMF Tree filterbank (4 channels)	84	46	46	55	53	32/34
QMF Tree filterbank (8 channels)	152	78	78	87	93	44/50
QMF Tree filterbank (16 channels)	400	166	166	200	227	92/106

RPMC generates schedules that have buffer memory requirements about 1.2 times the optimal. Conversely, the third and fourth entries show that RPMC can outperform APGAN significantly on graphs that have more irregular rate changes. These graphs represent nonuniform filterbanks with differing depths. In the table, for each example, the cell corresponding to the heuristic that gave the best buffer memory requirement has been shaded. If the best performer equals the BMLB, then the shading is the same as the BMLB column; otherwise a darker shade has been used.

Table 3-3 shows more detailed statistics for the performance of randomly obtained topological sorts. For example, the column titled "APGAN < random" represents the number of random schedules (again, these are obtained by starting with a BMUB schedule for a random topological sort and applying GDPPO to this schedule) that had a buffer memory requirement greater than that obtained by APGAN. The last two columns give the mean number of random schedules needed to outperform these heuristics. A dash indicates

Table 3-3: Performance of 100 random schedules against the heuristics

Comparison with random schedules (100 trials)	Apgan< random	Apgan= random	Rpmc < random	Rpmc = random	Avg. to beat Apgan	Avg. to beat Rpmc
Fractional decimation	92%	8%	54%	13%	----	3
Laplacian pyramid	74%	26%	74%	26%	----	----
Nonuniform filterbank (1/3,2/3 splits) (4 channels)	100%	0%	100%	0%	----	----
Nonuniform filterbank (1/3,2/3 splits) (6 channels)	100%	0%	100%	0%	----	----
QMF nonuniform-tree filterbank	100%	0%	81%	7%	----	8
QMF filterbank (one-sided tree)	100%	0%	77%	23%	----	----
QMF analysis only	99%	1%	99%	1%	----	----
QMF Tree filterbank (4 channels)	100%	0%	16%	13%	----	1.4
QMF Tree filterbank (8 channels)	100%	0%	87%	3%	----	9.1
QMF Tree filterbank (16 channels)	100%	0%	96%	1%	----	22.3

that no random schedules were found that had a buffer memory requirement lower that obtained by the corresponding heuristic.

While the above results on practical examples are encouraging, these heuristics have also been tested on a large number of randomly generated 50-actor acyclic SDF graphs. These graphs were sparse, having about 100 edges on average. The SDF parameters were chosen randomly according to the following rules. Firstly, it is determined whether a parameter is a “free variable” or not; it is a free variable if assigning an arbitrary number to the parameter does not lead to sample rate inconsistency. If the parameter is a “free variable”, then with probability 0.5, it is set to 1, and with probability 0.5, it is set to a uniformly generated random number between 1 and 10. Table 3-4 summarizes the performance of these heuristics, both against each other, and against randomly generated schedules. As can be seen, RPMC outperforms APGAN on these random graphs almost two-thirds of the time.These heuristics were compared against 2 random schedules because

Table 3-4: Performance of the two heuristics on random graphs

RPMC < APGAN	63%
APGAN < RPMC	37%
RPMC < min(2 random)	83%
APGAN < min(2 random)	68%
RPMC < min(4 random)	75%
APGAN < min(4 random)	61%
min(RPMC,APGAN) < min(4 random)	87%
RPMC < APGAN by more than 10%	45%
RPMC < APGAN by more than 20%	35%
APGAN < RPMC by more than 10%	23%
APGAN < RPMC by more than 20%	14%

measurements of the actual running time on 50-vertex graphs showed that we can construct and examine approximately 2 random schedules and post-optimize it by GDPPO in the time it takes for either APGAN or RPMC to construct its schedule and have it post-optimized by GDPPO. The comparison against 4 random schedules shows that in general, the relative performance of these heuristics goes down if a large number of random schedules are inspected. Of course, this also entails a proportionate increase in running time. However, as shown on practical examples already, it is unlikely that even picking a large number of schedules randomly will give better results than these heuristics since practical graphs usually have a significant amount of structure (as opposed to random graphs) that the heuristics can exploit well. Thus, the comparisons against random graphs give a worst case estimate of the performance that can be expected from these heuristics.

The experimental results presented in this section show that APGAN and RPMC complement each other. For the practical SDF graphs that were examined, APGAN performs well on graphs that have a simple structure topologically and regular rate changes, like the uniform QMF filterbanks, and RPMC performs well on graphs that have more irregular rate changes and irregular topologies. Since large random graphs can be expected to consistently have irregular rate changes and topologies, the average performance on random graphs of RPMC is better than APGAN by a wide margin — although, from the last two rows of Table 3-4, it can be seen that there is a significant proportion of random graphs for which APGAN outperforms RPMC by a margin of over 10%, which suggests that APGAN is a useful complement to RPMC even when mostly irregular graphs are encountered. However, the main advantage of adopting both APGAN and RPMC as a combined solution arises from complementing the strong performance of RPMC on general graphs with the formal properties of APGAN, as specified by Theorem 3-7, and the ability of APGAN to exploit regularity that arises frequently in practical applications.

There is a variation of APGAN that achieves significantly better performance on random graphs than the original version, although still significantly worse performance as compared to RPMC. This variation arises from changing the "priority function" associated with an edge e from $\rho(\{src(e), snk(e)\})$ to the product

$$TNSE(e) \times \rho(\{src(e), snk(e)\}) . \tag{3.17}$$

In other words, the variation repeatedly clusters the source and sink vertices of edges that maximize the measure given by (3.17). Thus, an adjacent pair is given more weight if a large amount of data is transferred between it as compared to other adjacent pairs.

We have found that on the random graphs that were used to generate Table 3-4, this modification of APGAN outperforms two random schedules ("min(2 random)") 76.5 percent of the time, which indicates a level of performance intermediate to APGAN and RPMC. Furthermore, its performance equaled the performance of APGAN on all of the practical examples except the six channel nonuniform filter bank, where it achieved a buffer memory requirement of 696 (8% better than APGAN), and the four channel nonuniform filter bank, where it achieved 136 (0.7% better than APGAN).

Interestingly, however, the modification of APGAN corresponding to (3.17) does not preserve the formal properties specified by Theorem 3-7. This is easily seen from the example in Figure 3-24. Here, $\mathbf{q}(W, X, Y) = (2, 2, 1)$, and thus $\rho(\{W, X\}) = 2$ and $\rho(\{W, Y\}) = 1$, and if we let $\hat{\rho}$ denote the measure defined by (3.17), then $\hat{\rho}(\{W, X\}) = 2 \times 2 = 4$, while $\hat{\rho}(\{W, Y\}) = 6 \times 1 = 6$. We see then that APGAN clusters $\{W, X\}$ in its first clusterization step, which leads to the final schedule $(2WX)Y$, and a buffer memory requirement of 7, while in the variation of APGAN, $\{W, Y\}$ is clustered first, and the resulting schedule $(2W)Y(2X)$ gives a buffer memory requirement 8. It is easily verified the BMLB for this graph is 7, and thus, APGAN generates a BMLB schedule, while the variation generates a suboptimal result.

Thus, the variation of APGAN introduces a trade-off between provable optimality for a class of graphs, and average-case performance. Since we are proposing to complement a heuristic — RPMC — whose average case performance significantly outweighs that of both APGAN and its variation, it is intuitively more appealing to choose the original version of APGAN since it adds a feature that RPMC lacks — optimality for a restricted, but useful, class of graphs. For the practical examples that were examined, the variation of APGAN outperformed the original APGAN only in cases where RPMC outperformed both APGAN and the APGAN variation, and thus adopting the new version of APGAN does not improve the final result of any of these examples when a combined solution with RPMC is employed.

3.22 Extension to Arbitrary Graphs

As shown in Chapter 2, the problem of constructing single appearance schedules that minimize buffer memory usage for arbitrary SDF graphs is NP-hard since the problem is NP-hard for HSDF graphs (where any valid schedule of blocking factor 1 is a single appearance schedule). However, we can use the heuristics introduced in this chapter to a limited extent on arbitrary SDF graphs as well.

First we review some pertinent results about single appearance scheduling for arbitrary graphs from [Bhat94b].

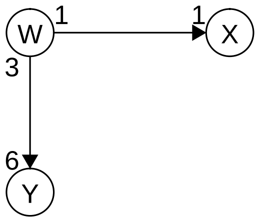


Figure 3-24 An example in which APGAN achieves the BMLB, but the modified version corresponding to (3.17) does not.

Definition 3-7: Suppose that G is a connected, sample rate consistent SDF graph. If Z_1 and Z_2 are disjoint nonempty subsets of $actors(G)$, we say that Z_1 *is* **subindependent** *of* Z_2 *in* G if for every edge α in G such that $src(\alpha) \in Z_2$ and $snk(\alpha) \in Z_1$, we have

$$del(\alpha) \geq TNSE(\alpha, G). \qquad (3.18)$$

We occasionally drop the "in G" qualification if G is understood from context. Also, if (Z_1 is subindependent of Z_2) **and** ($Z_1 \cup Z_2 = actors(G)$), then we say that Z_1 *is subindependent in* G, and we say that Z_1 and Z_2 form a **subindependent partition** of G.

Definition 3-8: A graph is called loosely interdependent if it does not have a subindependent partition and tightly interdependent otherwise.

The subindependent partition can be computed for a graph in time linear in the total number of edges of the graph [Bhat94b]. It basically removes from the graph, all edges that satisfy (3.18). Figure 3-25 shows the subindependent partition of an SDF graph. Intuitively, the idea behind subindependent partitioning is that the strongly connected components that result after edges satisfying (3.18) have been removed can each be clustered together, to give a graph that is acyclic. Hence, this acyclic graph has a single appearance schedule if each of the strongly connected components has a single appearance schedule. In fact, the following stronger result is proven in [Bhat94b]:

Theorem 3-8: A nontrivial, strongly connected, consistent SDF graph G has a single appearance schedule if and only if every nontrivial strongly connected subgraph of G is loosely interdependent.

This idea forms the basis of the loop scheduling framework developed in [Bhat94b] where the SDF graph is recursively broken up into strongly connected

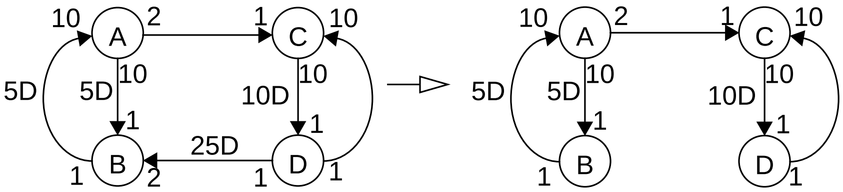


Figure 3-25 An illustration of algorithm *SubindependentPartition*.

components via subindependent partitioning until either no more strongly connected components are encountered or a strongly connected component no longer has a subindependent partition. The loop scheduling framework makes use of 3 algorithms: the subindependence partitioning algorithm, the acyclic scheduling algorithm, and the tight interdependence scheduling algorithm. It is shown in [Bhat94b] that the loop scheduling framework is modular in the sense that the three algorithms do not interact with one another either destructively or constructively; in other words, the acyclic scheduling algorithm cannot affect the performance of the subindependence partitioning algorithm or vice-versa insofar as the existence of single appearance schedules are concerned. Thus, a loose interdependence algorithm always obtains an optimally compact solution when a single appearance schedule exists. When a single appearance schedule does not exist, strongly connected graphs are repeatedly decomposed until tightly interdependent subgraphs are found.

The algorithms and heuristics described in this chapter for buffer-memory-optimal single appearance schedules can be used as the acyclic component in the loop scheduling framework to yield schedules better optimized for buffer-memory usage. However, this approach will not even be able to optimize over the space of all possible single appearance schedules; hence, there is no hope of optimality at all. Figure 3-26 makes this point clearer. The repetitions vector is given by $\mathbf{q}(A, B, C, D, E, F) = [12, 16, 18, 6, 18, 9]$. The Subindependence algorithm will produce the graph on the right after deleting the arcs DB and FD. The acyclic algorithm will now construct the schedule $(4\Omega_1)(6\Omega_2)(9\Omega_3)$, and this schedule can be optimally nested by GDPPO at the top-most level. Hence, the only two schedules that GDPPO could construct would be $2(2\Omega_1 3\Omega_2)(9\Omega_3)$ or $4\Omega_1 3(2\Omega_2 3\Omega_3)$. It is easy to verify that the Ω_i have the schedules $3A4B$, $(3C)(D)$, and $(2E)(F)$

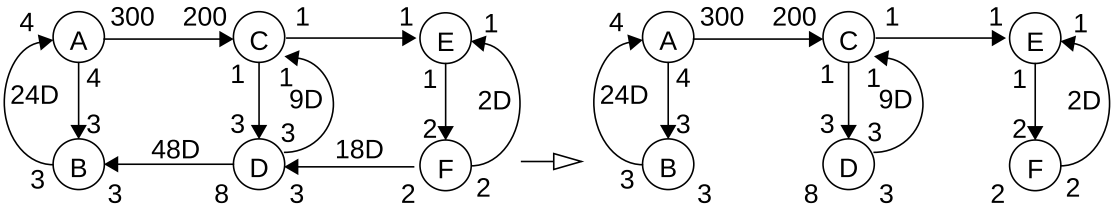


Figure 3-26 An example to illustrate the inability of the scheduling framework in [Bhat94b] to yield buffer-optimal single appearance schedules when combined with techniques presented in this chapter.

respectively for $i = 1, 2, 3$, and the overall schedule returned by the loop scheduling framework has these schedules substituted in the two top-level schedules for the Ω_i, $i = 1, 2, 3$. However, the schedule $2(3(2A3C)8B\ \ 3D)\ \ 9((2E)\ F)$ has lower buffering requirements since only 600 locations are needed on arc AC instead of the 1800 or more required by the other two schedules. It is easy to see that there is no way to get this schedule from the loop scheduling framework as described. So, our optimization for buffer memory usage is being done over a proper subset of the space of all possible single appearance schedules, and this is clearly sub-optimal. Hence, to fully incorporate buffer-memory considerations when constructing single appearance schedules for arbitrary SDF graphs, either the subindependence algorithm has to be modified non-trivially, or a new framework has to be designed. It would be interesting to investigate this issue in the future; however, the interest might be largely theoretical since there seem to be few practical SDF systems that have several multirate cycles.

3.23 Related Work

Some work has been done with SDF scheduling by other researchers; in this section, we describe some of these results and the specific objectives addressed

3.23.1 Minimum Activation Schedules

At Aachen, Professor Heinrich Meyr's group has studied the problem of constructing single appearance schedules that attempt to minimize context-switch overhead. These schedules are called minimum activation schedules and have been used in the COSSAP environment, now marketed by Synopsys. In a schedule, each time a new actor is invoked, there is a context switch; this overhead includes saving the contents of registers, and loading state variables and buffer pointers. In the code generation environment described in [Ritz93], this overhead includes all the usual ones associated with function calls since the code generated by their system is not inline. Hence, the objective in minimum activation scheduling is to minimize the number of actor invocations that occur in the schedule. For example, the schedule $(2(2B)(5A))(5C)$ has 5 invocations per schedule period while the "flat" version of the schedule, $(4B)(10A)(5C)$ has 3 invocations per schedule period. The average rate of activations for a periodic schedule is defined to be the number of activations divided by the blocking factor of the schedule. If we increase the blocking factor to 2, it is easily verified that the average activation rate for the two schedules above become 4.5 and 1.5 respectively. In general, for any consistent acyclic graph, we can make the average activation rate arbitrarily close to zero by using a flat single appearance schedule of arbitrarily high blocking factor. Thus, the problem becomes more interesting when the SDF graph has cycles. Algorithms (that in general are not polynomial time) are given in [Ritz93] that attempt to find minimum activation schedules for arbitrary SDF graphs. However, it turns out that non-single appearance schedules can have a lower average activation, but because code-size is also a primary constraint in the COSSAP code-generation system, Ritz et. al. only consider single appearance schedules.

As pointed out before, we favor nested single appearance schedules over flat schedules because our secondary concern was for buffer minimization, while Ritz et. al. have the average activation rate as the secondary objective. In [Ritz95], an attempt is made to address the buffering requirements by introducing it as a tertiary objective. The strategy is to focus on delayless acyclic graphs and construct flat single appearance schedules (flat because this minimizes the average activation rate) that minimize the total amount of buffer memory required when buffer-sharing techniques are used. Of-course, computing a schedule that minimizes the total number of live tokens at any stage in the schedule is an NP-hard problem even for acyclic HSDF graphs [Seth75]. The heuristic given in [Ritz95] is not even guaranteed to run in polynomial time, is not optimal, and it is not clear whether the complicated buffer-sharing strategy for a multirate SDF graph can be implemented easily in practice. Also, as pointed out in Section 3.2, buffer-sharing with flat single appearance schedules can yield total buffering requirements that are much greater than buffering requirements without buffer sharing but allowing nested schedules. Moreover, the presence of delays causes additional problems; for this reason, the technique in [Ritz95] applies only to delayless SDF graphs. The satellite receiver example of Section 3.20.2 shows that the APGAN+DPPO combination is able to achieve lower requirements than the heuristic presented in [Ritz95]. Hence, if buffer minimization is a secondary criterion, then the techniques developed in this chapter will in general be superior to those in [Ritz95]. There is clearly a trade-off in buffering requirements, context-switch overhead, generality of various buffering schemes, code size, and latency; in practice, it is likely that each of these considerations will be important sometimes. Hence, a good design tool should have a suite of all these schedulers so that the appropriate one may be chosen by the designer.

3.23.2 Retiming for State Minimization

Retiming is a technique that moves delays around in the graph in order to minimize the critical path in the graph. This technique was originally proposed [Leis91] in the context of optimizing the throughput of synchronous circuits, and in this context, the critical path, that is, the longest delay-free path in the circuit, determines the smallest period with which the circuit can be clocked. It is shown in [Leis91] that a secondary criterion of minimizing the total number of state; that is, delays on the arcs, can be incorporated into the retiming framework, and retimings that minimize the clock period along with total state can be computed efficiently. These techniques are not that useful to the problem that we have considered in this chapter since they do not attempt to construct schedules that minimize the total amount of buffering required. They also do not apply to multirate SDF graphs.

3.23.3 Lee’s Trellis-based Formulation

In [Lee86], Lee presents a technique by which schedules that minimize the amount of buffering can be constructed, both under the model where a separate buffer is assigned to each arc, or to the case where buffer sharing is employed. Lee associates a vector with the graph with dimension of the vector equalling the number of vertices. Each element of the vector represents the number of times each vertex has been invoked in a partial schedule. Starting from the 0 vector, we explore all the next vectors that result from firing one of the firable vertices. If we represent the vector by a vertex, and associate an edge between the vertex corresponding to a vector and the vertex corresponding to its successor, the edge can be weighted by the amount of buffering required until then using either of the buffer requirement criteria. Once all of the vectors have been computed, the minimum memory schedule is simply the least-weight path through this graph. The chief drawback of this approach is the exponential number of vectors that could be visited, even for an HSDF graph; hence, this approach would not be practical for large graphs. Of-course, for an multirate SDF graph, this approach does not even yield single appearance schedules.

3.24 Summary

In this chapter, we have tackled the problem of computing schedules that minimize code size as a primary goal, with buffer memory minimization as a secondary goal. Code-size minimization is achieved by restricting our attention to single appearance schedules. Buffer minimization is achieved by factoring single appearance schedules; that is, by organizing nested loops in the schedule. We have argued in favor of a buffering model where there is a buffer on each edge in the graph. We have shown that when the graph is well-ordered, meaning there is only one topological sort, the problem of finding a single appearance schedule that minimizes the total buffer memory can be solved in polynomial time by a dynamic programming algorithm. We show that the dynamic programming algorithm can be extended in several interesting ways to apply to arbitrary schedules (both single appearance and non-single appearance). We also show how it can be extended to apply to a restricted model of buffer sharing. We have shown that the buffer minimization problem becomes NP-complete for general acyclic graphs, and hence heuristics must be used. We give two heuristics: RPMC, and APGAN, and present an extensive experimental study to demonstrate the strengths and weaknesses of these two heuristics. The APGAN heuristic is optimal for a certain class of acyclic SDF graphs; this class is shown to be of practical value since several practical SDF systems fall into the class. Finally, we extend all of the algorithms to apply to cyclic graphs in certain restricted ways.

These algorithms have all been implemented in the Ptolemy programming environment. The APGAN approach has been implemented by the Alta Group of Cadence Design Systems Inc. in their Signal Processing Worksystem programming environment.

Most of the work reported in this chapter has been done in collaboration with Dr. Shuvra Bhattacharyya, a researcher at the Hitachi Systems Research Laboratories in San Jose, Ca. In addition, various subsets have appeared, or will appear, in published form in journals and conferences: [Bhat95, Bhat96b, Murt94a, Murt94c]. The original loop scheduling problem (without the goal of joint buffer minimization) was developed in [How90], [Bhat93] and [Bhat94b]. The approach in [Ho88a] had serious limitations and flaws; see [Bhat94b] for a summary. Most of the work in [Bhat94b, Bhat95, Bhat96b, Murt94a, Murt94c] can be found in the book [Bhat96a].

4

Multidimensional Synchronous Dataflow

The Synchronous dataflow model suffers from the limitation that its streams are one-dimensional. For multidimensional signal processing algorithms, it is necessary to have a model that where this restriction is not there, so that effective use can be made of the inherent data-parallelism that exists in such systems. As for one-dimensional systems, the specification model for multidimensional systems should expose to the compiler or hardware synthesis tool as much static information as possible so that run-time decision making is avoided as much as possible, and so that effective use can be made of both functional and data parallelism. Most multidimensional signal processing systems also have a predictable flow of control, like one-dimensional systems, and for this reason, an extension of SDF, called multidimensional synchronous dataflow was proposed in [Lee93].

4.1 Multidimensional Dataflow

The standard dataflow model suffers from the limitation that its streams are one dimensional. Although a multidimensional stream can be embedded within a one dimensional stream, it may be awkward to do so [Chen94]. In particular, compile-time information about the flow of control may not be immediately evident. The multidimensional SDF model is a straightforward extension of one-dimensional SDF. Figure 4-1 shows a trivially simple two-dimensional SDF graph. The number of tokens

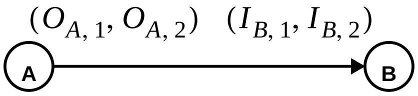


Figure 4-1 A simple MD-SDF graph.

produced and consumed are now given as M-tuples. Instead of one balance equation for each edge, there are now M. The balance equations for figure 4-1 are

$$r_{A,1}O_{A,1} = r_{B,1}I_{B,1} \tag{4.1}$$

$$r_{A,2}O_{A,2} = r_{B,2}I_{B,2} \tag{4.2}$$

These equations should be solved for the smallest integers $r_{X,i}$, which then give the number of repetitions of actor X in dimension i.

4.1.1 Application to Multidimensional Signal Processing

As a simple application of MD-SDF, consider a portion of an image coding system that takes a 40×48 pixel image and divides it into 8×8 blocks on which it computes a DCT. At the top level of the hierarchy, the dataflow graph is shown in figure 4-2. The solution to the balance equations is given by

$$r_{A,1} = r_{A,2} = 1, r_{DCT,1} = 5, r_{DCT,2} = 6. \tag{4.3}$$

A segment of the index space for the stream on the edge connecting actor A to the DCT is shown in the figure. The segment corresponds to one firing of actor A. The space is divided into regions of tokens that are consumed on each of the five vertical firings of

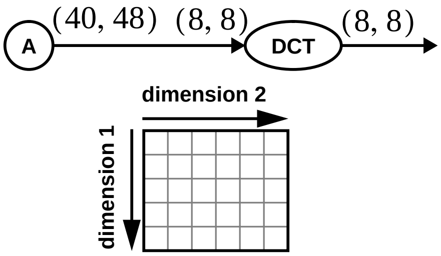


Figure 4-2 An image processing application in MD-SDF.

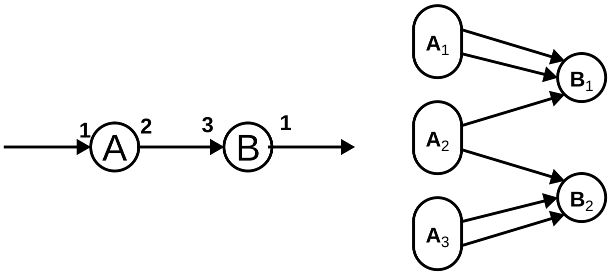


Figure 4-3 An SDF graph and its corresponding precedence graph.

each of the 6 horizontal firings. The precedence graph constructed automatically from this shows that the 30 firings of the DCT are independent of one another, and hence can proceed in parallel. Distribution of data to these independent firings can be automated.

4.1.2 Flexible Data Exchange

While the utility of MDSDF to image processing is obvious, a less obvious application of MDSDF is a flexible data exchange mechanism. Consider the graph in figure 4-3. This graph shows that an interesting type of control flow can be specified using an SDF graph. Figure 4-3 shows two actors with a 2/3 producer/consumer relationship. The precedence graph is shown on the right. Note that the first firing of A produces two samples consumed by the first firing of B. Suppose instead that we wish for firing A_1 to produce the first sample for each of B_1 and B_2. This can be obtained using MD-SDF as shown in figure 4-4. Here, each firing of A produces data consumed by each firing of B, resulting in a pattern of data exchange quite different from that in figure 4-3. The precedence graph in figure 4-4 shows this. Also shown is the index space of the tokens transferred along the

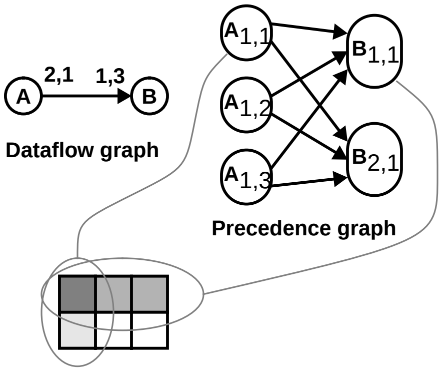


Figure 4-4 Data exchange in an MD-SDF graph.

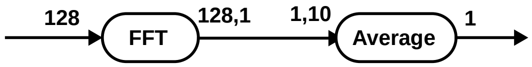


Figure 4-5 Averaging successive FFTs using MD-SDF.

edge, with the shaded regions indicating the tokens produced by the first firing of A and consumed by the first firing of B.

A DSP application of this more flexible data exchange is shown in figure 4-5. Here, ten successive FFTs are averaged. Averaging in each frequency bin is independent and hence may proceed in parallel. The ten successive FFTs are also independent, so if all input samples are available, they too may proceed in parallel. Notice here that the second dimension is being used as a temporary dimension to lay out data in a way more suitable to the particular application. The analogy is that a pancake cannot be flipped over without throwing it up in the air; that is, by using a third dimension. Hence, extra dimensions are needed sometimes as “workspace”; this is exactly one of the uses in Lucid as well. The matrix multiplication specification in [Lee93] is another example where a third dimension is used to lay out data in a form more amenable to the inner product computation involved in matrix multiplication.

4.1.2.1 Multilayer Perceptron

A more complicated example of how the flexible data-exchange mechanism in an MDSDF graph can be useful in practice is shown in figure 4-6, which shows how a multilayer perceptron (with a nodes in the first layer, b nodes in the second layer etc.) can be specified in a succinct way. However, as the precedence graph shows, none of the parallelism in the network is lost; it can be easily exploited by a good scheduler. Note that the net specified below is used only for computation once the weights have been trained. Specifying the training mechanism as well would require feedback edges with the appropriate delays and some control constructs; we do not develop such a system here.

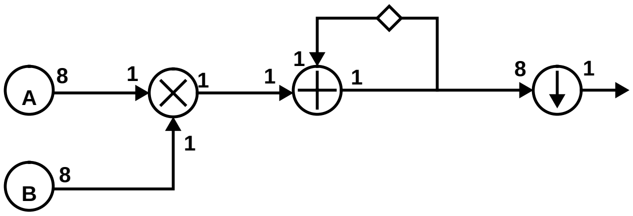


Figure 4-7 An attempt to use 1D-SDF to repeatedly compute inner products.

4.1.3 Computing Inner Products [Lee93]

Consider the problem of repeatedly computing an inner product on a stream of vectors. This can be easily generalized into an FIR filter, although for conciseness we will stick to the generic inner product. In particular, suppose we wish to express the inner product at its finest level of granularity, and further that we require the graphical representation to have a structure that is independent of the size of the vectors. To express this using 1D-SDF, we might try the configuration shown in figure 4-7. Actors A and B each supply vectors of length 8 by producing 8 tokens when they fire. The small white diamond is a “delay”, which in a dataflow context is simply an initial, zero-valued token on the edge. The actor with the downward arrow is a “downsample.” It simply consumes 8 tokens and outputs one of them, discarding the rest. This configuration will correctly compute the first inner product, but when the second set of vectors are generated by

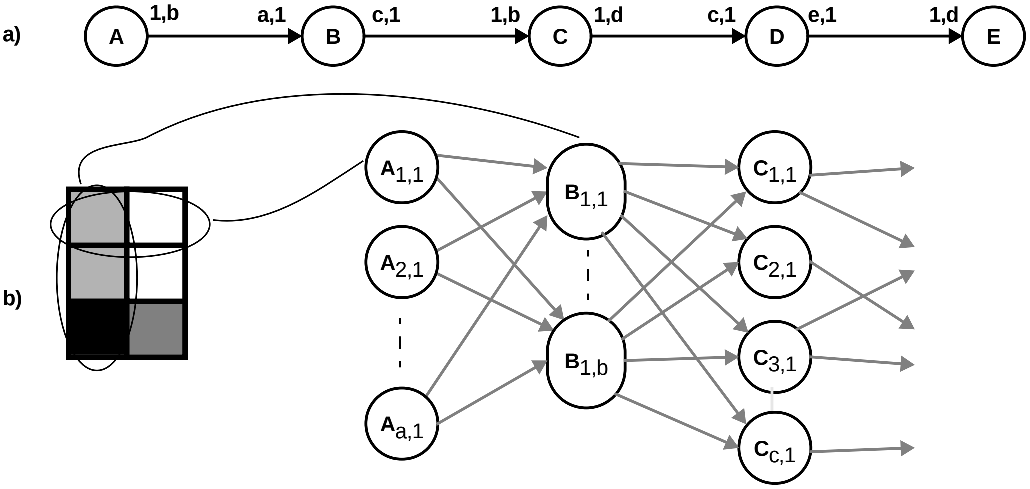


Figure 4-6 a) Multilayer perceptron expressed as an MDSDF graph.
b) The precedence graph

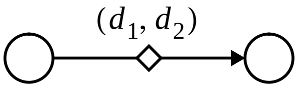


Figure 4-8 A delay in MD-SDF is multidimensional.

repeated firings of A and B, the delay on the feedback path will not be re-initialized. Hence, subsequent inner products will be incorrect.

A delay in MD-SDF in associated with a tuple as shown in figure 4-8. It can be interpreted as specifying boundary conditions on the index space. Thus, for 2D-SDF, as shown in the figure, it specifies the number of initial rows and columns. It can also be interpreted as specifying the direction in the index space of a dependence between two single assignment variables, much as done in reduced dependence graphs [Kung88].

Using MD-SDF delays, the repeated inner product can be specified as shown in figure 4-9. The only significant difference between this and figure 4-8 is the multidimensional delay. Its effect is illustrated schematically in figure 4-9, where the index space for the output of the delay is shown. The shaded area is the initial condition specified by the delay.

4.1.4 Multirate Actors

Some key multirate actors used in MDSDF specifications are shown in Figure 4-10. These are:

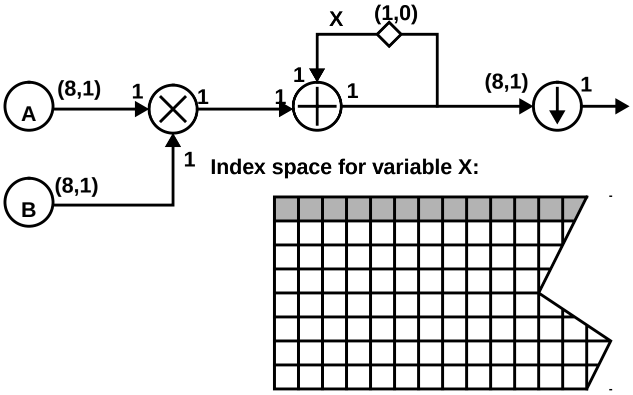


Figure 4-9 Repeated inner products in MD-SDF.

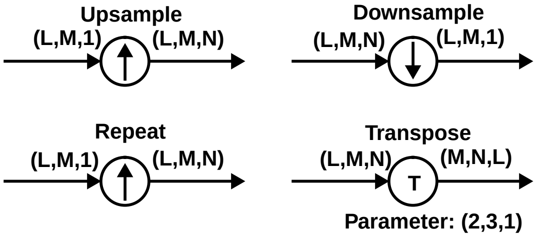


Figure 4-10 Some key MD-SDF actors that affect the flow of control.

- *Upsample:* In specified dimension(s), consumes 1 and produces N, inserting zero values.

- *Repeat*: In specified dimension(s), consumes 1 and produces N, repeating values.

- *Downsample*: In specified dimension(s), consumes N and produces 1, discarding samples.

- *Transpose:* Consumes and M-dimensional block of samples and outputs them with the dimensions rearranged.

These are identified in figure 4-10. Note that all of these actors simply control the way tokens are exchanged and need not involve any run-time operations. Of course, a compiler then needs to understand the semantics of these operators.

4.1.5 State

State in dataflow models of computation can be maintained by permitting actors to have self-loops. Consider the three actors with self loops shown in figure 4-11. Assume that dimension 1 indexes the row in the index space, and dimension 2 the column, as shown in figure 4-2. Then each firing of actor A requires state information from the previous row of

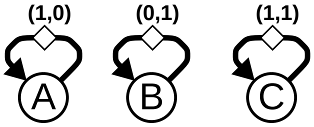


Figure 4-11 Three macro actors with state represented as a self-loop.

the index space for the state variable. Hence, each firing of A depends on the previous firing in the vertical direction, but there is no dependence in the horizontal direction. The first row in the state index space must be provided by the delay initial value specification. Actor B, in contrast, requires state information from the previous column in the index space. Hence there is horizontal, but not vertical dependence among firings. Actor C has both vertical and horizontal dependence, implying that both an initial row and an initial column must be specified. Note that this does imply that there is no parallelism, since computations along a diagonal wavefront can still proceed in parallel. Moreover, this property is easy to detect automatically in a compiler.

4.2 Scheduling

All of the scheduling techniques discussed in chapter 3 extend to the MDSDF model. We assume 2 dimensions for notational simplicity throughout this section unless otherwise stated. We use the notation $A_{[i,\ j]}$ to mean the $(i,\ j)$ th invocation of actor A in a complete periodic schedule. In a MDSDF schedule, a single appearance schedule like $(4, 2)A(6, 4)B$ means

```
for x = 0 to 3
          for y = 0 to 1
               fire $A_{[x, y]}$
end fory, forx

for x = 0 to 5
          for y = 0 to 3
               fire $B_{[x, y]}$
end fory, forx.
```

4.2.1 Self-loops

As mentioned before, delays in MDSDF represent boundary conditions. A delay (a, b) means that there are a initial rows and b initial columns. Suppose that A has a self loop. For the following discussion, we will assume that A will not write over any of the initial tokens on the edges. This is not a restriction since self-loops are usually used to specify states explicitly as inputs and outputs and actual implementations do not show these self-loops. In other words, we do not need to actually do things like buffer allocation for self-loops at the dataflow graph level; the codeblock inside the actor is responsible for managing its state. However, it is still conceptually important to consider the effect of this state since it imposes precedence constraints between different invocations of A. So we can ask whether a schedule like $(4, 2)A$ a valid schedule loop. Also, does the order of the loop nesting matter? We can state the following lemmas about self-loops:

Lemma 4-1: Suppose that an actor A has a self-loop as shown in figure 4-12. Actor A deadlocks iff $a_1 > d_1$ and $a_2 > d_2$ both hold.

Proof: If the inequalities both hold, then $A_{[0,\,0]}$ cannot fire since it requires a rectangle of data larger than that provided by the initial rows and columns intersected. The forward direction follows by looking at figure 4-12(b). If A deadlocks because $A_{[0,\,0]}$ cannot fire, then the inequalities must hold. If $A_{[0,\,0]}$ does fire, then it means that either $a_1 \le d_1$ or $a_2 \le d_2$. If $a_1 \le d_1$, then clearly $A_{[0,\,j]}$ can fire for any j since the initial rows provide the data for all these invocations. Then, $A_{[1,\,j]}$ can all fire since there are $a_1 + d_1$ rows of data now, and $2a_1 \le a_1 + d_1$. Continuing this argument, we can see that A can fire as many times as it wants. The reasoning if $a_2 \le d_2$ is symmetric; in this case, $A_{[i,\,0]}$ can all fire, and then $A_{[i,\,1]}$ can all fire and so on. So actor A deadlocks iff $A_{[0,\,0]}$ is not firable, and $A_{[0,\,0]}$ is not firable iff the condition in the lemma holds. **QED**

Corollary 4-1: In n dimensions, an actor A with a self-loop having $(d_1, \ldots, d_n)$ delays and producing and consuming hypercubes $(a_1, \ldots, a_n)$ deadlocks iff $a_i > d_i \quad \forall i$.

Let us now consider the precedence constraints imposed by the self-loop on the various invocations of A. Suppose that A fires (r_1, r_2) times. Then, the total array of data consumed is an array of size $(r_1 a_1, r_2 a_2)$. The same size array is written, but shifted to the

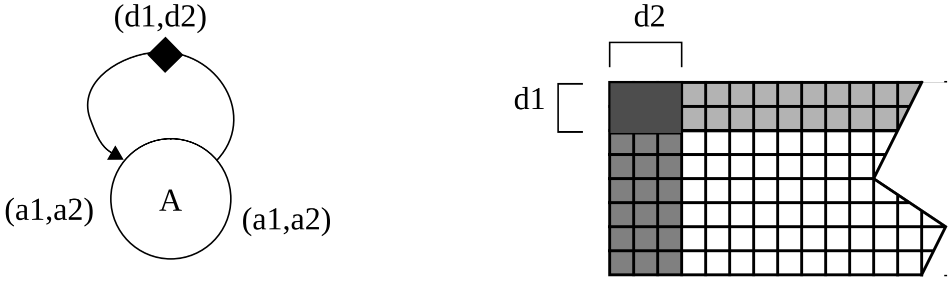


Figure 4-12 (a) An actor with a self loop. (b) Data space on the edge.

right and down of the origin by (d_1, d_2). In general, the rectangle of data read by a node is up and to the left of the rectangle of data written on this edge since we have assumed that the initial data is not being overwritten. Hence, an invocation $A_{[i,\,j]}$ can only depend on invocations $A_{[i',\,j']}$ where $i' \le i, j' \le j$. This motivates the following lemma:

Lemma 4-2: Suppose that actor A has a self-loop as in the previous lemma, and suppose that A does not deadlock. Then, the looped schedule $(r_1, r_2)A$ is valid, and the order of nesting the loops does not matter. That is,

```
for x = 0 : $r_1$-1
     for y = 0 : $r_2 - 1$
          fire $A_{[x,\,y]}$
end fory, forx.
```

and

```
for y = 0 : $r_2 - 1$
     for x = 0 : $r_1$-1
          fire $A_{[x,\,y]}$
end forx, fory.
```

give the same result.

Proof: We have to show that the ordering of the $A_{[x,\,y]}$ in the loop is a valid linearization of the partial order given by the precedence constraints of the self-loop. Suppose that in the first loop, the ordering is not a valid linearization. This means that there are indices (i_1, j_1) and (i_2, j_2) such that $A_{[i_2,\,j_2]}$ precedes $A_{[i_1,\,j_1]}$ in the partial order but $A_{[i_1,\,j_1]}$ is executed before $A_{[i_2,\,j_2]}$ in the loop. Then, by the order of the loop indices, it must be that $i_1 \le i_2$. But then $A_{[i_2,\,j_2]}$ cannot precede $A_{[i_1,\,j_1]}$ in the partial order since this violates the right and down precedence ordering. The other loop is also valid by a symmetric argument. **QED.**

The above two lemmas allow us to dispense with self-loops since as long as A does not deadlock due to a self-loop, the precedence constraints imposed by the self-loop can be met by the schedules we are interested in, namely, uniprocessor, single appearance schedules. Of-course, care must be exercised for multiprocessor scheduling but we are not concerned with multiprocessor scheduling in this chapter because standard multiprocessor scheduling techniques can be applied on the precedence graph of the MDSDF graph, and the precedence graph can be constructed in an analogous manner to the construction for SDF graphs.

4.2.2 GDPPO for MDSDF Graphs

The loop factoring process proceeds identically to SDF, with the factoring done in each dimension separately. For example, a looped schedule of the form $(4, 2)A(6, 4)B$ can be factored as $(2, 2)((2, 1)A(3, 2)B)$. GDPPO can be extended to single appearance schedules in order to yield buffer-optimal nested hierarchies. However, it is not enough to apply GDPPO on each dimension separately since the nesting hierarchies may be different for each dimension, meaning that we cannot combine the schedules into one schedule for the MDSDF graph.

Consider the simple graph shown in figure 4-13. If A fires $(r_{A,1}, r_{A,2})$ times, then the total buffer size on the edge is $r_{A,1}r_{A,2}O_{A,1}O_{A,2}$. Consider a chain-structured MDSDF graph with actors $A_1, \ldots, A_n$. Then, the relevant formulation for the dynamic programming algorithm (see Chapter 3) is given by

$$b[i, j] = min(\{(b[i, k] + b[k+1, j] + c_{i,j}[k]) | (i \le k < j)\}) \tag{4.4}$$

where

$$c_{i,j}[k] = \frac{r_{A_k,1}r_{A_k,2}O_{A_k,1}O_{A_k,2}}{gcd(\{r_{A_m,1} | (i \le m \le j)\})gcd(\{r_{A_m,2} | (i \le m \le j)\})} \tag{4.5}$$

Clearly, all of the extensions of the above basic dynamic programming algorithm presented in Chapter 3 can also be applied to MDSDF graphs, but we omit them here since the main challenge is the notation, more than anything else. In particular, given any single

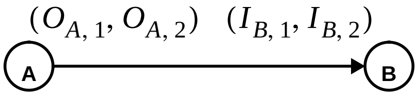


Figure 4-13 A simple MD-SDF graph.

appearance schedule for an MDSDF graph, a buffer-optimal loop hierarchy can be determined by the dynamic programming algorithm in polynomial time.

4.2.3 BMLB for MDSDF Graphs

The BMLB can be computed similarly, with a product of gcds in the denominator. First define

$$x(AB) = \frac{r_{A,1}}{gcd(r_{A,1}, r_{B,1})}O_{A,1}, y(AB) = \frac{r_{A,2}}{gcd(r_{A,2}, r_{B,2})}O_{A,2}$$

Then, the BMLB is defined as

$$BMLB(AB) = \begin{cases} (x(AB) + d_1)(y(AB) + d_2) & d_1 < x \vee d_2 < y \\ d_1 d_2 & d_1 \geq x \wedge d_2 \geq y \end{cases} \tag{4.6}$$

4.2.4 RPMC for MDSDF Graphs

The same cost function given in equation 4.5 can be used as the weighting function for the edges in a weighted graph; applying RPMC to this will then give us a heuristic procedure for generating a topological sort of an acyclic MDSDF graph, for which an optimal loop hierarchy can be computed using GDPPO.

4.2.5 APGAN for MDSDF Graphs

APGAN can be used on an acyclic MDSDF graph in the following way. Define the following two quantities:

$$\rho_1(\{A, B\}) = gcd(r_{A,1}, r_{B,1}) \text{ and } \rho_2(\{A, B\}) = gcd(r_{A,2}, r_{B,2}) \tag{4.7}$$

The clustering function is a tuple and is given by

$$\rho(\{A, B\}) \equiv (\rho_1(\{A, B\}), \rho_2(\{A, B\})) \tag{4.8}$$

At each step in the algorithm, we cluster the adjacent pair (A, B) that maximizes $\rho(\{A, B\})$ component-wise. This means that for any other adjacent clusterable pair $\{X, Y\}$, with $\rho'(\{X, Y\}) = (\rho'_1(\{X, Y\}), \rho'_2(\{X, Y\}))$ we should have $\rho_1 \geq \rho'_1, \rho_2 \geq \rho'_2$. If such a pair does not exist, we pick the adjacent clusterable pair $\{U, V\}$ that maximizes $\rho_1(\{U, V\})\rho_2(\{U, V\})$. Now, for an SDF graph, we define the **proper clustering condition** in the following way:

Definition 4-1: If G is a connected, consistent SDF graph, and $\{X, Y\}$ is an adjacent pair in G that does not introduce a cycle, we say that $\{X, Y\}$ satisfies the **proper clustering condition** in G if for each actor $Z \notin \{X, Y\}$ that is adjacent to a member of $\{X, Y\}$, we have that $\sigma(\{Z, P\})$ divides $\sigma(\{X, Y\})$, for each $P \in \{X, Y\}$ that Z is adjacent to. $\sigma(\{X, Y\})$ is the clustering function and is defined as $\sigma(\{X, Y\}) = gcd(\{\mathbf{q}_G(X), \mathbf{q}_G(Y)\})$.

Similarly, we define this condition for an MDSDF graph:

Definition 4-2: If G is a connected, consistent MDSDF graph, and $\{X, Y\}$ is an adjacent pair in G that does not introduce a cycle, we say that $\{X, Y\}$ satisfies the **proper clustering condition** in G if for each actor $Z \notin \{X, Y\}$ that is adjacent to a member of $\{X, Y\}$, we have that $\rho_i(\{Z, P\})$ divides $\rho_i(\{X, Y\})$, $i = 1, 2$, and for each $P \in \{X, Y\}$ that Z is adjacent to.

The proper clustering condition plays a key role in the proof that an APGAN instance returns a schedule for an acyclic SDF graph with buffer memory requirement equal to the BMLB if such a schedule exists [Bhat96a]. Essentially, it is shown in [Bhat96a] that clustering an adjacent pair that satisfies the proper clustering condition results in a graph where the BMLB on each edge is the same as the BMLB on the corresponding edge in the graph before clustering. If the adjacent pair that is clustered does not satisfy the proper clustering condition, then there is at least one edge in the clustered graph where the BMLB is greater than the BMLB on the corresponding edge in the graph before clustering. Hence, if the graph has a BMLB schedule, then the pair chosen for clustering at each step has to satisfy the proper clustering condition; otherwise, the BMLB on some edge will be increased and there is no hope of getting the optimal schedule. Fortunately, for a graph that has a BMLB schedule, the adjacent pair that maximizes $\sigma(\{A, B\})$ also satisfies the proper clustering condition.

Unfortunately, it is not enough to ensure that the clustered pair always satisfies the proper clustering condition. This is because even though the BMLB is preserved on each edge, the *existence* of a BMLB schedule may be cancelled in the clustered graph. The proper clustering condition is a local property, affecting only the edges adjacent to the actors in the clustered pair, while the existence of a BMLB schedule is a global property of the graph. Hence, the clustered pair has to not only satisfy the proper clustering condition, but it has to also ensure that the clustered graph has a BMLB schedule. Fortunately, clustering the adjacent pair that maximizes $\sigma(\{A, B\})$ also ensures that the existence of the BMLB schedule is not cancelled. Hence, this allows us to show that APGAN will return a BMLB schedule whenever one exists. All of the above results are non-trivial and are proven in detail for SDF graphs in [Bhat96a].

We can extend all of these results to MDSDF graphs using the definition of proper clustering given in definition 4-2. Using similar proof techniques, it can be shown that clustering an adjacent pair that satisfies the proper clustering condition in an MDSDF graph preserves all of the BMLBs, and clustering an adjacent pair that does not satisfy the proper clustering condition increases the BMLB on at least one edge. Hence, a graph that has a BMLB schedule will have the property that there is a clusterable adjacent pair at each step that maximizes the clustering function componentwise. If there is no such pair, then the graph does not have a BMLB schedule. It can be shown that the rest of the steps described above also hold for MDSDF graphs, and hence APGAN on an MDSDF graph will return a BMLB schedule whenever one exists.

Consider the example graph shown in figure 4-14. The repetitions vector is given by $r(A, B, C, D) = \{(2, 8), (6, 4), (4, 2), (1, 3)\}$. The clusterable pairs are $\{A, B\}$, $\{B, C\}$, and $\{C, D\}$. The clustering function values are $\rho(\{A, B\}) = (2, 4)$, $\rho(\{B, C\}) = (2, 2)$, and $\rho(\{C, D\}) = (1, 1)$. Hence, $\{A, B\}$ is the pair chosen for clustering since its clustering function has maximum component-wise value over the three clusterable pairs. Similarly, at the next step, there are two clusterable pairs, $\{W1, C\}$ and $\{C, D\}$, and the clustering function values are $\rho(\{W1, C\}) = (2, 2)$ and $\rho(\{C, D\}) = (1, 1)$. So $\{W1, C\}$ is clustered next, and the final schedule is

$(2, 2)(\ (1, 2)(\ (1, 2)A(3, 1)B\ )\ (2, 1)C\ )\ (1, 3)D$, and it can be verified that this is indeed a BMLB schedule.

The graph in figure 4-15 shows an example where there is no adjacent pair whose clustering function has the maximum-componentwise value. Hence, the graph does not have a BMLB schedule either, as is verified by looking at the two possible single appearance schedules. The repetitions vector is given by $\{(4, 5), (6, 15), (9, 3)\}$. The clustering function values for the two clusterable pairs are $\rho(\{A, B\}) = (2, 5)$ and $\rho(\{B, C\}) = (3, 3)$. The two possible single appearance schedules are $(2, 5)((2, 1)A(3, 3)B)\ (9, 3)C$ and $(4, 5)A\ (3, 3)((2, 5)B\ (3, 1)C)$ and neither of these is a BMLB schedule. The APGAN algorithm in this case will choose to cluster $\{A, B\}$ first because $2 \times 5 > 3 \times 3$; this results in the first of the two schedules given

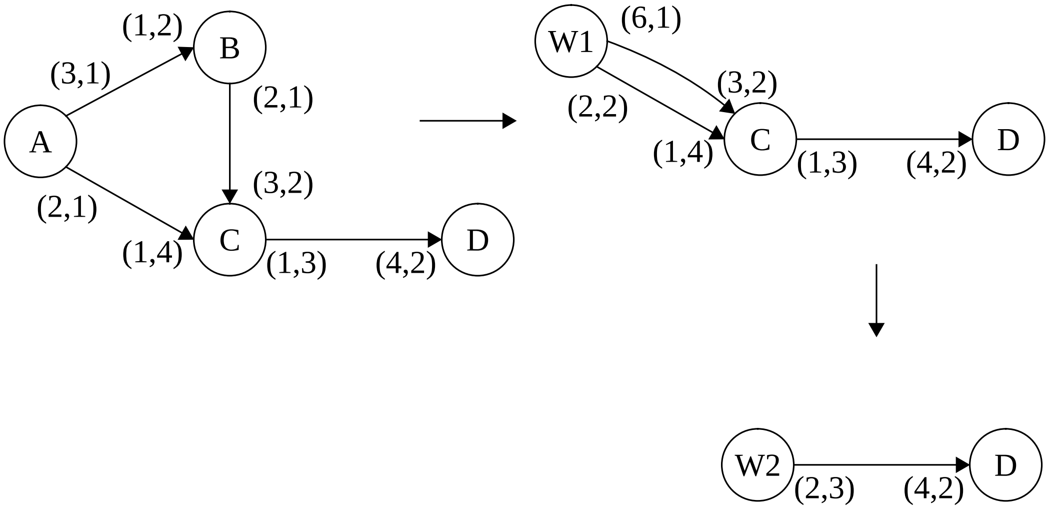


Figure 4-14 An MDSDF graph that has a BMLB schedule.

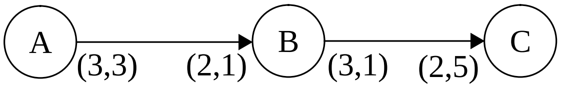


Figure 4-15 An example of a graph that does not have a BMLB schedule.

above. The first schedule has higher buffering requirements than the second; hence, APGAN is not optimal when the graph does not have a BMLB schedule.

4.3 Related Work

In [Watl95], Watlington and Bove discuss a stream-based computing paradigm for programming video processing applications. Rather than dealing with multidimensional dataspaces directly, as is done in this chapter, the authors sketch some ideas of how multidimensional arrays can be collapsed into one-dimensional streams using simple horizontal/vertical scanning techniques. They propose to exploit data parallelism from the one-dimensional stream model of the multidimensional system. While these ideas are interesting, it has been our experience that going to a one-dimensional model usually incurs extensive overhead since, often, extraneous actors that merely rearrange data have to be inserted into the system description to ensure correct operation. For example, in order to represent a 2-dimensional FFT operation using only 1 dimensional FFTs operating on one-dimensional streams, actors that transpose arrays have to be inserted into the algorithm [Chen94]. These actors are not only an unnecessary computational burden, but also obfuscate the semantics of the algorithm since they are an artifact of the model being used and not inherently part of the algorithm itself. Watlington and Bove also propose using dynamic (run-time) scheduling for their systems. However, our experience suggests that a great many signal processing systems can be modeled using SDF and MDSDF graphs since they seldom involve any data-dependent actors or control operations; these graphs can be scheduled statically. Systems that do involve control actors and data-dependent actors can be modeled by the more general boolean dataflow (BDF) model. Although the BDF model is Turing-complete, it is often possible to construct static schedules by graph clustering techniques [Buck93], and only when these techniques fail does one resort to dynamic scheduling. Even when dynamic scheduling becomes necessary, clustering techniques can still reduce the number of modules that have to be scheduled at run-time since significantly-sized subgraphs can often be scheduled statically even if the entire graph cannot be. Hence, a multidimensional extension of boolean dataflow would be desirable as this would allow us to have static schedules in many cases, with dynamic scheduling used only when the techniques of [Buck93] fail.

The Philips Video Signal Processor (VSP) is a commercially available processor designed for video processing applications. A single VSP chip contains 12 arithmetic/logic units, 4 memory elements, 6 on-chip buffers, and ports for 6 off-chip buffers. These are all interconnected through a full cross-point switch. Philips provides a programming environment for developing applications on the VSP. Programs are specified as signal flow graphs. Streams are one-dimensional, as in [Watl95]. Multirate operations are supported by associating a clock period with every operation. Because all of the streams are unidimensional, data-parallelism has to be exploited by inserting actors like multiplexors and de-multiplexors into the signal flow graphs.

4.3.1 AOL

There has been interesting work done at Thomson-CSF in developing the Array-Oriented language (AOL) [Deme94]. AOL is a specification formalism that tries to formalize the notion of array access patterns. The observation is that in many multidimensional signal processing algorithms, a chief problem is in specifying how multidimensional arrays are accessed. For example, in a 3 dimensional array of data, an FFT may be performed on blocks of input data taken in one dimension. Furthermore, these input blocks may overlap each other. It is observed that the computations themselves are often no different from similar computations in 1 dimensional signal processing. For example, a sum-of-products computations behaves the same way regardless of whether the dimension of samples it is computing on belongs to the time dimension (where it is called FIR filtering) or in spatial dimensions (in which case, the computation is called “beam-forming”).

It is assumed that computations are specified by a precedence graph in which actors represents computations (called “elementary transformations”), and the edges represent precedence constraints. The goal of the AOL formalism is to separate the computational aspects from the array-access aspects so that computations need not know about how the data they are operating on is arranged. This is done by associating a “template” for each input and each output of a computational node; the node simply assumes that the data it needs is in this template. These templates often are simply 1 dimensional arrays. Each edge also has the multidimensional array from which the elements of the template will be drawn from. It is the job of the user to specify how this multidimensional data object should actually feed the template (in the case of any input), and how data written into templates by the computation should be arranged in the multidimensional array (for an output). To this end, a graphical user interface (GUI) has been developed that allows the user to specify these access patterns graphically.

A few assumptions are made about the geometry of the templates in order to make the specification problem easier. It is observed that these assumptions are usually valid in practice. Access patterns are specified through two definitions: a "fitting" relationship, and a "paving" relationship.

The fitting relationship specifies how elements of the multidimensional array map into an input template (and conversely, how elements of an output template map into an output multidimensional array). This relationship is assumed to be affine, meaning that if two "examples" are given; that is, if two points and their mappings are specified, then the mapping for filling in the rest of the template can be inferred by extrapolation. Formally, let $x(0)$ and $x(1)$ be the first two elements of the input template. Then, the user specifies, using the GUI, the two points y and z in the multidimensional array that map to $x(0)$ and $x(1)$. By extrapolating the line from y to z in the multidimensional array, the remaining elements of the template can be filled in.

The paving relationship specifies how translations of the template in the input multidimensional array affect the translations of the output template in the output multidimensional array. This relationship is also assumed to be affine so that two examples, specified graphically in the GUI, are sufficient to specify the entire tiling of the input and output multidimensional arrays. The execution of the graph then proceeds by firing each computational node as many times as needed to completely fill its output multidimensional arrays.

The approach of AOL is different from dataflow-oriented models such as multidimensional synchronous dataflow (MDSDF) where no attempt has been made to address the access issue. Although it appears that all of the operations specifiable through

AOL are also specifiable in MDSDF, specifications in MDSDF might need extraneous nodes (such as repeaters, transposers, and commutators) to take care of the data manipulation. Although an intelligent compiler could in principle recognize the function of these extraneous data-manipulation actors and perform optimizations accordingly, so that little overhead is incurred for including them, their very presence in the graph might obscure the meaning of the algorithm, making it more difficult to modify it or maintain it. It is hoped that the AOL formalism will eliminate or at least ameliorate this problem. This should result in faster design times, visual programs that are understood more easily, which can then be maintained or modified or debugged much more easily.

The development of AOL has been in response to actual needs of signal processing systems designers at Thomson, and thus, should be of help to such designers. The formalism also opens up several research issues such as designing efficient schedulers, exploiting data parallelism effectively, implications for code generation for programmable video signal processing architectures, (none of which have been dealt with yet in [Deme94]), and better visualization techniques for these access patterns. Currently, in order to specify the fitting and paving relationships through the GUI, a multidimensional array has to be visualized using 2 dimensional projections. Since arrays with dimensions of upto 7 are not uncommon, the number of 2 dimensional projections can become too large; it would be cumbersome to have to deal with each of these sequentially for the designer (for example, a 7 dimensional array has 21 2-dimensional projections). Although it is not clear whether we can do any better than dealing with 2 dimensional projections, it certainly seems to be an interesting research issue.

5

Generalized Multidimensional Synchronous Dataflow

The multidimensional dataflow model presented in the previous chapter has been shown to be useful in a number of contexts including expressing multidimensional signal processing programs, specifying flexible data-exchange mechanisms, and scalable descriptions of computational modules. Perhaps the most compelling of these uses is the first one: for specifying multidimensional, multirate signal processing systems; this is because such systems, when specified in MDSDF have the same intuitive semantics that 1 dimensional systems have when expressed in SDF. However, the MDSDF model described so far is limited to modeling multidimensional systems sampled on the standard rectangular lattice. Since many multidimensional signals of practical interest are sampled on non-rectangular lattices [Mers83][Vaid90], for example, 2:1 interlaced video signals [Dubo85], and many multidimensional multirate systems use non-rectangular multirate operators like hexagonal decimators (see [Bamb90][Bosv92][Mand93] for examples), it is of interest to have an extension of the MDSDF model that allows signals on arbitrary sampling lattices to be represented, and that allows the use of non-rectangular downsamplers and upsamplers.

5.1 Multidimensional Signal Processing Fundamentals[1]

A multidimensional signal of dimension m, $x_a(t_1, \ldots, t_m)$, is a real-valued function of m real variables $t_1, \ldots, t_m$. This signal can be sampled to generate a discrete time signal. However sampling a multidimensional signal is fundamentally more complicated than sampling a unidimensional signal because of the many different ways the sampling geometry can be chosen. The straightforward extension of unidimensional sampling would result in the sequence $x(n_1, n_2) = x_a(n_1 T_1, n_2 T_2)$ where we have considered the $m = 2$ case for simplicity. Thus all values of t_1, t_2 that are **integer** multiples of the sampling periods T_1, T_2 are retained by the sampler. Figure 5-1 shows the samples that are retained for the case where $T_1 = 2, T_2 = 1$. As can be seen, the samples are arranged in a rectangular pattern, and thus this sampling scheme is known as **rectangular sampling**. A more general sampling scheme is to consider the sequence generated by

$$x(n_1, n_2) = x_a(a_{11} n_1 + a_{12} n_2, a_{21} n_1 + a_{22} n_2) \tag{5.1}$$

Notice that the sample locations retained are given by the equation

$$\hat{t} = \begin{bmatrix} t_1 \\ t_2 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} n_1 \\ n_2 \end{bmatrix} = V\hat{n} \tag{5.2}$$

The matrix V is called the sampling matrix. Every sample location $\hat{t}$ is of the form

$$\hat{t} = n_1 \hat{v}_1 + n_2 \hat{v}_2 \tag{5.3}$$

where

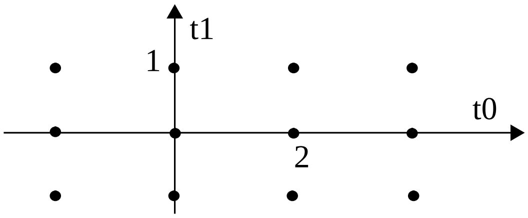


Figure 5-1 An illustration of rectangular sampling.

[1] The notation is taken from [Vaid93]

$$\hat{v}_1 = \begin{bmatrix} a_{11} \\ a_{21} \end{bmatrix}, \hat{v}_2 = \begin{bmatrix} a_{12} \\ a_{22} \end{bmatrix}, \text{ and}$$

n_1, n_2 are integers. That is, the sample locations are vectors $\hat{t}$ that are linear combinations of the columns of the sampling matrix V. Given the sampling matrix, the sample locations can be obtained graphically by first drawing the two vectors $\hat{v}_1$, $\hat{v}_2$ from the origin. Then draw two sets of equispaced parallel lines such that the two vectors form two sides of a parallelogram generated by these lines. The sample points are then located at the intersections of these lines. Figure 5-2(a) shows an example.

Note that the sampling matrix need not be an integer matrix but must be real and non-singular. Using the above terminology, rectangular sampling can be represented by a diagonal sampling matrix:

$$V = \begin{bmatrix} T_1 & 0 \\ 0 & T_2 \end{bmatrix}$$

In fact, rectangular sampling is defined to be a sampling scheme for which the sampling matrix is diagonal.

The set of all sample points $\hat{t} = V\hat{n}$, $\hat{n} \in \aleph^2$, where $\aleph$ is the set of natural numbers; that is, the set of vectors

$$\sum_{k=1}^{m} n_k \hat{v}_k,$$

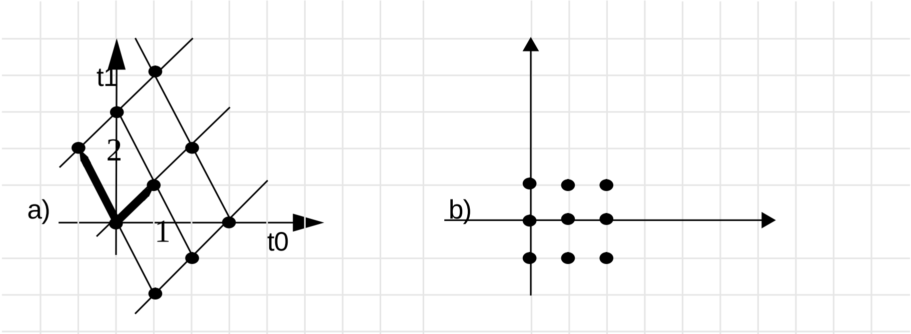


Figure 5-2 Sampling on a non-rectangular lattice. a) The samples on the lattice. b) The renumbered samples of the lattice.

is the set of all integer linear combinations of the columns $\hat{v}_1, \ldots, \hat{v}_m$ of the sampling matrix V. This set is called the ***lattice*** generated by V, and is denoted $LAT(V)$. The columns of the matrix V form the ***basis*** that generates the lattice $LAT(V)$. The basis for a lattice is not unique; for example,

$$LAT\left(\begin{bmatrix} 1 & -1 \\ 1 & 2 \end{bmatrix}\right) = LAT\left(\begin{bmatrix} 0 & -1 \\ 3 & 2 \end{bmatrix}\right)$$

The set of matrices that generate the same lattice can be characterized as follows.

Definition 5-1: A unimodular integer matrix E is a matrix with integer entries such that $det(E) = \pm 1$. Note that the inverse of E is also a unimodular integer matrix.

Lemma 5-1: Let V be a an $m \times m$ real non-singular matrix generating the lattice $LAT(V)$. Then, for any integer unimodular matrix E, $LAT(V) = LAT(VE)$. If $\widehat{V}$ is any other basis for $LAT(V)$, then there exists an integer unimodular matrix E such that $\widehat{V} = VE$.

Proof: This proof can be found in [Vaid93].

5.1.1 Numbering on a Lattice

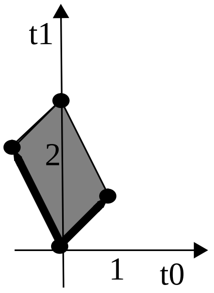
Suppose that $\hat{n}$ is a point on $LAT(V)$. Then there exists an integer vector $\hat{k}$ such that $\hat{n} = V\hat{k}$. The points $\hat{k}$ are called the **renumbered points** of $LAT(V)$. Figure 5-2(b) shows the renumbered samples for the samples on $LAT(V)$ shown in figure 5-2(a).

5.1.2 Sampling Density

The determinant of the sampling matrix plays a role in the sampling of an MD signal. The **sampling density** ρ, defined as the number of sample points per unit volume (or area in the 2-dimensional case), is given by $\rho = 1/|det(V)|$. The sampling density is an important concept that plays a role in the multidimensional sampling theorem. Intuitively, for an MD signal that is bandlimited in some fashion, we expect the sampling density to be above some minimum if we expect to retain all of the information in the signal. However, the sampling density depends on the sampling matrix, and hence the sampling geometry, and thus, by a judicious choice of sampling geometry, we can make sampling density as low as possible. An example of this is the sampling of an m-dimensional signal that is bandlimited to a hyperspherical region in the frequency plane. The savings in sampling density by using a particular non-diagonal sampling matrix (“hexagonal” sampling) over a diagonal sampling matrix (rectangular sampling) for sampling this hyperspherically bandlimited signal is a factor of 1.15 for $m = 2$. The savings factor goes up to a factor of 2 for 4 dimensional signals, and a factor of 16 to 8 dimensional signals [Dudg84]. Hence, there is a practical reason to favor non-rectangular sampling schemes over rectangular ones because of the potential savings in storage requirements and processing rates due to the lower sampling density that can result [Mers83]. Some applications that favor non-rectangular sampling schemes include 2:1 interlaced TV scanning [Dubo85], hierarchical video coding applications [Bosv92], and filterbanks for doing directional decomposition [Bamb90]. Filter design techniques for non-rectangular lattices have also been maturing as evidenced by many publications in the area [Karl90][Visc91][Ansa88][Bamb90][Vaid90][Vaid93].

5.1.3 The Fundamental Parallelepiped

Given a sampling matrix V, suppose that the column vectors are sketched from the origin, as shown in figure 5-3 for the example from figure 5-2. The completed parallelogram resulting from these vectors is called the fundamental parallelepiped of V, denoted $FPD(V)$. The points which fall inside $FPD(V)$ can be represented as the set $V\hat{x}$ where $\hat{x} = \begin{bmatrix} x_1 & x_2 \end{bmatrix}^T$, with $0 \le x_1, x_2 < 1$. The entire lattice $LAT(V)$ can be thought of as



$$V = \begin{bmatrix} 1 & -1 \\ 1 & 2 \end{bmatrix}$$

Figure 5-3 The fundamental parallelepiped for a matrix V

a tiling of the plane by copies of $FPD(V)$ shifted so that there is no overlap with other tiles, with the points of the lattice always falling on the corners of these tiles.

From geometry it is well known that the volume of $FPD(V)$ is given by $|det(V)|$. Since only one integer sample point falls inside $FPD(V)$, namely the origin, we can see why the sampling density is given by the inverse of the volume of $FPD(V)$.

Definition 5-2: Denote the set of integer points within $FPD(V)$ as the set $N(V)$. That is, $N(V)$ is the set of integer vectors of the form $V\hat{x}, \hat{x} \in [0, 1)^m$.

The following lemma characterizes the number of integer points that fall inside $FPD(V)$, or the size of the set $N(V)$. Even though the lemma is well known, we have not found a satisfactory proof for it anywhere; hence, we give a proof here. But before that, we state another lemma that is required for the proof of the lemma characterizing $|N(V)|$.

Lemma 5-2: For any integer matrix A, there exists an integer, unimodular matrix C such that CA is upper triangular.

Proof: See [Nemh88]; it also gives a polynomial time algorithm for finding C.

Lemma 5-3: Let V be an integer matrix. The number of elements in $N(V)$ is given by

$$|N(V)| = |det(V)| \tag{5.4}$$

Proof: Consider an upper triangular matrix M. In two dimensions,

$$M = \begin{bmatrix} e & f \\ 0 & g \end{bmatrix},$$

and this corresponds to a parallelogram with one of its sides on the x-axis. Since the side that is on the x axis has integer length e, the number of integer points on the x axis that lie inside $FPD(M)$ is just e. It is easy to see that there are f such rows of integer points in $FPD(M)$, and each of these corresponds to the vector $\begin{bmatrix} e & 0 \end{bmatrix}^T$ being shifted by an integer vector. Hence the number of integer points in each row is also e, making the total number of points in $FPD(M)$ equal to $ef = |det(M)|$. This argument can be extended to higher dimensions in the same way; the number of points will be the product of all the entries in M along the diagonal. Hence the lemma is true for integer, upper triangular matrices.

By lemma 5-2, we can always find an integer unimodular matrix C such that CV is upper triangular for any integer matrix V. Let x be an integer point in $N(CV)$. The point $C^{-1}x$ is an integer point and is in $N(V)$. Hence, for every integer point x in $N(CV)$, there is a unique integer point $C^{-1}x$ in $N(V)$. For every integer point x in $N(V)$, there is a unique integer point Cx in $N(CV)$. Hence, the size of $N(V)$ equals the size of $N(CV)$. Since CV is upper triangular, $|N(CV)| = |det(CV)| = |det(V)|$, where the last equality follows from the unimodularity of C. **QED**

5.1.4 Multidimensional Decimators

The two basic multirate operators for multidimensional systems are the decimator and expander. For an MD signal $x(\hat{n})$ on $LAT(V_I)$, the M-fold decimated version is given by $y(\hat{n}) = x(\hat{n}), \hat{n} \in LAT(V_I M)$ where M is an $m \times m$ non-singular integer matrix, called the ***decimation matrix***. Figure 5-4 shows two examples of decimation. The example on the left is for a diagonal matrix M; this is called rectangular decimation because $FPD(M)$ is a rectangle rather than a parallelepiped. In general, a rectangular decimator is one for which the decimation matrix is diagonal. The example on the right is for a non-diagonal M and is loosely termed "hexagonal" decimation. Note that $LAT(V_I) \supseteq LAT(V_I M)$

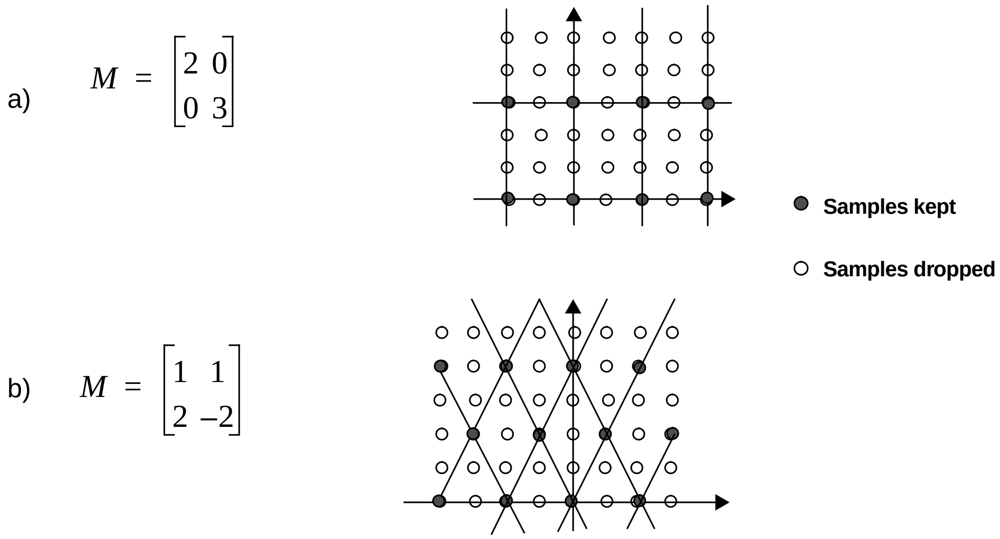

a) $M = \begin{bmatrix} 2 & 0 \\ 0 & 3 \end{bmatrix}$
● Samples kept
○ Samples dropped
b) $M = \begin{bmatrix} 1 & 1 \\ 2 & -2 \end{bmatrix}$

Figure 5-4 a) Rectangular decimation. b) Hexagonal decimation

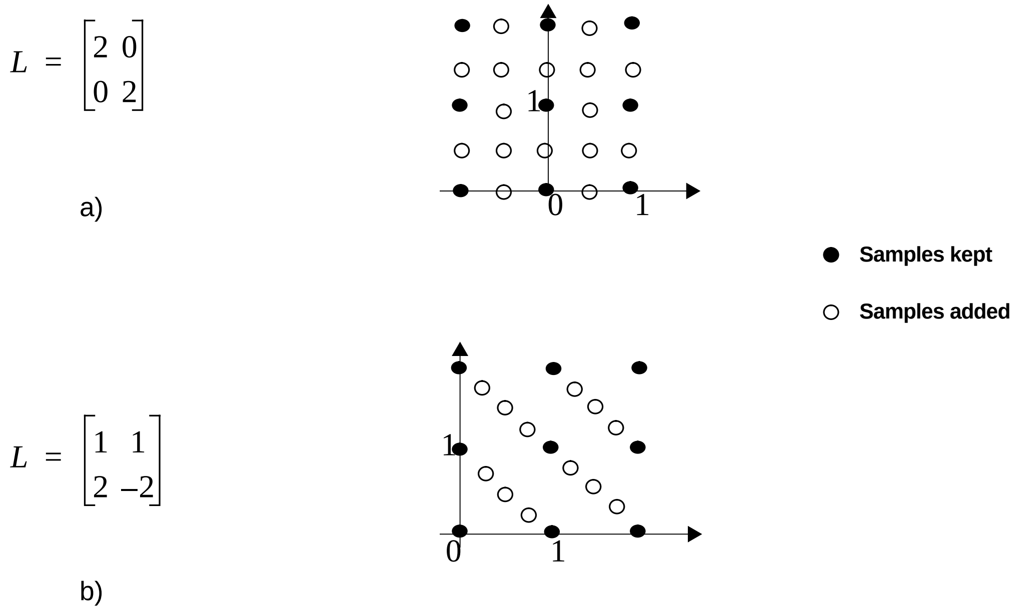
The ***decimation ratio*** for a decimator with decimation matrix M is defined to be the number of points thrown away for every point kept from the input and is given by $|N(M)| = |det(M)|$. The decimation ratio for the example on the left in figure 5-4 is 6, while it is 4 for the example on the right.

5.1.5 Multidimensional Expanders

In the multidimensional case, the “expanded” output $y(\hat{n})$ of an input signal $x(n)$ is given by:

$$y(n) = \begin{pmatrix} x(n) & n \in LAT(V_I) \\ 0 & \text{otherwise} \end{pmatrix} \forall n \in LAT(V_I L^{-1}) \tag{5.5}$$

where V_I is the input lattice to the expander. Note that $LAT(V_I) \subseteq LAT(V_I L^{-1})$. The expansion ratio, defined as the number of points added to the output lattice for each point in the input lattice, is given by $|det(L)|$. Figure 5-5 shows two examples of expansion. In the example on the left, the output lattice is also rectangular and is generated by



$L = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$
a)
1
0
1
● Samples kept
○ Samples added
$L = \begin{bmatrix} 1 & 1 \\ 2 & -2 \end{bmatrix}$
b)
1
0
1

Figure 5-5 a) Rectangular expansion. b) Non-rectangular expansion

$$\begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}.$$

The example on the right shows non-rectangular expansion, where the lattice is generated by

$$L^{-1} = \begin{bmatrix} 0.5 & 0.25 \\ 0.5 & -0.25 \end{bmatrix}$$

An equivalent way to view the above diagrams is to plot the renumbered samples. Notice that the samples from the input will now lie on $LAT(L)$ (figure 5-6).Some of the points have been labeled with letters to show where they would map to on the output signal.

5.2 Semantics of the Generalized Model

Consider the system depicted in figure 5-7, where a source actor produces an array of 6×6 samples each time it fires ((6,6) in MDSDF parlance). This actor is connected to the decimator with a non-diagonal decimation matrix. The circled samples indicate the samples that fall on the output lattice of the decimator; these are retained by the decimator. In order to represent these samples on the output of the decimator, we will think of the buffers on the arcs as containing the renumbered equivalent of the samples on a lattice. For a decimator, if we renumber the samples at the output according to $LAT(V_I M)$, then the samples get written to a parallelogram-shaped array rather than a rectangular array. To see what this parallelogram is, we introduce the concept of a "support matrix" that describes

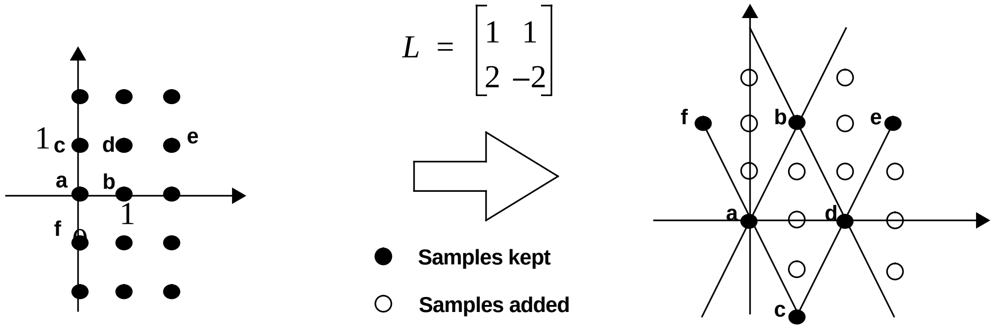


Figure 5-6 Renumbered samples from the output of the expander.

precisely the region of the rectangular lattice where samples have been produced. Figure 5-7 illustrates this for a decimation matrix, where the retained samples have been renumbered according to $LAT(M)$ and plotted on the right. The labels on the samples show the mapping. The renumbered samples can be viewed as the set of integer points lying inside the parallelogram that is shown in the figure. In other words, the ***support*** of the renumbered samples can be described as $FPD(Q)$ where

$$Q = \begin{bmatrix} 3 & 1.5 \\ 3 & 1.5 \end{bmatrix}$$

We will call Q the **support matrix** for the samples on the output arc. In the same way, we can describe the support of the samples on the input arc to the decimator as $FPD(P)$ where

$$P = \begin{bmatrix} 6 & 0 \\ 0 & 6 \end{bmatrix}$$

It turns out that $Q = M^{-1}P$.

Definition 5-3: The **containability condition:** let X be a set of integer points in $\Re^m$. We say that X satisfies the *containability condition* if there exists an $m \times m$ rational-valued matrix W such that $N(W) = X$.

Definition 5-4: We will assume that any source actor in the system produces data in the

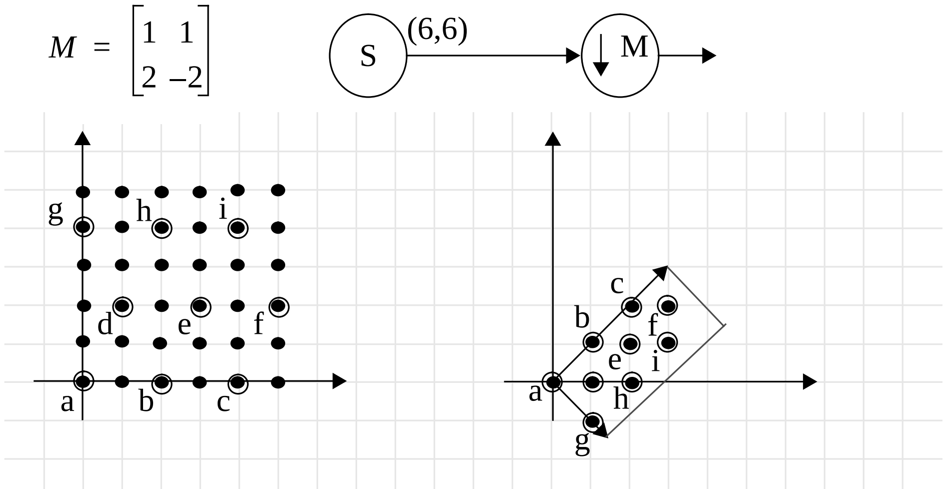


Figure 5-7 Output samples from the decimator renumbered to illustrate concept of support matrix. The circled samples on the left are labeled as shown.

following manner. A source S will produce a set of samples ζ on each firing such that each sample in ζ will lie on the lattice $LAT(V_S)$. Hence, the set $\bar{\zeta} = \{V_S^{-1}\hat{n}: \hat{n} \in \zeta\}$ is a set of integer points, consisting of the points of ζ renumbered by $LAT(V_S)$. We assume that the set $\bar{\zeta}$ satisfies the containability condition.

Given a decimator with decimation matrix M as shown in figure 5-8, we make the following definitions and statements. Denoting the input arc to the decimator as e and the output arc as f, V_e, and V_f are the bases for the input and output lattice respectively. W_e, and W_f are the support matrices for the input and output arcs respectively, in the sense that samples, numbered according to the respective lattices, are the integer points of fundamental parallelepipeds of the respective support matrices. Similarly, we can also define these quantities for the expander depicted in figure 5-8. With this notation, we can state the following theorem:

Theorem 5-1: The relationships between the input and output lattices, and the input and output support matrices for the decimator and expander depicted in figure 5-8 are:

Decimator $V_f = V_e M, \quad W_f = M^{-1} W_e$.

Expander $V_f = V_e L^{-1}, W_f = L W_e$.

Proof: The relationships between the input and output lattices follow from the definition of the expander and decimator. Consider a point n on the decimator's input lattice. There exists an integer vector k such that $n = V_e k$. If $M^{-1}k$ is an integer vector, then this point will be kept by the decimator since it will fall on the output lattice; i.e, $n = V_e M k'$ where $k' = M^{-1}k$. This point n is renumbered as $k' = M^{-1}V_e^{-1}n = M^{-1}k$ by the output lattice. Since k was the renumbered point corresponding to n on the input lattice, and hence in $N(W_e)$, every point k in $N(W_e)$ that is kept by the decimator is mapped to $M^{-1}k$ by the output lattice. Now, $k \in N(W_e) \Rightarrow \exists z \in [0, 1)^2$ s.t. $k = W_e z$. So $M^{-1}k \in N(M^{-1}W_e)$ because $M^{-1}k = M^{-1}W_e z$. Conversely, let j be any point in $N(M^{-1}W_e)$. Then,

V_e, W_e
M
V_f, W_f
e
f
V_e, W_e
L
V_f, W_f
e
f

Figure 5-8 Generalized expander and decimator with arbitrary input lattices and support matrices.

$\exists z \in [0, 1)^2$ s.t. $j = M^{-1}W_e z$. Since $W_e z = Mj$, we have that $Mj \in N(W_e)$. Also, the corresponding point to this on the input lattice is $V_e Mj$ implying that the point is retained by the decimator. Hence, $W_f = M^{-1}W_e$. The derivation for the expander is identical, only with different expressions. **QED**

Corollary 5-1: In an acyclic network of actors, where the only actors that are allowed to change the sampling lattice are the decimator and expander in the manner given by theorem 5-1, and where all source actors produce data according to definition 5-4, the set of samples on every arc, renumbered according to the sampling lattice on that arc, satisfies the containability condition.

Proof: Immediate from theorem.

In the following, we develop the semantics of a model that can express these non-rectangular systems by going through a detailed example. In general, our model for the production and consumption of tokens will be the following: an expander produces $FPD(L)$ samples on each firing where L is the upsampling matrix. The decimator consumes a “rectangle” of samples where the “rectangle” has to be suitably defined by looking at the actor that produces the tokens that the decimator consumes.

Definition 5-5: An **integer** (a, b) **rectangle** is defined to be the set of integer points in $[0, a) \times [0, b)$, where a, b are arbitrary real numbers.

Definition 1: Let X be a set of points in $\Re^2$, and x, y be two positive integers such that $xy = |X|$. X is said to be organized as a **generalized** (x, y) **rectangle** of points, or just a generalized (x, y) rectangle, by associating a **rectangularizing** function with X that maps the points of X to an integer (x, y) rectangle.

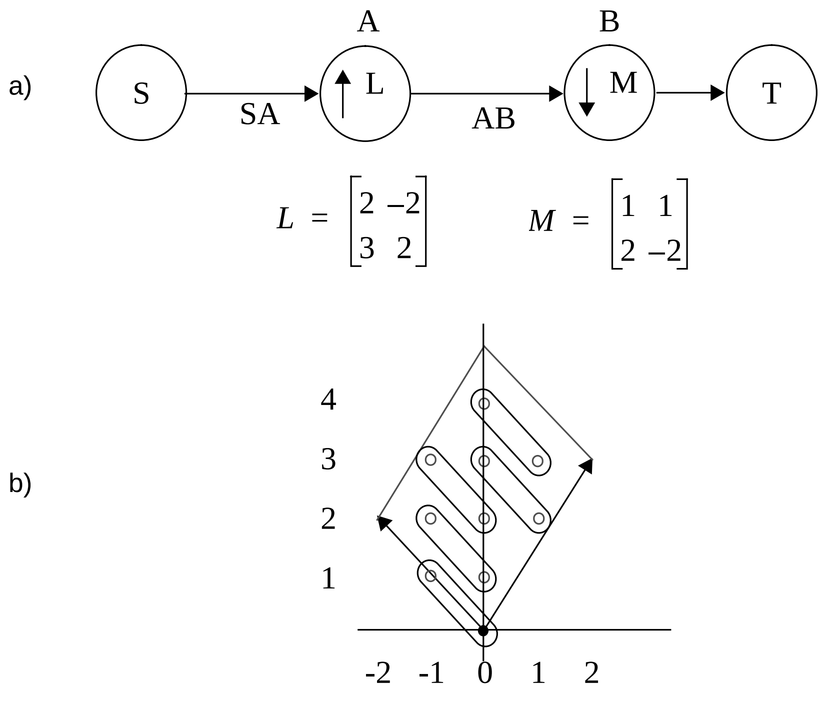
Example 5-1: Consider the system below, where a decimator follows an expander (figure 5-9(a))

We start by specifying the lattice and support matrix for the arc SA. Let

$$V_{SA} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \text{ and } W_{SA} = \begin{bmatrix} 3 & 0 \\ 0 & 3 \end{bmatrix}.$$

So the source produces (3,3) in MDSDF parlance. For the system above, we can compute the lattice and support matrices for all other arcs given these. We will need to specify the scanning order for each arc as well that tells the node the order in which samples should be consumed. Assume for the moment that the expander will consume the samples on arc SA in some natural order; for example, scanning by rows. We need to specify what the expander produces on each firing. The natural way to specify this is that the expander produces $FPD(L)$ samples on each firing; these samples are organized as a generalized (L_1, L_2) rectangle. This allows us to say that the expander produces (L_1, L_2) samples per firing; this is understood to be the set $FPD(L)$ of points organized as a generalized (L_1, L_2) rectangle.

Suppose we choose the factorization 5×2 for $|det(L)|$. Consider figure 5-9(b) where the samples in $FPD(L)$ are shown. One way to map the samples into an integer $(5, 2)$ rectangle is as shown by the groupings. Notice that the horizontal direction for



$$L = \begin{bmatrix} 2 & -2 \\ 3 & 2 \end{bmatrix} \qquad M = \begin{bmatrix} 1 & 1 \\ 2 & -2 \end{bmatrix}$$

Figure 5-9 An example to illustrate balance equations and the need for some additional constraints. a) The system. b) Ordering of data into a 5x2 rectangle inside FPD(L).

$FPD(L)$ is the direction of the vector $\begin{bmatrix}2 & 3\end{bmatrix}^T$ and the vertical direction is the direction of the vector $\begin{bmatrix}-2 & 2\end{bmatrix}^T$. We can number the samples as follows:

Table 5-1: Ordering the samples produced by the expander

Original sample	(0,0)	(0,1)	(0,2)	(1,2)	(1,3)	(-1,1)	(-1,2)	(-1,3)	(0,3)	(0,4)
Renumbered sample	(0,0)	(1,0)	(2,0)	(3,0)	(4,0)	(0,1)	(1,1)	(2,1)	(3,1)	(4,1)

Hence, $FPD(L)$ is a generalized $(5, 2)$ rectangle if we associate the function given in the table above with it as the rectangularizing function. Given a factoring of the determinant of L, the function given above can be computed easily; for example, by ordering the samples according to their Euclidean distance from the two vectors that correspond to the horizontal and vertical directions. The scanning order for the expander across invocations is determined by the numbering of the input sample on the output lattice. For example, the sample at (1,0) that the source produces maps to location (2,3) in the re-numbered lattice at the expanders output. Hence, consuming samples in the [1 0] direction on arc SA results in 5x2 samples (i.e, $FPD(L)$ samples but ordered according to the table) being produced along the vector [2 3] on the output. Similarly, the sample (0,1) produced by the source corresponds to (-2,2) on the output. A global ordering on the samples is imposed by renumbering the sample at (2,3) as (5,0) since the first $FPD(L)$ of samples produced ended with sample (4,1). With this global ordering, it becomes clear what the semantics for the decimator should be. Again, choose a factorization of $|det(M)|$, and consume a “rectangle” of those samples, where the “rectangle” is deduced from the global ordering imposed above. For example, if we choose 2x2 as the factorization, then the (0,0) invocation of the decimator consumes the (original) samples at (0,0), (-1,1), (0,1), and (-1,2). The (0,2)th invocation of the decimator would consume the (original) samples at (1,3), (0,4), (2,3) and (1,4). The decimator would have to determine which of these samples falls on its lattice; this can be done easily.

We have already mentioned the manner in which the source produces data. We add that the subsequent firings of the source are always along the directions established by the vectors in the support matrix on the output arc of the source.

Now we can write down a set of “balance” equations using the “rectangles” that we have defined. Denote the repetitions of a node X in the “horizontal” direction by $r_{X,1}$ and the “vertical” direction as $r_{X,2}$. These directions are dependent on the geometries that have been defined on the various arcs. Thus, for example, the directions are different on the input arc to the expander from the directions on the output arc. We have

$$\begin{aligned} 3r_{S,1} &= 1r_{A,1} \\ 3r_{S,2} &= 1r_{A,2} \\ 5r_{A,1} &= 2r_{B,1} \\ 2r_{A,2} &= 2r_{B,2} \\ r_{B,1} &= r_{T,1} \\ r_{B,2} &= r_{T,2} \end{aligned} \tag{5.6}$$

where we have assumed that the sink actor T consumes (1,1) for simplicity. We have also made the assumption that the decimator produces exactly (1,1) every time it fires. This assumption is usually invalid but the calculations done below are still valid as will be discussed later. These equations can be solved to yield

$$\begin{aligned} r_{S,1} &= 2, r_{S,2} = 1 \\ r_{A,1} &= 6, r_{A,2} = 3 \\ r_{B,1} &= 15, r_{B,2} = 3 \\ r_{T,1} &= 15, r_{T,2} = 3 \end{aligned} \tag{5.7}$$

Figure 5-10 shows the data space on arc AB with this solution to the balance equations. As we can see, the assumption that the decimator produces (1,1) on each invocation is not valid; sometimes it is producing no samples at all and sometimes 2 samples or 1 sample. Hence, we have to see if the total number of samples retained by the decimator is equal to the total number of samples it consumes divided by the decimation ratio.

In order to compute the number of samples output by the decimator, we have to compute the support matrices for the various arcs assuming that the source is invoked (2,1) times (so that we have the total number of samples being exchanged in one schedule period). We can do this symbolically using $r_{S,1}, r_{S,2}$ and substitute the values later. We get

$$W_{SA} = \begin{bmatrix} 3 & 0 \\ 0 & 3 \end{bmatrix} \begin{bmatrix} r_{S,1} & 0 \\ 0 & r_{S,2} \end{bmatrix} = \begin{bmatrix} 3r_{S,1} & 0 \\ 0 & 3r_{S,2} \end{bmatrix},$$

$$W_{AB} = LW_{SA} = \begin{bmatrix} 2 & -2 \\ 3 & 2 \end{bmatrix} \begin{bmatrix} 3r_{S,1} & 0 \\ 0 & 3r_{S,2} \end{bmatrix} = \begin{bmatrix} 6r_{S,1} & -6r_{S,2} \\ 9r_{S,1} & 6r_{S,2} \end{bmatrix}, \text{ and}$$

$$W_{BT} = M^{-1}W_{AB} = \frac{1}{4}\begin{bmatrix} 2 & 1 \\ 2 & -1 \end{bmatrix} \begin{bmatrix} 6r_{S,1} & -6r_{S,2} \\ 9r_{S,1} & 6r_{S,2} \end{bmatrix} = \frac{1}{4}\begin{bmatrix} 21r_{S,1} & -6r_{S,2} \\ 3r_{S,1} & -18r_{S,2} \end{bmatrix} \tag{5.8}$$

Recall that the samples that the decimator produces are the integer points in $FPD(W_{BT})$. Hence, we want to know if

$$\left|N(W_{BT})\right| = \left|N(W_{AB})\right| / \left|M\right| \tag{5.9}$$

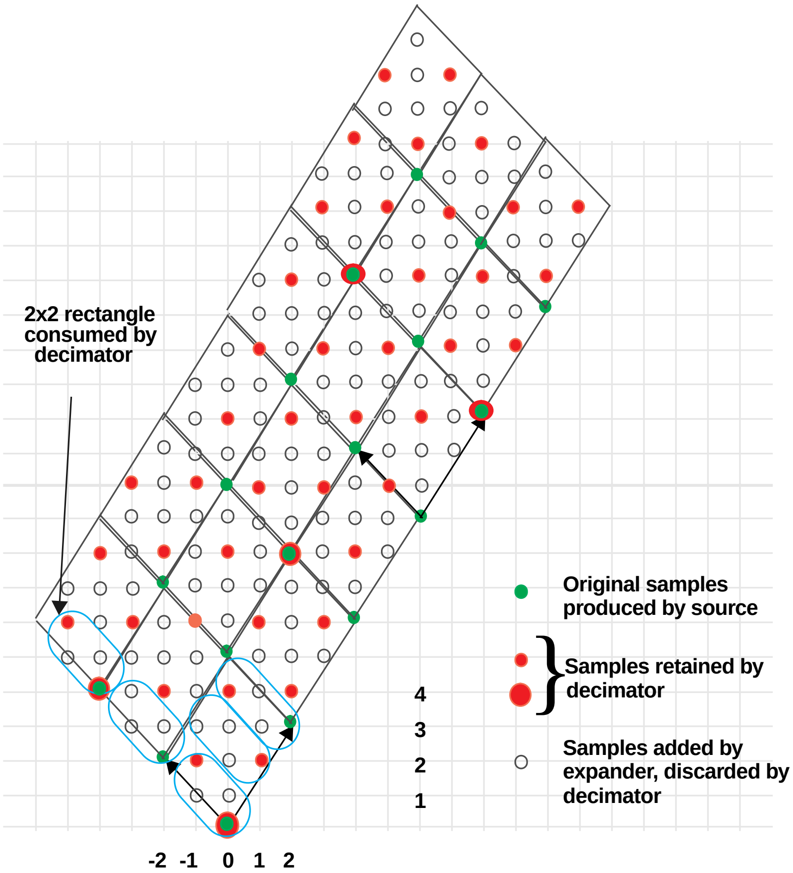

2x2 rectangle consumed by decimator
Original samples produced by source
Samples retained by decimator
Samples added by expander, discarded by decimator
4
3
2
1
-2
-1
0
1
2

Figure 5-10 Total amount of data produced by the source in one iteration of the periodic schedule determined by the balance equations in equation 5.7.

is satisfied by our solution to the balance equations. By lemma 5-3, the size of the set $N(A)$ for an integer matrix A is given by $|det(A)|$. Since W_{AB} is an integer matrix for any value of $r_{S,1}, r_{s,2}$, we have $|N(W_{AB})| = |det(W_{AB})| = 90r_{S,1}r_{S,2}$. The right hand side of equation 5.9 becomes $(90r_{S,1}r_{S,2})/4 = (45r_{S,1}r_{S,2})/2$. Hence, our first requirement is that $r_{S,1}r_{S,2} = 2k \quad k = 0, 1, 2, \ldots$. The balance equations gave us $r_{S,1} = 2, r_{S,2} = 1$; this satisfies the requirement. With these values, we get

$$W_{BT} = \begin{bmatrix} 21/2 & -3/2 \\ 3/2 & -9/2 \end{bmatrix}.$$

Since this matrix is not integer-valued, lemma 5-3 cannot be invoked to calculate the number of integer points in $FPD(W_{BT})$. For non-integer matrices, the only way to compute $|N(W_{BT})|$ appears to be by brute force: by drawing this out on graph paper, it can be determined that there are 47 points inside. Hence, equation 5.9 is not satisfied! One way to satisfy equation 5.9 is to force W_{BT} to be an integer matrix. This implies that $r_{S,1} = 4k, k = 1, 2, \ldots$ and $r_{S,2} = 2k, k = 1, 2, \ldots$. The smallest values that make W_{BT} integer valued are $r_{S,1} = 4, r_{S,2} = 2$. From this, the repetitions of the other nodes are also multiplied by 2, thus increasing the *blocking factor* to 2, where the definition of the blocking factor is as in the MDSDF case. Note that the solution to the balance equations by themselves are not “wrong”; it is just that for non-rectangular systems equation 5.9 gives a new constraint that must also be satisfied.

We can formalize the ideas developed in the example above in the following.

Lemma 5-4: The support matrices in the network can each be written down as functions of the repetitions variables of one particular source actor in the network.

Proof: Immediate from the fact that all of the repetitions variables are related to each other via the balance equations.

Lemma 5-5: In a multidimensional system, the j^{th} column of the support matrix on any arc can be expressed as a matrix that has entries of the form $a_{ij}r_{S,j}$, where $r_{S,j}$ is the repetitions variable in the j^{th} dimension of some particular source actor S in the network, and a_{ij} are rationals.

Proof: Without loss of generality, assume that there are 2 dimensions. Let the support matrix on the output arc of source S for one firing be given by

$$W_S = \begin{bmatrix} p & q \\ r & s \end{bmatrix}.$$

For $r_{S,1}, r_{S,2}$ firings in the “horizontal” and “vertical” directions (these are the directions of the columns of W_S), the support matrix becomes

$$W'_S = \begin{bmatrix} p & q \\ r & s \end{bmatrix} \begin{bmatrix} r_{S,1} & 0 \\ 0 & r_{S,2} \end{bmatrix} = \begin{bmatrix} pr_{S,1} & qr_{S,2} \\ rr_{S,1} & sr_{S,2} \end{bmatrix}$$

(in multiple dimensions, the right multiplicand would be a diagonal matrix with $r_{S,j}$ in j^{th} row).

Now consider an arbitrary arc (u, v) in the graph. Since the graph is connected, there is at least one undirected path P from source S to node u. Since the only actors that change the sampling lattice (and thus the support matrix) are the decimator and expander, all of the transformations that occur to the support matrix W_S along P are left multiplications by some rational valued matrix. Hence, the support matrix on arc e, W_e, can be expressed as $W_e = AW_S$, where A is some rational valued matrix. The claim of the lemma follows from this.

Theorem 5-2: In an acyclic network of actors, where the only actors that are allowed to change the sampling lattice are the decimator and expander in the manner given by theorem 5-1, and where all source actors produce data according to definition 5-4, whenever the balance equations for the network have a solution, there exists a blocking factor vector J such that increasing the repetitions of each node in each dimension by the corresponding factor in J will result in the support matrices being integer valued for all arcs in the network.

Proof: By lemma 5-5, a term in an entry in the j^{th} column of the support matrix on any arc is always a product of a rational number and repetitions variable $r_{S,j}$ of source S. We force this term to be integer valued by dictating that each repetitions variable $r_{S,j}$ be the lcm of the values needed to force each entry in the j^{th} column to be an integer. Such a value can be computed for each support matrix in the network. The lcm of all these values and the balance equations solution for the source would then give a repetitions vector for the source that makes all of the support matrices in the network integer valued and solves the balance equations. **QED**

5.2.2 The Rectangular Case

Here we show that the constraint of the type in equation 5.9 is always satisfied by the solution to the balance equations when all of the lattices and matrices are diagonal. Since we are only interested in these additional constraints for arcs between an expander and decimator, consider the system in figure 5.12. The balance equations for arc AB are

$$\begin{aligned} L_1 r_{A,1} &= M_1 r_{B,1} \\ L_2 r_{A,2} &= M_2 r_{B,2} \end{aligned} \qquad (5.10)$$

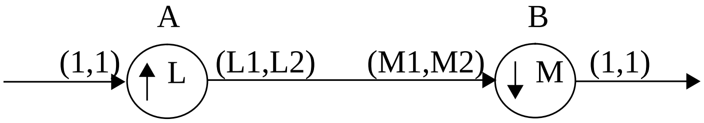
The support matrix for arc AB is given by

$$W_{AB} = \begin{bmatrix} L_1 r_{A,1} & 0 \\ 0 & L_2 r_{A,2} \end{bmatrix}$$

since the input lattice and support matrix on the expanders input are both diagonal. The support matrix for arc BT is given by

$$W_{BT} = M^{-1} W_{AB} = \begin{bmatrix} M_1^{-1} & 0 \\ 0 & M_2^{-1} \end{bmatrix} \begin{bmatrix} L_1 r_{A,1} & 0 \\ 0 & L_2 r_{A,2} \end{bmatrix} = \begin{bmatrix} M_1^{-1} L_1 r_{A,1} & 0 \\ 0 & M_2^{-1} L_2 r_{A,2} \end{bmatrix}$$

We have $|N(W_{AB})| = L_1 L_2 r_{A,1} r_{A,2}$. A solution to the balance equations in equation 5.10 implies that the matrix W_{BT} is an integer matrix; hence,



$$L = \begin{bmatrix} L_1 & 0 \\ 0 & L_2 \end{bmatrix} \qquad M = \begin{bmatrix} M_1 & 0 \\ 0 & M_2 \end{bmatrix}$$

Figure 5-11 For a rectangular system, the constraint of equation 5.9 is always met.

$|N(W_{BT})| = |det(W_{BT})| = M_1^{-1}L_1r_{A,1}M_2^{-1}L_2r_{A,2}$ and we see that equation 5.9 is satisfied since $|det(M)| = M_1M_2$. So we see that the rectangular MDSDF case is a special case of the more general set of constraints needed for non-rectangular systems.

The fact that the decimator produces a varying number of samples per invocation might suggest that it falls nicely into the class of cyclostatic actors. However, there are a couple of differences. In the CSDF model of [Lauw94], the number of cyclostatic phases are assumed to be known beforehand, and is only a function of the parameters of the actor, like the decimation factor. In our model for the decimator, the number of phases is not just a function of the decimation matrix; it is also a function of the sampling lattice on the input to the decimator (which in turn depends on the actor that is feeding the arc), and the factorization choice that is made by the scheduler. Secondly, in CSDF, SDF actors are represented as cyclostatic by decomposing their input/output behavior over one invocation. For example, a CSDF decimator behaves exactly like the SDF decimator except that the CSDF decimator does not need all M data inputs to be present before it fires; instead, it has a 4-phase firing pattern. In each phase, it will consume 1 token, but will produce one token only in the first phase, and produce 0 tokens in the other phases. In our case, the cyclostatic behavior of the decimator is arising *across* invocations rather than within an invocation. It is as if the CSDF decimator with decimation factor 4 were to consume {4,4,4,4,4,4} and produce {2,0,1,1,0,2} instead of consuming {1,1,1,1} and producing {1,0,0,0}.

One way to avoid dealing with constraints of the type in equation 5.9 would be to choose a factorization of $|det(M)|$ that ensured that the decimator produced one sample on each invocation. For example, if we were to choose the factorization 1x4 for the example above, the solution to the balance equations would automatically satisfy equation 5.9. As we show later, we can find factorizations where the decimator produces one sample on every invocation in certain situations but generalizing this result appears to be a difficult problem since there does not seem to be an analytical way of writing down the renumbering transformation that was shown in table 1.

5.2.3 Implications of the Above Example for Streams

In SDF, there is only one dimension, and the stream is in that direction. Hence, whenever the repetitions of a node is greater than unity, then the data processed by that node corresponds to data along the stream. In MDSDF, only one of the directions is the stream. Hence, if the repetitions of a node, especially a source node, is greater than unity for the non-stream directions, the physical meaning of invocations in those directions becomes unclear. For example, consider a 3-dimensional MDSDF model for representing a progressively scanned video system. Of these 3 dimensions, 2 of the dimensions correspond to the height and width of the image, and the third dimension is time. Hence, a source actor that produces the video signal might produce something like (512,512,1) meaning 1 512x512 image per invocation. If the balance equations dictated that this source should fire (2,2,3) times, for example, then it is not clear what the 2 repetitions each in the height and width directions signify since they certainly do not result in data from the next iteration being processed, where an iteration corresponds to the processing of an image at the next sampling instant. Only the repetitions of 3 along the time dimension makes physical sense. Hence, there is potentially room for great inefficiency if the user of the system has not made sure that the rates in the graph match up appropriately so that we do not actually end up generating images of size 1024x1024 when the actual image size is 512x512. In rectangular MDSDF, it might be reasonable to assume that the user is capable of setting the MDSDF parameters such that they do not result in absurd repetitions being generated in the non-stream directions since this can usually be done by inspection. However for non-rectangular systems, we would like to have more formal techniques for keeping the repetitions matrix in check since it is much less obvious how to do this by inspection. The number of variables are also greater for non-rectangular systems since different factorizations for the decimation or expansion matrices give different solutions for the balance equations.

To explore the different factoring choices, suppose we use 1×4 for the decimator instead of 2×2. The solution to the balance equations become

$$
\begin{aligned}
r_{S,1} &= 1, r_{S,2} = 2\\
r_{A,1} &= 3, r_{A,2} = 6\\
r_{B,1} &= 15, r_{B,2} = 3\\
r_{T,1} &= 15, r_{T,2} = 3
\end{aligned}
\tag{5.11}
$$

From equation 5.8, W_{BT} is given by

$$W_{BT} = \begin{bmatrix} 21/4 & -3 \\ 3/4 & -9 \end{bmatrix}$$

and it can be determined that $|N(W_{BT})| = 45$, as required. So in this case, we do not need to increase the blocking factor to make W_{BT} an integer matrix, and this is because the decimator is producing 1 token on every firing as shown in figure 5-12.

However, if the stream in the above direction were in the horizontal direction (from the point of view of the source), then the solution given by the balance equations (eq. 5.11)

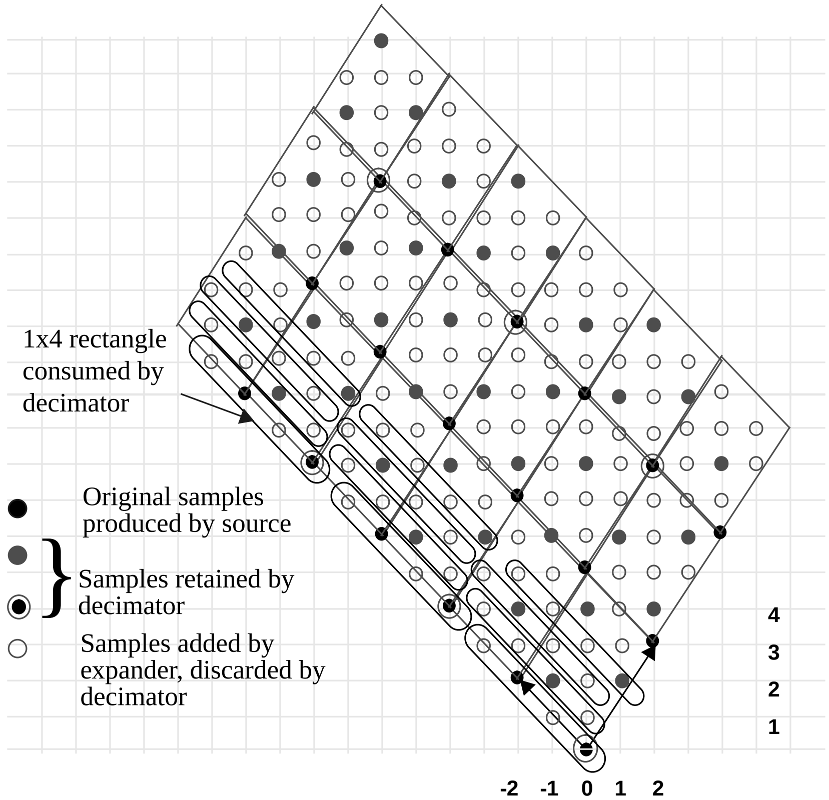

1x4 rectangle consumed by decimator
● Original samples produced by source
● Samples retained by decimator
◉
○ Samples added by expander, discarded by decimator
4
3
2
1
-2
-1
0
1
2

Figure 5-12 Total amount of data produced by the source in one iteration of the periodic schedule determined by the balance equations in equation 5.11. The samples that are kept by the decimator are the lightly shaded samples.

may not be satisfactory for reasons already mentioned. For example, the source may be forced to produce only zeros for invocation (0,1). One way to incorporate such constraints into the balance equations computation is to specify the repetitions vector instead of the number produced or consumed. That is, for the source, we specify that $r_{S,2} = 1$ but leave the number it produces in the vertical direction unspecified. The balance equations will give us a set of acceptable solutions involving the number produced vertically; we can then pick the smallest such number that is greater than or equal to three. Denoting the number produced vertically by y_S, our balance equations become

$$\begin{aligned} 3r_{S,1} &= 1r_{A,1} \\ y_S 1 &= 1r_{A,2} \\ 5r_{A,1} &= 1r_{B,1} \\ 2r_{A,2} &= 4r_{B,2} \\ r_{B,1} &= r_{T,1} \\ r_{B,2} &= r_{T,2} \end{aligned} \tag{5.12}$$

The solution to this is given by

$$\begin{aligned} &r_{S,1} = 1, y_S = 2k \qquad k = 1, 2, \ldots \\ &r_{A,1} = 3, r_{A,2} = 2k \\ &r_{B,1} = 15, r_{B,2} = k \\ &r_{T,1} = 15, r_{T,2} = 3 \end{aligned} \tag{5.13}$$

and we see that $k = 2$ satisfies our constraint. Recalculating the other quantities,

$$W_{BT} = M^{-1}W_{AB} = \frac{1}{4}\begin{bmatrix} 21r_{S,1} & -8r_{S,2} \\ 3r_{S,1} & -24r_{S,2} \end{bmatrix} = \begin{bmatrix} 21/4 & -2 \\ 3/4 & -6 \end{bmatrix}$$

and we can determine that $|N(W_{BT})| = 30$ as required (i.e., $3 \times 4 \times 10/4 = 30$). Hence, we get away with having to produce only one extra row rather than three, assuming that the source can only produce 3 meaningful rows of data (and any number of columns).

5.3 Eliminating Cyclostatic Behavior

The fact that the decimator does not behave in a cyclostatic manner in figure 5-12 raises the question of whether factorizations that result in non-cyclostatic behavior in the decimator can always be found. The following example and lemma give an answer to this question for the special case of a decimator whose input is a rectangular lattice.

Example 5-2: Consider the system in figure 5-13 where a 2-D decimator is connected to a source actor that produces an array of (6,6) samples on each firing. The black dots represent the samples produced by the source and the circled black dots show the samples that the decimator should retain; these are the samples that lie on $LAT(M)$ intersected with the samples produced by the source. Since $|det(M)| = 4$, there are three possible ways to choose M_1, M_2. With $M_1 = M_2 = 2$ we see that the 6×6 array can be tiled with 2×2 arrays such that 1 sample is produced in each 2×2 array (figure 5-13 (b)). However, since some of the 2×2 blocks retain the sample on the left-bottom corner and some the sample on the right-bottom corner, a time-varying phase would have to be used to determine which sample should be output on a given invocation. There are two other ways to choose M_1, M_2 so that their product is 4: $M_1 = 1, M_2 = 4$ and $M_1 = 4, M_2 = 1$. Figures 5-13 (b),(c) illustrate the tiling with these choices. For both

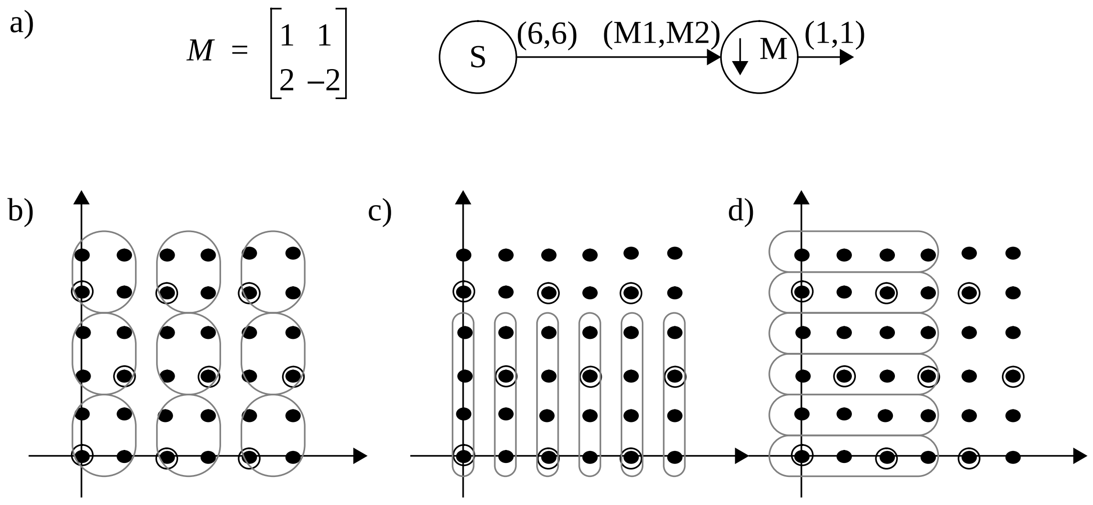


Figure 5-13 An example to illustrate that two factorizations always exist that result in non-cyclostatic behavior with the decimator. a) The system. b) M1=2, M2=2. c) M1=1, M2=4. d) M1=4, M2=1

these choices, the repetitions of the source is different than (1,1); hence the tiling isn't complete but can be completed if the source produces more data as specified by the solution to the balance equations ((2,1) and (1,2) respectively). In figure 5-13 (c) also, we see that for every (1,4) consumed, (1,1) is produced, although again, a time-varying phase will be required. However, in figure 5-13(d), we see that on some invocations, *no* samples are produced (that is, (0,0) samples are produced) while in some invocations, 2 samples are produced. This raises the question of whether there is always a factorization that ensures that the decimator produces (1,1) for all invocations. The following lemma ensures that for any matrix, there are always two factorizations of the determinant such that the decimator produces (1,1) for all invocations. Also, this example illustrates two other points: it is only *sufficient* that the decimator produce 1 sample on each invocation for the additional constraints on decimator outputs to be satisfied by the balance equation solution. It is only *sufficient* that the support matrix on the decimators output be integer valued for the additional constraints to be satisfied. Indeed, we have

$$W_{SM} = \begin{bmatrix} 6r_{S,1} & 0 \\ 0 & 6r_{S,2} \end{bmatrix}, W_{MO} = \begin{bmatrix} 3r_{S,1} & 1.5r_{S,2} \\ 3r_{S,1} & -1.5r_{S,2} \end{bmatrix},$$

where W_{MO} is the support matrix on the decimators output. For the case where $M1 = 4, M2 = 1$, we have $r_{S,1} = 2, r_{S,2} = 1$, making W_{MO} non-integer valued. However, we do have that $|N(W_{MO})| = |N(W_{SM})|/|det(M)|$, despite the fact that W_{MO} is non-integer valued and the decimator is cyclostatic.

Lemma 5-6: If

$$M = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

is any non-singular, integer 2x2 matrix, then there are at most two factorizations (and at least one) of $|det(M)|$, $A_1B_1 = |det(M)|$ and $A_2B_2 = |det(M)|$ such that if $M_1 = A_1, M_2 = B_1$ or $M_1 = A_2, M_2 = B_2$ in figure 5-13, then the decimator produces (1,1) for all invocations. Moreover,

$$A_1 = gcd(a, b), B_1 = \frac{|det(M)|}{gcd(a, b)}, \text{ and } A_2 = \frac{|det(M)|}{gcd(c, d)}, B_2 = gcd(c, d)$$

Remark: Note that $gcd(a, 0) = a$; hence, if M is diagonal, the two factorizations are the same and there is only one unique factorization. This implies that for rectangular decimation, there is only one way to set the MDSDF parameters and get non-cyclostatic behavior.

Proof: Firstly, note that since $det(M) = ad - bc$, $gcd(a, b)$ divides $det(M)$ and $gcd(c, d)$ divides $det(M)$. The decimator keeps samples with coordinates given by

$$M\begin{bmatrix} k_1 \\ k_2 \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} \tag{5.14}$$

or $ak_1 + bk_2 = x$ and $ck_1 + dk_2 = y$; hence, x and y have to be multiples of $gcd(a, b)$ and $gcd(c, d)$ respectively. Suppose $y = 0$. Then, with a little algebra, it can be seen that the smallest positive, non-zero value of x that solves equation 5.14 (meaning that k_1, k_2 are integers) is given by $x = |det(M)|/gcd(c, d)$. Similarly, if $x = 0$, the smallest non-zero, positive value of y is given by $y = |det(M)|/gcd(a, b)$. Hence, any rectangle consumed by the decimator cannot have a vertical dimension of greater than $gcd(c, d)$ or a horizontal dimension of greater than $gcd(a, b)$ since if it did, then at least two samples that are kept by the decimator will fall inside the rectangle based at the origin. So it remains to show that the two factorizations given in the statement of the lemma do in fact result in one sample being kept on *all* invocations. Fix some value for x that is a multiple of; call it x_0. Let y_0 be the smallest positive integer solution to equation 5.14. Then, the next positive integer y that solves equation 5.14 is given by $y_0 + |det(M)|/gcd(a, b)$. To see this, note that and $k_2 = (ay_0 - cx_0)/(ad - bc)$ are both integers. We want to determine the smallest positive constant j such that makes

$$k_1' = \frac{dx_0 - b(y_0 + j)}{ad - bc} \text{ and } k_2' = \frac{a(y_0 + j) - cx_0}{ad - bc}$$

also integers. Rearranging the above expressions, we get

$$k_1' = k_1 - \frac{bj}{ad - bc} \text{ and } k_2' = k_2 + \frac{aj}{ad - bc}$$

Clearly, $j = |det(M)|/gcd(a, b)$ makes both k_1' and k_2' integers. It is also the smallest: let $m = ad - bc$. Then, we have that $bj/m = i_1$ for some integer i_1 and $aj/m = i_2$ for some integer i_2. Hence, $i_1 a = i_2 b$ giving us the claimed value of j as the smallest such value.

A symmetric argument shows that if y is fixed, then the values of x that solve equation 5.14 differ by $|det(M)|/gcd(c, d)$.

Without loss in generality, consider the rectangle of dimensionality given by the first of the factorings in the statement of the lemma. When this rectangle is placed at the origin, only the sample at the origin falls inside it and is output by the decimator (see figure 5-14). Invocation (0,1) of the decimator would consume the rectangle whose lower left corner is at $y = |det(M)|/gcd(a, b)$, $x = 0$; this also contains only one sample that is output by the decimator (namely, the lower left corner sample). Clearly, as the rectangle is moved up along the y-axis by steps of $|det(M)|/gcd(a, b)$, the sample kept is always the one on the lower left corner. Now consider moving the rectangle to the right in steps of $gcd(a, b)$. None of these rectangles can contain two samples which do not have the same x coordinate. This is because the x coordinate in the solution equation 5.14 has to be a multiple of $gcd(a, b)$. These rectangles cannot contain two samples which have the same x coordinate either because, as was shown, when x is fixed, the y values differ by $|det(M)|/gcd(a, b)$. Hence, all of the rectangles that the decimator consumes will contain exactly one sample that falls on the output lattice, allowing the decimator to be non-cyclostatic. **QED.**

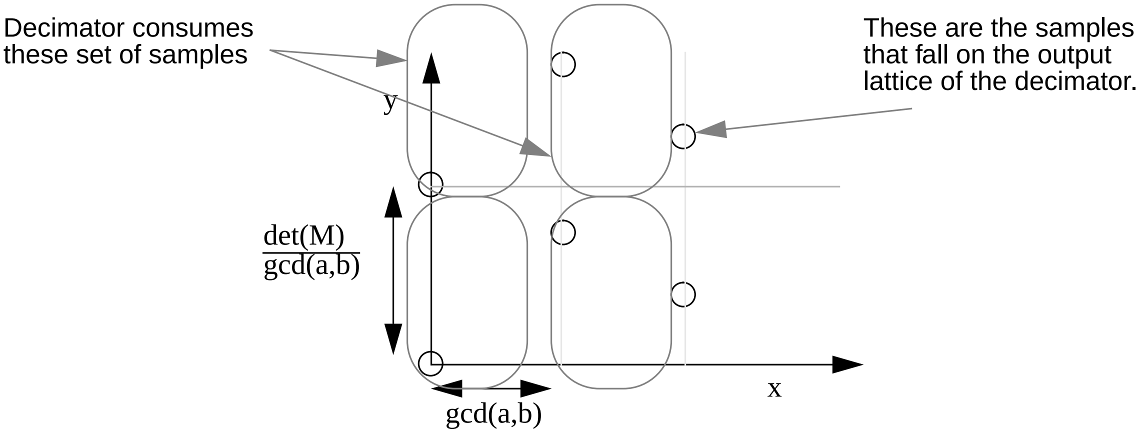

Decimator consumes these set of samples
These are the samples that fall on the output lattice of the decimator.
y
det(M)
gcd(a,b)
x
gcd(a,b)

Figure 5-14 Figure to illustrate proof of lemma 5-6.

For the example matrix in the figure 5-13, we can see that the two factorizations that work from the above equation are $A_1 = 1, B_1 = 4$ and $A_2 = 2, B_2 = 2$; this is what we see from figure 5-13 (b),(c).

5.4 Delays in the Generalized Model

Delays can be interpreted as translations of the buffer of produced values along the vectors of the support matrix (in the renumbered data space) or along the vectors in the basis for the sampling lattice (in the lattice data space). Figure 5-15 illustrates a delay of (1,2) on a non-rectangular lattice.

5.5 Summary of Generalized Model

In summary, our generalized model for expressing non-rectangular systems has the following semantics:

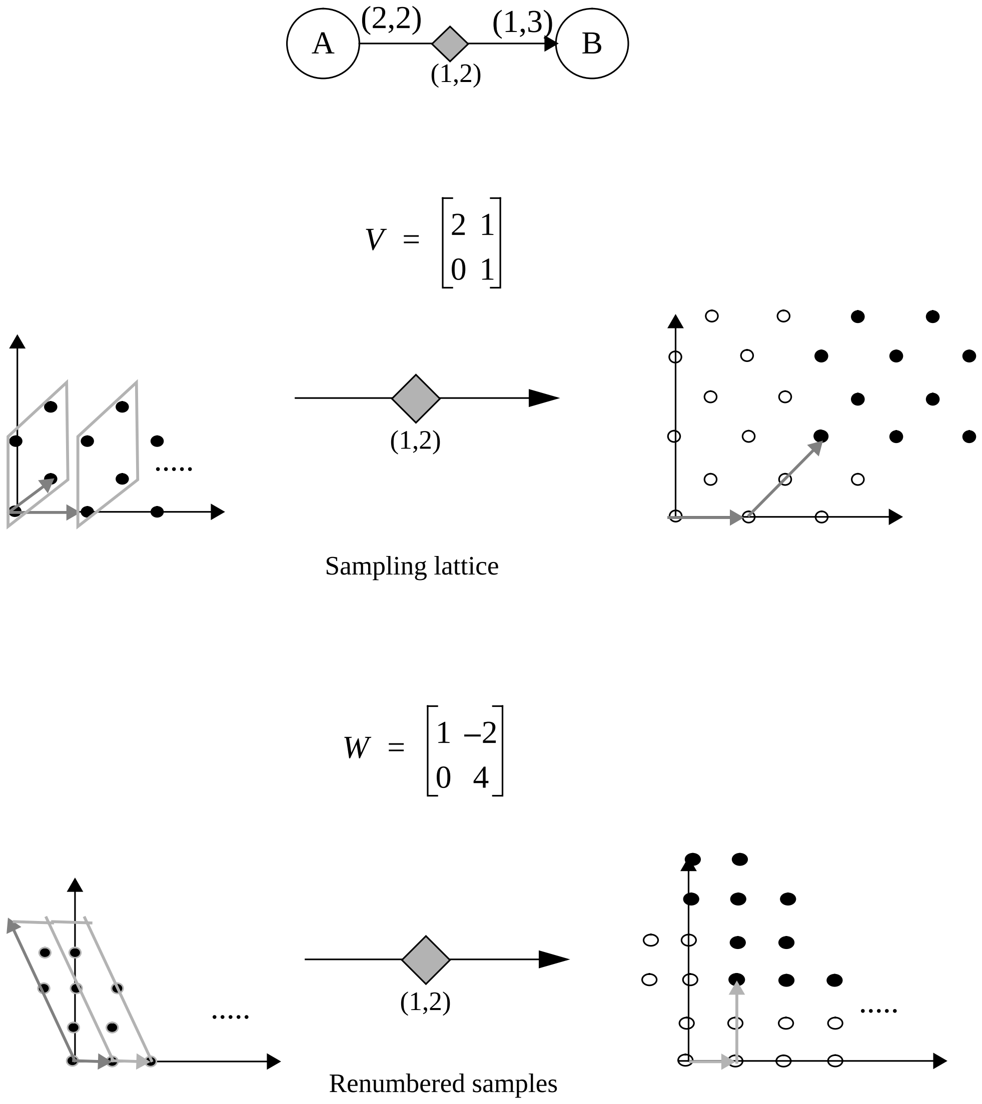
- Sources produce data in accordance with definition 5-4. The support matrix and lattice-generating matrix on the sources output arcs are specified by the source. The source produces a generalized (S_1, S_2) rectangle of data on each firing.

- An expander with expansion matrix L consumes (1,1) and produces the set of samples in $FPD(L)$ that is ordered as a generalized (L_1, L_2) rectangle of data where L_1, L_2 are positive integers such that $L_1L_2 = |det(L)|$.

- A decimator with decimation matrix M consumes a rectangle (M_1, M_2) of data where this rectangle is interpreted according to the way it has been ordered (by the use of some rectangularizing function) by the actor feeding the decimator. It produces (1,1) on average. Unfortunately, there does not seem to be any way of making the decimators output any more concrete.

- On any arc, the global ordering of the samples on that arc is established by the actor feeding the arc. The actor consuming the samples follows this ordering.

A set of balance equations are written down using the various factorizations. Additional constraints for arcs that feed a decimator are also written down. These are solved to yield the repetitions matrix for the network. A scheduler can then construct a static schedule by firing firable nodes in the graph until each node has been fired the requisite number of times as given by the repetitions matrix.



$$V = \begin{bmatrix} 2 & 1 \\ 0 & 1 \end{bmatrix}$$

$$W = \begin{bmatrix} 1 & -2 \\ 0 & 4 \end{bmatrix}$$

Figure 5-15 Delays on non-rectangular lattices

5.6 Multistage Sampling Structure Conversion Example

In this section, we illustrate another example; this is a practical example of a system that does sampling structure conversion for video signals. We will show how the semantics developed above can be used to specify and determine a schedule for the system. This example is drawn from [Mand93].

5.6.1 Video Signals

A video signal can be thought of as a three-dimensional signal where two of the dimensions correspond to the height(vertical) and width(horizontal) of the image while the third dimension is time. The current practice is to sample the signal in two of the dimensions and keep the third dimension continuous. The vertical and temporal directions are sampled (this processed is called scanning) while the horizontal direction is not. Hence, as a discrete-time signal, a video signal is two-dimensional, with samples occupying the *vertico-temporal* plane. Note that an actual video signal is one-dimensional since the resulting lines (from the scanning) are abutted to form a one-dimensional signal [Dubo85]. There are currently two types of vertico-temporal lattices in use: the progressively scanned signal, which corresponds to a rectangular lattice, and the 2:1 interlaced signal, which corresponds to a "quincunx" lattice. There are trade-offs between using these two lattices in various types of distortions and interline flicker but in general, 2:1 interlaced scanning is preferable to sequential scanning for a given scanning density [Dubo85][1]. The two lattices are shown in figure 5-16.

5.6.2 Sampling Structure Conversion

An application of considerable interest in current television practice is the format conversion from 4/3 aspect ratio to 16/9 aspect ratio for 2:1 interlaced TV signals. It is well known in one-dimensional signal processing theory that sample rate conversion can be

[1] The Advisory Committee on Advanced Television Service, a committee formed by the FCC in 1987 to assist the FCC in establishing an HDTV standard for the United States, tested 4 all-digital HDTV proposals (DigiCipher, DSC-HDTV, AD-HDTV, CCDC) and found that systems using interlaced scanning (DigiCipher and AD-HDTV) had the best quality overall.[Hopk93].

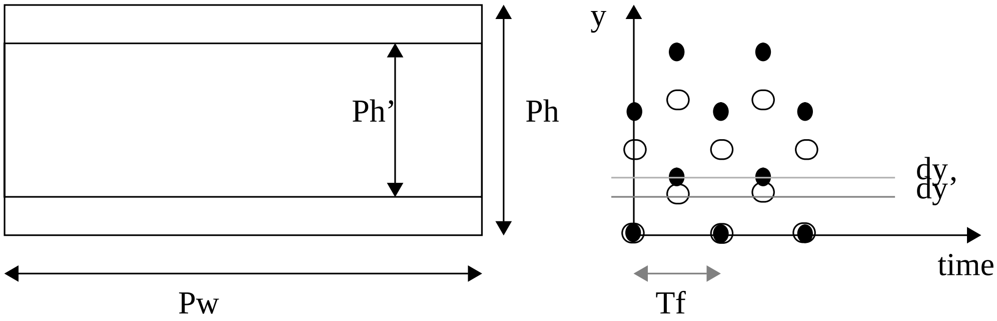


Figure 5-17 Picture sizes and lattices for the two aspect ratios 4/3 and 16/9

done efficiently in many stages [Rams84]. Similarly, it is more efficient to do both sampling rate and sampling structure conversion in stages for multidimensional systems. The two aspect ratios and the two lattices are shown in figure 5-17.

The relationship between the height and width for the two aspect ratios is given by $P_h = (3/4)P_w$ and $P_h' = (9/16)P_w$. Since the number of lines N is the same in both cases, the interline distances d_y' and d_y, corresponding to the 16/9 and 4/3 aspect ratios respectively, are related as $d_y' = P_h'/N = (3/4)(P_h/N) = (3/4)d_y$. Thus the bases for the lattices for the two aspect ratios are given by

$$A = \begin{bmatrix} 2T_f & T_f \\ 0 & d_y \end{bmatrix} \text{ and } A' = \begin{bmatrix} 2T_f & T_f \\ 0 & (3/4)d_y \end{bmatrix} \tag{5.15}$$

for the 4/3 and 16/9 aspect ratios respectively. By the CCIR System M standard 625/2:1/50, $T_f = 1/50\ s$ and $N = 625$. One way to do the conversion between the two lattices

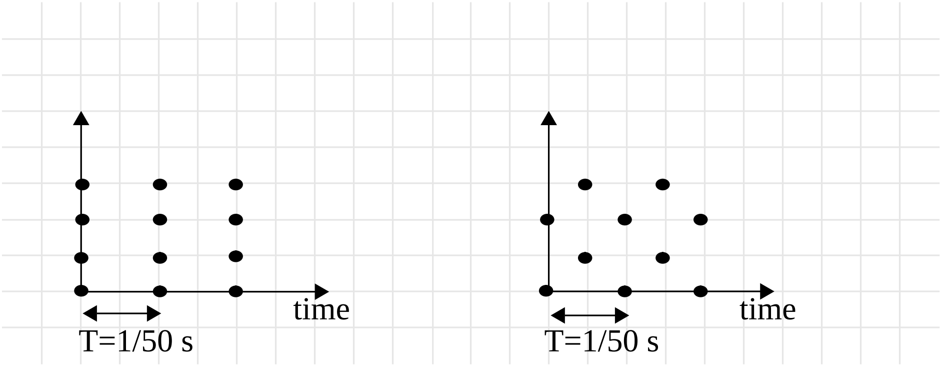


Figure 5-16 Progressive scanning and 2:1 interlaced scanning in the vertico-temporal plane

above is as shown below in figure 5-18 [Mand93]. Below each arc, the desired lattice on that arc is shown. For simplicity, the filters that are needed between the upsampling and downsampling stages are omitted. From the lattices shown, we can compute the required values for L_1, L_2, M:

$$V_{SA} = A = \begin{bmatrix} 2T_f & T_f \\ 0 & d_y \end{bmatrix}, V_{AB} = \begin{bmatrix} T_f & 0 \\ 0 & d_y/2 \end{bmatrix}, V_{BC} = \begin{bmatrix} T_f & 0 \\ 0 & d_y/4 \end{bmatrix}, \text{and}$$

$$V_{CT} = A' = \begin{bmatrix} 2T_f & T_f \\ 0 & 3d_y/4 \end{bmatrix}$$

Since using realistic values for T_f, d_y will make all the calculations rather ugly, we will just use $T_f = 1, d_y = 1$ without any loss in generality.

By theorem 5-1,

$$L_1 = V_{AB}^{-1}V_{SA} = \begin{bmatrix} 1/T_f & 0 \\ 0 & 2/d_y \end{bmatrix}\begin{bmatrix} 2T_f & T_f \\ 0 & d_y \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 0 & 2 \end{bmatrix}, L_2 = V_{BC}^{-1}V_{AB} = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}, \text{and}$$

$M = V_{BC}^{-1}V_{CT} = \begin{bmatrix} 2 & 1 \\ 0 & 3 \end{bmatrix}$. Note that these matrices do not depend on T_f and d_y.
Suppose

$$|det(L_1)| = 4 = 2 \times 2, |det(L_2)| = 2 = 1 \times 2, \text{and } |det(M)| = 6 = 3 \times 2$$

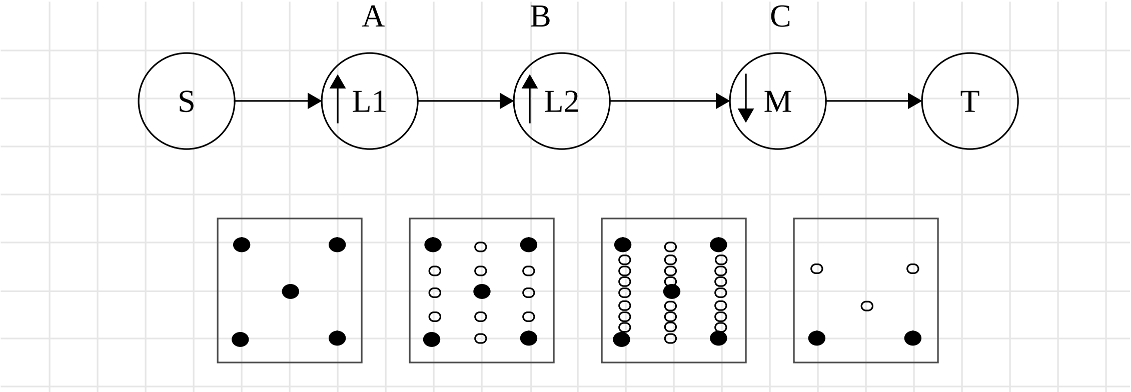


Figure 5-18 System for doing multistage sampling structure conversion from 4/3 aspect ratio to 16/9 aspect ratio for a 2:1 interlaced TV signal.

Let us assume that S produces samples in $LAT(V_{SA}) \cap N\left(\begin{bmatrix} 2 & 0 \\ 0 & 8 \end{bmatrix}\right)$. The support matrix for this can be computed as

$$V_{SA}^{-1}\begin{bmatrix} 2 & 0 \\ 0 & 8 \end{bmatrix} = \frac{1}{2}\begin{bmatrix} 1 & -1 \\ 0 & 2 \end{bmatrix}\begin{bmatrix} 2 & 0 \\ 0 & 8 \end{bmatrix} = \begin{bmatrix} 1 & -4 \\ 0 & 8 \end{bmatrix}$$

Suppose further that the samples that the source produces are rectangularized in the following obvious way: sample at (0,0) is (0,0); sample at (1,1) is (1,0); sample at (0,2) is (0,1); sample at (1,3) is (1,1) etc. So S produces (2,8) where the rectangle is understood to be as above. We can write down the following balance equations:

$$\begin{aligned} 2r_{S,1} &= 1r_{A,1} \\ 8r_{S,2} &= 1r_{A,2} \\ 2r_{A,1} &= 1r_{B,1} \\ 2r_{A,2} &= 1r_{B,2} \\ 1r_{B,1} &= 3r_{C,1} \\ 2r_{B,2} &= 2r_{C,2} \\ r_{C,1} &= r_{T,1} \\ r_{C,2} &= r_{T,2} \end{aligned} \tag{5.16}$$

These solve to

$$\begin{aligned} r_{S,1} &= 3, r_{S,2} = 1 \\ r_{A,1} &= 6, r_{A,2} = 8 \\ r_{B,1} &= 12, r_{B,2} = 16 \\ r_{C,1} &= 4, r_{C,2} = 16 \end{aligned} \tag{5.17}$$

Again, we can calculate the various support matrices symbolically in order to verify whether

$$|N(W_{CT})| = |N(W_{BC})| / |det(M)| \tag{5.18}$$

We have

$$|N(W_{BC})| = det\left(\begin{bmatrix} 2r_{S,1} & 0 \\ 0 & 32r_{S,2} \end{bmatrix}\right) = 64r_{S,1}r_{S,2}\text{, and}$$

$$W_{CT} = M^{-1}W_{BC} = \begin{bmatrix} r_{S,1} & -(16/3)r_{S,2} \\ 0 & (32/3)r_{S,2} \end{bmatrix}$$

With the values obtained from the balance equations, equation 5.18 turns out to be not satisfied because $|N(W_{CT})| = 33$! In actuality, since there are 625 lines vertically, a more accurate model for the source is that it produces samples in (2,625), again according to the rectangle defined above. However, the resulting matrix

$$W_{CT} = \begin{bmatrix} r_{S,1} & -(1250/3)r_{S,2} \\ 0 & (2500/3)r_{S,2} \end{bmatrix}$$

presents a rather nasty challenge for determining the number of integer points inside its FPD (since it is not an integer matrix with the solution to the balance equations). However, in the model where the source produces (2,8), we can force W_{CT} to be an integer matrix by making $r_{S,2}$ a multiple of 3. If the source is in fact trying to produce 625 lines vertically, then the smallest multiple of 3 that is greater than or equal to 625 is given by 209; we can make this the vertical blocking factor. As in the previous example, we can try to find better models for the source, one that will allow it to cover exactly 625 lines vertically for example. Of-course, this is assuming that the source actor can be coded in a flexible way that allows it to produce as much data or as little as desired.

5.7 Computing the Integer Volume of a Rational Matrix

Since the augmented balance equations frequently require the quantity $|N(A)|$ (which we call the *integer volume* of A) for a non-integer matrix (but rational) A, we develop a method in this section that is better than the brute force method of simply counting the integer points. First we note that obvious methods do not work and that the problem appears to be more difficult than it would appear.

5.7.1 Some simple approaches

5.7.1.1 Scaling

The most obvious technique to try is to scale the matrix by an appropriate constant to make all the entries integer valued and deduce the integer volume from that of the resulting integer valued matrix. If we scale the matrix A by c to make it integer valued, then the determinant, and thus the integer volume, of cA is given by $c^2|det(A)|$, and this is an integer. However, this tells us nothing about the integer volume of A since in general, $|det(A)|$ may not even be an integer. Even if $|det(A)|$ is an integer, it is not the case that its integer volume is equal to $|det(A)|$; for example, consider $diag(1/2, 2)$ ($diag$ means a diagonal matrix). The integer volume of $diag(1/2, 2)$ is 2, not 1.

5.7.1.2 Rounding

However, for a diagonal matrix $diag(p, q)$ it is easy to see that the integer volume is actually given by $\lceil p \rceil \lceil q \rceil$, where $\lceil \bullet \rceil$ is the ceiling operator. Hence, we might wonder whether there is some way of rounding the entries of an arbitrary matrix so that the resulting integer matrix has the same number of integer points in its FPD as the original matrix. However, this does not seem to be the case. Consider the matrix

$$\begin{bmatrix} 5/3 & 1/2 \\ 0 & 5/3 \end{bmatrix}.$$

It can be verified that this matrix has an integer volume of 3. However, there is no way of rounding the entries (either floors or ceilings) of this matrix to get an integer matrix that has a determinant of 3.

5.7.1.3 Smith Form Decomposition

Another approach is to try to use the Smith form decomposition. The Smith form decomposition of an integer matrix A gives two integer, unimodular matrices U, V such that $UAV = \Lambda$, where Λ is a diagonal matrix, and this decomposition can be computed in polynomial time by the Lenstra-Lenstra-Lovasz algorithm. If A is a rational-valued matrix, then we can simply represent it as a constant times an integer matrix, and use the Smith form decomposition to get a rational-valued diagonal matrix, the integer volume of which is easy to compute as shown above. However, the integer volume of this diagonal matrix is not equal to the integer volume of the original matrix in general as the following counter-example shows. Consider the matrix (it has an integer volume of 47):

$$\begin{bmatrix} 21/2 & -3/2 \\ 3/2 & -9/2 \end{bmatrix} \tag{5.19}$$

$$= \begin{bmatrix} 21 & -3 \\ 3 & -9 \end{bmatrix} \frac{1}{2}.$$

The Smith form decomposition is given by

$$\frac{1}{2}\begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}\begin{bmatrix} 21 & -3 \\ 3 & -9 \end{bmatrix}\begin{bmatrix} 1 & 1 \\ 2 & 3 \end{bmatrix} = \frac{1}{2}\begin{bmatrix} 15 & 0 \\ 0 & -12 \end{bmatrix}.$$

The integer volume of the diagonal matrix is given by 8*6=48, and this is not equal to the integer volume of the original matrix. The problem with the Smith form approach is that the unimodular matrix transformations do not result in a one-to-one mapping of the integer points in $FPD(A)$ to $FPD(\Lambda)$.

5.7.2 A Method for Computing the Integer Volume of a Rational Matrix

More insightful techniques are clearly needed. The development here is based on the proof we gave for the integer matrix case; namely, we develop the method for an upper triangular matrix and use lemma 5-2 to transform any matrix to an upper triangular matrix.

Fact 5-1: For any rational-valued matrix A, there is an integer unimodular matrix C such that CA is a rational-valued, upper triangular matrix.

Proof: Immediate from the fact that we can write any rational matrix as a constant times an integer matrix, and apply lemma 5-2 to the integer matrix. **QED**

Fact 5-2: Given a rational-valued matrix A, and a corresponding integer unimodular matrix C such that CA is upper-triangular, $|N(CA)| = |N(A)|$.

Proof: Similar to the last part of the proof of lemma 5-3. Let $\hat{k} \in N(A)$. Then $\exists \hat{x} \in [0, 1)^2$ s.t. $A\hat{x} = \hat{k}$. Since $CA\hat{x} = C\hat{k}$ is an integer vector, every point in $N(A)$ maps to some point in $N(CA)$. Similarly, if $\hat{k} \in N(CA)$, then $\exists \hat{x} \in [0, 1)^2$ s.t. $CA\hat{x} = \hat{k}$. So $C^{-1}\hat{k} = A\hat{x}$, and $C^{-1}\hat{k}$ is an integer vector since C is integer valued and unimodular. So every point in $N(CA)$ maps to some point in $N(A)$; this shows that the mapping is one-to-one. **QED**

Hence, the problem is to compute the integer volume of a rational-valued, upper triangular matrix. From now on, we only consider 2x2 matrices. Let the rational-valued, upper triangular matrix be given by $\begin{bmatrix} a & b \\ 0 & c \end{bmatrix}$.

Fact 5-3: Let $x \in [0, 1]$. In the interval $[x, x + a)$, the number of integers in the interval is $\lfloor a \rfloor$ if $0 < x \le \lceil a \rceil - a$ and $\lceil a \rceil$ otherwise.

Since the first column of A is aligned with the horizontal axis, we imagine moving the interval $[0, a]$ along the vector $\begin{bmatrix} b & c \end{bmatrix}^T$. We are interested in intervals $[bk/c, bk/c + a]$ for $k = 0, 1, \ldots, \lfloor c \rfloor$ since these are the intervals for which the vertical dimension is an integer (along the vector $\begin{bmatrix} b & c \end{bmatrix}^T$). Clearly, the number of integers in $[bk/c, bk/c + a]$ is the same as the number of integers in $[bk/c - \lfloor bk/c \rfloor, bk/c + a - \lfloor bk/c \rfloor]$. Hence, by fact 5-3, whenever

$$bk/c - \lfloor bk/c \rfloor \le \lceil a \rceil - a, \tag{5.20}$$

the number of integers for that k is $\lfloor a \rfloor$ and $\lceil a \rceil$ for the rest. Therefore, we want to know the number of values of k, $k = 1, \ldots, \lfloor c \rfloor$ for which equation 5.20 is satisfied; define this number to be K. Define $b \equiv b_1/b_2$ and $c \equiv c_1/c_2$, where b_1, b_2, c_1, c_2 are integers and the fractions are in reduced form. Define $gc = gcd(b_1c_2, b_2c_1)$, $e_1 = b_1c_2/gc$, and $e_2 = b_2c_1/gc$. Also define $\lceil a \rceil - a \equiv a_1/a_2$ in reduced form with a_1, a_2 integers. The left hand side of equation 5.20 becomes

$$e_1k/e_2 - \lfloor e_1k/e_2 \rfloor.$$

Moreover, since $x \bmod y \equiv x - \lfloor x/y \rfloor y$ for integers x, y, we have

$$e_1k/e_2 - \lfloor e_1k/e_2 \rfloor = (e_1k \bmod e_2)/e_2 .$$

As k increases, the above equation is periodic with period e_2; hence, it suffices to determine the number K' of k in $\{1, 2, \ldots, e_2 - 1\}$ that satisfies

$$(e_1k \bmod e_2)/e_2 \le a_1/a_2 . \tag{5.21}$$

and to determine the number K'' of k in $\{0, \ldots, \lfloor c \rfloor \bmod e_2\}$ that satisfies equation 5.21. (If c is an integer, then K'' is the number of k in $\{0, \ldots, \lfloor c \rfloor \bmod e_2 - 1\}$ that satisfies equation 5.21. This is because if c is an integer, then the very last interval vertically is right on the side of the FPD and does not therefore count). From this, $K = K'\lfloor \lfloor c \rfloor / e_2 \rfloor + K''$. Since e_1, e_2 are coprime, $e_1k \bmod e_2$ takes each value in the set $\{1, \ldots, e_2 - 1\}$ as k varies in $\{1, \ldots, e_2 - 1\}$. Hence, $K' = \lfloor (a_1e_2)/a_2 \rfloor$. However, it is not clear whether K'' can be determined efficiently. However, if $\lfloor c \rfloor \bmod e_2$ is sufficiently small, then enumeration is not as big of a problem, and in practice, the development above leads to a more efficient way of computing the integer volume, although the formula is rather messy. We can collect the results above into the following lemma:

Lemma 5-7: The integer volume $|N(A)|$ of a rational-valued, upper triangular 2x2 matrix

$$A = \begin{bmatrix} a & b \\ 0 & c \end{bmatrix}$$

is given by the formula

$$|N(A)| = K\lfloor a \rfloor + (\lceil c \rceil - K)\lceil a \rceil$$

where $K = K'\lfloor \lfloor c \rfloor / e_2 \rfloor + K''$, $K' = \lfloor (a_1e_2)/a_2 \rfloor$, K'', a_1, a_2, e_2 are all as defined before. Note that if a is an integer, then $|N(A)| = a\lceil c \rceil$ as expected since it is a that determines how many points there are in each horizontal interval.

5.7.3 Examples

Example 5-3: Consider the matrix in equation 5.19:

$$A = \begin{bmatrix} 21/2 & -3/2 \\ 3/2 & -9/2 \end{bmatrix}.$$

This can be transformed to an upper triangular matrix as:

$$\frac{1}{2}\begin{bmatrix} 1 & -6 \\ 1 & -7 \end{bmatrix}\begin{bmatrix} 21 & -3 \\ 3 & -9 \end{bmatrix} = \begin{bmatrix} 3/2 & 51/2 \\ 0 & 30 \end{bmatrix}.$$

The various quantities are:

$\lceil a \rceil - a \equiv a_1/a_2 = 1/2$, $b \equiv b_1/b_2 = 51/2$, $e_1 = 17, e_2 = 20$, $\lfloor \lfloor c \rfloor / e_2 \rfloor = 1$, and

$$K' = \lfloor (a_1 e_2)/a_2 \rfloor = 10.$$

To compute K'', we need the number of k in $\{0, \ldots, 30 \bmod 20 - 1 = 9\}$ that satisfy

$$17k \bmod 20 \le 10.$$

By counting, $K'' = 3$. Hence $K = K' \lfloor \lfloor c \rfloor / e_2 \rfloor + K'' = 13$. So

$$|N(A)| = K\lfloor a \rfloor + (\lceil c \rceil - K)\lceil a \rceil = 47.$$

Example 5-4: This is a more dramatic example where the counting step in the above method is extremely small. Consider

$$A = \begin{bmatrix} 125/3 & 61/7 \\ 0 & 122/3 \end{bmatrix}.$$

The various quantities are:

$\lceil a \rceil - a \equiv a_1/a_2 = 1/3$, $b \equiv b_1/b_2 = 61/7$, $e_1 = 3, e_2 = 14$, $\lfloor \lfloor c \rfloor / e_2 \rfloor = 2$, and

$$K' = \lfloor (a_1 e_2)/a_2 \rfloor = 4.$$

To compute K'', we need the number of k in $\{0, \ldots, 40 \bmod 14 = 12\}$ that satisfy

$$3k \bmod 14 \le 4.$$

Since $e_2 - 1 = 13$, we just need to check one value of k, namely $k = 13$ to determine K''. Since $39 \bmod 14 = 11 > 4$, $K'' = K' = 4$. Hence $K = K' \lfloor \lfloor c \rfloor / e_2 \rfloor + K'' = 12$, and

$$|N(A)| = K\lfloor a \rfloor + (\lceil c \rceil - K)\lceil a \rceil = 12 \times 41 + 29 \times 42 = 1710,$$

and this has been verified by counting.

5.7.4 Some Final Thoughts on the Problem

The method given above, although much more efficient in practice than a brute force method, is not efficient in the technical sense of giving us a polynomial time algorithm for the problem. This is because the complexity of the above method is dominated by the counting step where there are potentially $e_2/2$ steps. Since $e_2 = b_2 c_1/gc$, where these quantities were defined before, e_2 is not bounded by a logarithmic function of any of the quantities that appear in the problem. However, unlike the brute force method, where counting has to be done in both directions, the counting step above is only along the vertical direction and is hence considerably more efficient than the brute force method in general.

Finally, the method above results in a formula that is messy and very non-intuitive, and is far from elegant. Perhaps there is no elegant solution to this problem at all, and it might even be NP-hard if the counting step turns out to be NP-hard (we would still have to give a reduction). Also, the generalization to higher dimensions will only be messier since we have relied on operations on the individual entries of the matrix, and not on matrix operations. Perhaps an algorithm can be developed, but we do not pursue it here.

5.8 Conclusions and Open Problems

We have described an extension of MDSDF to handle arbitrary sampling lattices. The key extensions have been to associate two parameters for each arc in the graph: the sampling lattice for the data on that arc, and a “support matrix” that describes the region of the space where current data has been produced. Equivalently, these two parameters can be specified for source actors, and can be determined for the other arcs by tracing the operations that each node in the graph performs. Since decimation and expansion are the only two actors (of signal processing interest) that change lattices, it should be straightforward to determine the lattice and support matrix for every arc in the graph given the information at the source (assuming that there are no cycles in the graph). Given these two parameters, we have shown how we can compute a global ordering for the data on the arcs, and how generalized “rectangles” can be defined. Using these, we can derive a set of decoupled balance equations in an analogous manner to the rectangular case. However, this is not enough; for decimators, some other constraints must also be satisfied. The directions of the repetitions of a node is also generalized to depend on the lattice and support matrix on outgoing and incoming arcs; that is, the direction can be different for an input arc from an output arc.

There are many open problems and issues that have not been tackled yet. Some of these are listed below:

- The issue of buffering efficiency and buffer implementations has not been addressed. It would be desirable to have systematic ways of determining schedules that minimize for code size and buffer-usage, as was done in the SDF case for well-ordered graphs and general acyclic graphs. The techniques in can be easily applied to the rectangular MDSDF case since rectangular MDSDF can be thought of as several independent SDF graphs (one for each dimension); with non-rectangular systems, it is less obvious how to extend the techniques of. The buffers would probably have to be implemented directly as linear arrays but indexed appropriately (at least for simulation) since techniques like using matrix and submatrix data-structures [Chen94] may not be feasible for non-rectangular support matrices.

- An extension of lemma 5-6 to arbitrary support regions would be desirable. This would give us a way of choosing factorizations that do not have the cyclostatic behavior that an arbitrary factorization generally does. This would also ensure that solving balance equations is sufficient and constraints of the type in equation 5.18 do not have to be solved. However, this appears to be difficult since there does not seem to be a clean, analytical way of expressing the ordering of samples as done in table 1. Since the understanding of how points on $LAT(M)$, where M is the decimation matrix, map to the rectangle under this ordering is needed before a claim of the sort made in lemma 5-6, this extension would be non-trivial.

- Methods of choosing factorizations that lead to the smallest solution to the balance equations are desirable, provided of-course that the direction of the resulting streams is acceptable.

- More complicated examples. The examples presented in this report have been those of simple, chain-structured graphs. Concrete examples of acyclic graphs that represent non-rectangular systems with a lot of inherent functional parallelism include directional decomposition filterbanks as described in [Bamb90], and hierarchical video coding applications [Bosv92].

- Examination of higher dimensions. The examples in this report have all been for 2-D MDSDF; generalizations to higher dimensions may be trickier. A concrete example of a 3 dimensional digital signal is a fully scanned TV signal (where the horizontal direction is scanned also).

- There are many other ways of doing the sampling-structure conversion [Mand93]. If more decimation stages are used, more of the constraints of the type in equation 5.18 have to be solved. Some of these ways of doing the conversion might be better in computational terms than other ways in that the repetitions of the various actors in between are lower for some ways than others. It would be interesting to explore systematic ways of evaluating the various ways.

6

Conclusion and Future Work

In this thesis, we have solved several scheduling problems for programs expressed as SDF and MDSDF graphs. Since the potentially large rate changes in SDF and MDSDF specifications can result in a code and buffer-memory size explosion, a primary focus has been on generating schedules that minimize code and data memory size. Code size optimality is obtained by generating single appearance schedules. Algorithms and heuristics are given that generate single appearance schedules optimized for buffering memory. In particular, three basic algorithms are developed and discussed: a dynamic programming algorithm that is used for post-optimization, and two heuristics: RPMC and APGAN. These two algorithms are complementary in the way they approach the problem, and an extensive experimental study confirms this by showing that for graphs that have irregular topologies and rate changes, RPMC exhibits better performance, while for graphs having regular topologies and rate changes, APGAN performs better. In addition APGAN is optimal for a class of SDF graphs that includes several practical systems. All these algorithms are shown to extend to MDSDF also. The three algorithms discussed have all been implemented in the Ptolemy framework, in the SDF domain.

In Chapter 5, the thesis shifts focus to a rather different type of problem: namely, generalizing the MDSDF model to permit arbitrary sampling lattices. Multidimensional multirate systems sampled on non-rectangular lattices are interesting and useful, and it is desirable to have a dataflow model capable of expressing such systems. We generalize the MDSDF model, and show that the key property of compile-time schedulability can be retained in the generalized model. In particular, we show that in the generalized model, the problem of solving balance equations becomes much more challenging, and we give an algorithm for solving the augmented balance equations that arise in this model. We also show a practical system using this model.

Below we outline some future directions for research in these areas.

6.1 Trading Code Size and Memory Size.

Our approach to joint code and data minimization in SDF graphs has been to minimize the amount of buffering memory required by single appearance schedules. While single appearance schedules are optimal with respect to program memory, it might be desirable in some instances to allow non-single appearance schedules in order to gain lower buffer requirements. As shown in Chapter 2, the lower bound on the amount of memory required by *any* schedule on an edge e that has $prd(e) = a$, $cns(e) = b$, and 0 delays is $a + b - gcd(a, b)$. The BMLB for this edge, the lower bound if we restrict the schedules to be *single* appearance schedules, is given by $ab/gcd(a, b)$. Clearly, there is a large gap between these two quantities. Correspondingly, the length of a schedule that achieves the first bound can be as high as $a + b$ while the length for the single appearance schedule is only 2 (neglecting looping code overhead). Hence, there is potential for trading-off these two parameters in interesting ways. The graph in figure 6-1 illustrates this trade-off rather dramatically. The BMLB for the graph is 10100 while the lower bound on the buffer size for any schedule is 200. The schedule $A\ \ 100(AB)$ also has a buffering requirement of 200, and is in fact the same schedule generated by the minimum buffer scheduling heuristic given in Chapter 2, except that it has been looped (or "compressed") optimally (the dynamic programming algorithm given in Chapter 3 can accomplish this for example). The schedule is a 2-appearance schedule, and is likely to be much more useful in practice than the single appearance schedule because the increase in code size is negligible compared to

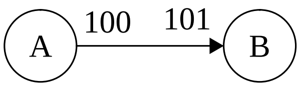


Figure 6-1 A graph to illustrate the dramatic decrease in buffering memory by allowing 2-appearance schedules.

the decrease in buffering memory. Hence, it would be interesting to develop systematic ways of generating k-appearance schedules optimized for buffering memory usage, where no actor appears more than k times and k is fairly small. One approach is to simply use the heuristic from Chapter 2 and use the dynamic programming algorithm to generate an optimal looping hierarchy. However, in addition to this algorithm taking exponential time, it is not clear whether the optimal looping structure will ensure that each actor appears only a “few” times. More desirable would be an algorithm that can systematically expand a single appearance schedule until the buffering requirement has been lowered suitably. While it is easy to devise heuristics to do this right away, it would be more interesting to develop algorithms that are optimal, at least for a restricted class of graphs.

6.2 Some Open Complexity Issues

As mentioned in Chapter 3, the complexity of the `legalCutIntoBddSets` problem remains open although we strongly suspect it to be NP-complete. If it turns out to be polynomial time solvable, it would be interesting to see its impact on the RPMC heuristic; whether RPMC would exhibit better performance. Recall that `legalCutIntoBddSets` is used by RPMC, which is itself a heuristic.

The complexity of generating buffer-optimal single appearance schedules for delayless, multirate SDF graphs also remains open. The NP-completeness result given in Chapter 3 allows delays on edges; however, the presence of rate changes should more than make up in difficulty whatever is lost by not allowing delays. Recall that the NP-completeness result of Chapter 3 applied to homogenous graphs with delays allowed. In a homogenous graph without delays, all schedules have the same buffer memory requirement; namely, $|E|$, the number of edges. However, if the graph is not homogenous, and rate changes are allowed, then different topological sorts result in different buffering costs (even if there are no delays of-course), and the problem appears to be quite difficult; we strongly suspect that the problem is NP-complete.

There is also the interesting question of whether all SDF graphs have a valid schedule of length polynomial in the size of the graph. For example, it might be the case that for some strongly connected, tightly interdependent graph, *all* schedules have the property that they cannot be compressed to a polynomial length by the dynamic programming algorithm presented in Chapter 3. For instance, consider a string like $ABCBACBC$. This string is incompressible in the sense that no loops (of iteration counts more than 1) can be organized in it in order to shorten its length. This issue might in fact be a rather deep problem, that might require Kolmogorov complexity-theoretic or information theoretic arguments. We recall that there are proofs as to why an arbitrary integer k requires $\Theta(\log(k))$ bits in general to encode it; it is based on the argument that most binary strings have maximum entropy, and are incompressible in a Kolmogorov-complexity theoretic sense [Cove91]. Perhaps, the strings corresponding to SDF schedules can be mapped to integers in some manner in order to apply these arguments to SDF schedules. In any case, we suspect that this problem is not simple.

6.3 MDSDF and GMDSDF Scheduling

SDF is a fairly mature model of computation now, at least compared to MDSDF. It is not yet clear whether MDSDF will prove to be as useful and intuitive to use as SDF is. Certainly, many specifications in MDSDF (for example, the matrix multiplication specification in [Lee93]) are cryptic and require considerable work to understand. Even so, a number of multiprocessor scheduling problems have to be solved for MDSDF graphs. It might be possible to extend techniques discussed in Chapter 2 to MDSDF. Some recent work on multidimensional retiming [Pass94][Denk96] appears to be promising. Also, we outlined several open issues in the GMDSDF model at the end of Chapter 5.

REFERENCES

[Aho88]

A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers Principles, Techniques, and Tools*, Addison-Wesley, 1988.

[Alli86]

L. Allison, *A Practical Introduction to Denotational Semantics*, Cambridge University Press, 1986.

[Alur95]

R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, and others, “The Algorithmic Analysis of Hybrid Systems,” *Theoretical Computer Science*, **Vol. 138, No.1**, February 1995.

[Ambl92]

A. L. Ambler, M. M. Burnett, and B. A. Zimmerman, “Operational Versus Definitional: A Perspective on Programming Paradigms,” *IEEE Computer Magazine*, **Vol. 25**, **No. 9**, September, 1992.

[Ansa88]

R. Ansari, S. H. Lee, “Two Dimensional Nonrectangular Interpolation, Decimation Filterbanks”, *Proceedings of the ICASSP ‘88*, New York, USA, 1988.

[Ashc95]

E. A. Ashcroft, A. A. Faustini, R. Jagannathan, W. W. Wadge, *Multidimensional Programming*, Oxford University Press, 1995.

[Bacc93]

F. Baccelli, G. Cohen, G. J. Olsder, J. P. Quadrat, *Synchronization and Linearit*y, Prentice Hall, 1993.

[Back78]

J. Backus, “Can Programming be Liberated from the Von-Neumann style? A functional style and its algebra of programs,” *Communications of the ACM,* **Vol.21, No.8**, August, 1978.

[Bamb90]

R. H. Bamberger, “The Directional Filterbank: a Multirate Filterbank for the Directional Decomposition of Images”, Ph.D. Thesis, Georgia Institute of Technology, November 1990.

[Bane88]

U. Banerjee, *Dependence Analysis for Supercomputing*, Kluwer Academic Publishers, 1988.

[Barr91]

B. Barrera and E. A. Lee, “Multirate Signal Processing in Comdisco’s SPW,” *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Toronto, April, 1991.

[Beec95]

M. Von der Beeck, “A Comparison of Statecharts Variants,” *Formal Techniques in Real-Time and Fault-Tolerant Systems, Third International Symposium Proceedings*, Germany, September 1994.

[Benv91]

A. Benveniste and G. Berry, “The Synchronous Approach to Reactive and Real-Time Systems,” *Proceedings of the IEEE,* **Vol.79, No. 9**, September 1991.

[Berr85]

G. Berry and P-L Curien, “Theory and Practice of Sequential Algorithms: the Kernel of the Programming Language CDS,” Algebraic Methods in Semantics, Cambridge University Press, pp35-88, 1988.

[Bhat93]

S. Bhattacharyya and E. A. Lee, “Scheduling synchronous dataflow graphs for efficient looping", *Journal of VLSI Signal Processing*, **Vol. 6**, **No. 3**, December, 1993.

[Bhat94a]

S. S. Bhattacharyya and E. A. Lee, "Memory management for dataflow programming of multirate signal processing algorithms", *IEEE Transactions on Signal Processing*, **Vol. 42, No. 5**, May, 1994.

[Bhat94b]

S. S. Bhattacharyya, *Compiling Dataflow Graphs for Signal Processing*, Ph.D. thesis, Memorandum No. UCB/ERL M94/52, Electronics Research Laboratory, University of California at Berkeley, July, 1994.

[Bhat95]

S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, "Optimal Parenthesization of Lexical Orderings for DSP Block Diagrams," *IEEE Workshop on VLSI Signal Processing*, Osaka, Japan, October 1995.

[Bhat96a]

S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, *Software Synthesis from Dataflow Graphs*, Kluwer Academic Publishers, Norwood, Massachusetts, 1996.

[Bhat96b]

S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, "APGAN and RPMC: Complimentary Heuristics for Translating DSP Block Diagrams into Efficient Software Implementations", **to appear** in the *Design Automation for Embedded Systems Journal*, 1997.

[Bier95]

J. C. Bier, P. D. Lapsley, and E. A. Lee, *Design Tools and Methodologies for DSP Systems — Volumes 1 & 2: DSP Design Challenges, Methodologies, and Tools*, Berkeley Design Technologies, Inc., Fremont, California, 1995.

[Bosv92]

F. Bosveld, R. L. Lagendijk, J. Biemond, "Compatible Spatio-Temporal Subband

Encoding of HDTV", *Signal Processing*, **Vol. 28, No. 3**, September 1992.

[Buck91]

J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Multirate Signal Processing In Ptolemy," *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Toronto, April, 1991.

[Buck93]

J. T. Buck, *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*, Ph.D. thesis, Memorandum No. UCB/ERL M93/69, Electronics Research Laboratory, University of California at Berkeley, September, 1993.

[Buck94]

J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems," *International Journal of Computer Simulation*, **Vol. 4**, April, 1994.

[Cass93]

C. Cassandras, *Discrete Event Systems*, Irwin, 1993.

[Chan96]

W. -T. Chang, S. Ha, and E. A. Lee, "Heterogenous Simulation -- Mixing Discrete Event Models with Dataflow," **to appear**, invited paper in the RASSP special issue of the *Journal on VLSI Signal Processing*, 1996.

[Chen94]

M. C. Chen, "Developing a Multidimensional Synchronous Dataflow Domain in Ptolemy", MS Report, UC Berkeley, June 1994.

[Corm90]

T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, McGraw-Hill, 1990.

[Cove91]

T. Cover, J. Thomas, *Elements of Information Theory*, Wiley, 1991.

[Covi87]

C. D. Covington, G. E. Carter, and D. W. Summers, “Graphic Oriented Signal Processing Language — GOSPL,” *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Dallas, April, 1987.

[Cubr93]

M. Cubric and P. Panangaden, “Minimal Memory Schedules for Dataflow Networks,” *CONCUR ‘93*, Hildesheim, Germany, August, 1993.

[Deme94]

A. Demeure, “Formalisme de Traitement du Signal: Array-OL,” Technical report, Thomson SINTRA ASM, Sophia Antipolis, Valbonne, 1994.

[Denk96]

T. C. Denk, M. Majumdar, and K. K. Parhi, “Two-Dimensional Retiming with Low Memory Requirements,” *Proceedings of the ICASSP ’96*, Atlanta, Ga, May 1996.

[Denn75]

J. B. Dennis, *First Version of a Data Flow Procedure Language*, MAC Technical Memorandum 61, Laboratory for Computer Science, Massachusetts Institute of Technology, May, 1975.

[Denn80]

J. B. Dennis, “Dataflow Supercomputers,” *IEEE Computer Magazine*, **Vol. 13**, **No. 11**, November 1980.

[Desm93]

D. Desmet and D. Genin, “ASSYNT: Efficient Assembly Code Generation for DSPs Starting from a Data Flowgraph,” *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Minneapolis, April, 1993.

[Dubo85]

E. Dubois, “The Sampling and Reconstruction of Time-varying Imagery with Applications in Video Systems”, *Proceedings of the IEEE*, **Vol. 73**, April 1985.

[Dudg84]

D. E. Dudgeon, R. M. Mersereau, *Multidimensional Digital Signal Processing*, Prentice Hall, 1984.

[Epst89]

R. L. Epstein, W. A. Carnielli, *Computability Computable Functions, Logic, and the Foundation of Mathematics*, Wadsworth & Brooks/Cole, 1989.

[Gao92]

G. R. Gao, R. Govindarajan, and P. Panangaden, “Well-Behaved Programs for DSP Computation,” *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, San Francisco, March, 1992.

[Gare79]

M. R. Garey and D. S. Johnson, *Computers and Intractability*, W. H. Freeman and Co., NY, 1979.

[Geni89]

D. Genin, J. De Moortel, D. Desmet, and E. Van de Velde, “System Design, Optimization, and Intelligent Code Generation for Standard Digital Signal Processors,” *Proceedings of the International Symposium on Circuits and Systems*, Portland, Oregon, May, 1989.

[Henz96]

T. A. Henzinger, “The theory of hybrid automata,” *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, New Brunswick, NJ, USA, July 1996.

[Ho88a]

W. H. Ho, *Code Generation for Digital Signal Processors Using Synchronous Dataflow*, Master’s project report, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, May, 1988.

[Ho88b]

W. H. Ho, E. A. Lee, and D. G. Messerschmitt, “High Level Dataflow Programming for Digital Signal Processing,” *VLSI Signal Processing III*, IEEE Press, 1988.

[Hopk93]

R. Hopkins, “Progress on HDTV Broadcasting Standards in the United States”, Signal Processing: *Image Communication*, **Vol. 5**, December 1993.

[How90]

S. How, *Code Generation for Multirate DSP Systems in Gabriel*, Master’s project report, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, May, 1990.

[Huda89]

P. Hudak, “Conception, evolution, and application of functional programming languages” *Computing Surveys*, **Vol. 21, No. 3**, September, 1989.

[Iver62]

K. E. Iverson, *A Programming Language*, Wiley, 1962.

[Kahn74]

G. Kahn, “The Semantics of a Simple Language for Parallel Programming,” *Proceedings of the IFIP Congress 74*, North-Holland Publishing Co., 1974.

[Kahn93]

G. Kahn, G. D. Plotkin, “Concrete Domains,” *Theoretical Computer Science,* **Vol. 121, No. 1-2**, December, 1993.

[Kala93]

A. Kalavade, and E. A. Lee, “A Hardware/Software Codesign Methodology for DSP Applications,” *IEEE Design and Test,* **Vol. 10, No. 3**, September 1993.

[Kapl87]

D. J. Kaplan, *et al.*, *Processing Graph Method Specification Version 1.0*, Unpublished memorandum, Naval Research Laboratory, Washington D.C, December, 1987.

[Karj88]

M. Karjalainen and S Helle, “Block Diagram Compilation and Graphical Editing of DSP Algorithms in the QuickSig System”, *Proceedings of the International Symposium on Circuits and Systems*, Espoo, Finland, June, 1988.

[Karl90]

G. Karlsson, M. Vetterli, “Theory of Two Dimensional Multirate Filter Banks”, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, **Vol. ASSP-38**, June 1990.

[Karp66]

R. M. Karp and R. E. Miller, “Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing,” *SIAM Journal of Applied Mathematics*, **Vol. 14**, **No. 6**, November, 1966.

[Karp69]

R. M. Karp, and R. E. Miller, “Parallel Program Schemata,” *Journal of Computer and System Sciences*, **Vol. 3**, 1969.

[Karp72]

R. M. Karp, “Reducibility Among Combinatorial Problems,” Symposium on the Complexity of Computer Computations, Raymond E. Miller and James W. Thatcher editors, Plenum Press, 1972.

[Kell61]

J. Kelly, Lochbaum, and V. Vyssotsky, “A Block Diagram Compiler,” *Bell System Technical Journal*, **Vol. 40**, **No. 3**, May, 1961.

[Kern70]

B. W. Kernighan and S. Lin, “An Efficient Heuristic Procedure for Partitioning Graphs,” *Bell System Technical Journal*, **Vol. 49**, **No.2**, February 1970.

[Kons94]

K. Konstantinides and J. R. Rasure, “The Khoros Software Development Environment for Image and Signal Processing,” *IEEE Transactions on Image Processing*, **Vol. 3, No. 3**, May, 1994.

[Kung88]

S. Y. Kung, *VLSI Array Processors*, Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[Lauw90]

R. Lauwereins, M. Engels, J.A. Peperstraete, E. Steegmans, and J. Van Ginderdeuren, “GRAPE: A CASE Tool for Digital Signal Parallel Processing,” *IEEE ASSP Magazine*, **Vol. 7**, **No. 2**, April, 1990.

[Lauw94]

R. Lauwereins, P. Wauters, M. Ade, and J. A. Peperstraete, “Geometric Parallelism and Cyclo-Static Data Flow in GRAPE-II,” *IEEE Workshop on Rapid System Prototyping*, Grenoble, June, 1994.

[Lear90]

K. W. Leary and W. Waddington, “DSP/C: A Standard High Level Language for DSP and Numeric Processing,” *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Albuquerque, April, 1990.

[Lee86]

E. A. Lee, *A Coupled Hardware and Software Architecture for Programmable Digital Signal Processors*, Ph.D. thesis, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, May, 1986.

[Lee87]

E. A. Lee and D. G. Messerschmitt, “Static Scheduling of Synchronous Dataflow Programs for Digital Signal Processing,” *IEEE Transactions on Computers*, **Vol. C-36, No. 2**, February, 1987.

[Lee89]

E. A. Lee, W. H. Ho, E. Goei, J. Bier, and S. Bhattacharyya, "Gabriel: A Design Environment for DSP," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, **Vol. 37**, **No. 11**, November, 1989.

[Lee91]

E. A. Lee, "Consistency in Dataflow Graphs," *IEEE Transactions on Parallel and Distributed Systems*, **Vol. 2**, **No. 2**, April, 1991.

[Lee93]

E. A. Lee, "Multidimensional Streams Rooted in Dataflow," *Proceedings of the IFIP Working Conference on Architectures and Compilation Techniques for Fine and Medium Grained Parallelism*, Orlando, January, 1993.

[Lee95]

E. A. Lee and T. M. Parks, "Dataflow Process Networks," *Proceedings of the IEEE*, **Vol. 83, No. 5**, May, 1995.

[Lee96]

E. A. Lee, and A. Sangiovanni-Vincentelli, "The Tagged Signal Model —A Preliminary Version of a Denotational Framework for Comparing Models of Computation," UCB/ERL Memo. M96/33, Electronics Research Laboratory, UC Berkeley, Ca, June 1996.

[Lee96b]

E. A. Lee, *EE290N—Specification and Modeling of Reactive Real-Time Systems*, class notes, Dept. of EECS, University of California, Berkeley, http://ptolemy.eecs.berkeley.edu/~eal/ee290n/notes.html, Fall semester, 1996.

[Leis91]

C. E. Leiserson, J. B. Saxe, "Retiming Synchronous Circuitry, " *Algorithmica*, **Vol.6, No.1**, 1991.

[Liao95]

S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang, “Code Optimization Techniques for Embedded DSP Microprocessors,” *Proceedings of the 32nd Design Automation Conference*, June, 1995.

[Mand93]

R. Manduchi, G. M. Cortelazzo, and G. A. Mian, “Multistage Sampling Structure Conversion of Video Signals”, *IEEE Transactions on Circuits and Systems for Video Technology,* **Vol. 3, No. 5**, October 1993.

[McGr83]

J. McGraw, S. Skedzielewski, S. Allan, D. Grit, R. Oldehoeft, J. Glauert, I. Dobes, and P. Hohensee, *SISAL: Streams and Iteration in a Single Assignment Language: Language Reference Manual Version 1.1*, Lawrence Livermore Laboratory, July, 1983.

[Mers83]

R. M. Mersereau, T. C. Speake, “The Processing of periodically sampled Multidimensional Signals”, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, **Vol. ASSP-31**, Feb 1983.

[Mess84]

D. G. Messerschmitt, “Structured Interconnection of Signal Processing Programs,” *Proceedings of Globecom*, Atlanta, 1984.

[Mull91]

Arrays, Functional Languages, and Parallel Systems, editted by L. M. R. Mullin, M. Jenkins, G. Hains, R. Bernecky, G. Gao, Kluwer Academic Publishers, 1991.

[Murt93]

P. K. Murthy, *Multiprocessor DSP Code Synthesis in Ptolemy*, Master’s project report, Memorandum No. UCB/ERL M93/66, Electronics Research Laboratory, University of California at Berkeley, August, 1993.

[Murt94a]

P. K. Murthy, S. S. Bhattacharyya, and E. A. Lee, “Minimizing Memory Requirements for Chain Structured Synchronous Dataflow Graphs”, *Proceedings of the ICASSP ‘94*, Adelaide, Australia, April, 1994.

[Murt94b]

P. K. Murthy and E. A. Lee, “On the Optimal Blocking Factor for Blocked, Non-Overlapped Schedules”, ERL Memo No. UCB/ERL M94/46, Electronics Research Lab, UC Berkeley, Ca 94720, June 1994; condensed version in *Proceedings of the 28th Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, CA, USA, November, 1994.

[Murt94c]

P. K. Murthy, S. S. Bhattacharyya, and E. A. Lee, “Combined Code and Data Minimization for Synchronous Dataflow Programs”, ERL Memo No. UCB/ERL M94/93, Electronics Research Lab, November 1994, UC Berkeley, CA 94720, and **to appear** in the *Formal Methods in System Design* Journal, 1997.

[Murt95]

P. K. Murthy, E. A. Lee, “A Generalization of Multidimensional Synchronous Dataflow to Arbitrary Sampling Lattices”, *Proceedings of the ICASSP ‘96*, Atlanta, USA, May, 1996.

[Naye93]

K. Nayebi, T. P. Barnwell, and M. J. T. Smith, “Nonuniform Filter Banks: A Reconstruction and Design Theory,’’ *IEEE Transactions on Signal Processing*, **Vol. 41, No. 3**, March, 1993.

[Nemh88]

G. L. Nemhauser, L. A. Woolsey, *Integer and Combinatorial Optimization*, Wiley, 1988.

[Ohal91]

D. R. O’Hallaron, *The Assign Parallel Program Generator*, Memorandum CMU-

CS-91-141, School of Computer Science, Carnegie Mellon University, May, 1991.

[Olso92]

T. J. Olson, N. G. Klop, M. R. Hyett, and S. M. Carnell, “MAVIS: a Visual Environment for Active Computer Vision,” *Proceedings of the 1992 IEEE Workshop on Visual Languages*, Seattle, September, 1992.

[Parh91]

K. K. Parhi, D. G. Messerschmitt, “Static Rate-Optimal Scheduling of Iterative Data-Flow Programs via Optimum Unfolding,” *IEEE Transactions on Computers*, February, 1991.

[Pass94]

N. Passos, E. H. -M. Sha, and S. Bass, “Schedule-based Multidimensional Retiming on Data Flow Graphs,” *Proceedings of the 8th International Parallel Processing Symposium*, April 1994.

[Petr81]

J. L. Peterson, *Petri Net Theory and the Modeling of Systems*, Prentice Hall, 1981.

[Pino95a]

J. Pino, S. Ha, E. A. Lee, and J. T. Buck, “Software Synthesis for DSP Using Ptolemy,” invited paper in *Journal of VLSI Signal Processing*, January, 1995.

[Pino95b]

J. Pino, S. S. Bhattacharyya, and E. A. Lee, “A Hierarchical Multiprocessor Scheduling System for DSP Applications,’’ *Proc. IEEE Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, CA, November, 1995.

[Powe92]

D. B. Powell, E. A. Lee, and W. C. Newman, “Direct Synthesis of Optimized DSP Assembly Code from Signal Flow Block Diagrams,” *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, San Francisco, March, 1992.

[Rams84]

T. A. Ramstad, “Digital Methods for Conversion between Arbitrary Sampling Frequencies”, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, **Vol. ASSP-32**, June 1984.

[Reek92]

H. J. Reekie, “Integrating Block-Diagram and Textual Programming for Parallel DSP,” *Proceedings of the 3d International Symposium on Signal Processing and its Applications*, Queensland, Australia, August, 1992.

[Reit68]

R. Reiter, “Scheduling Parallel Computations,” *Journal of the ACM*, **Vol. 15**, **No. 4**, October, 1968.

[Ritz92]

S. Ritz, M. Pankert, and H. Meyr, “High Level Software Synthesis for Signal Processing Systems,” *Proceedings of the International Conference on Application Specific Array Processors*, Berkeley, August, 1992.

[Ritz93]

S. Ritz, M. Pankert, and H. Meyr, “Optimum Vectorization of Scalable Synchronous Dataflow Graphs,” *Proceedings of the International Conference on Application-Specific Array Processors*, Venice, October, 1993.

[Ritz95]

S. Ritz, M. Willems, H. Meyr, “Scheduling for Optimum Data Memory Compaction in Block Diagram Oriented Software Synthesis,” *Proceedings of the ICASSP 95*, Detroit, Michigan, May 1995.

[Seth75]

R. Sethi, “Complete Register Allocation Problems,” *SIAM Journal on Computing*, **Vol. 4, No. 3**, September, 1975.

[Shan87]

K. S. Shanmugan, G. J. Minden, E. Komp, T. C. Manning, and E. R. Wiswell, *Block-Oriented System Simulator (BOSS)*, Telecommunications Laboratory, University of Kansas, Internal Memorandum, 1987.

[Sih91]

G. C. Sih, *Multiprocessor Scheduling to Account for Interprocessor Communication*, Ph.D. thesis, Memorandum No. UCB/ERL M91/29, Electronics Research Laboratory, University of California at Berkeley, April, 1991.

[Stoy77]

J. Stoy, *Denotational Semantics : the Scott-Strachey Approach to Programming Language Theory*, MIT Press, 1977.

[Tow88]

J. Tow, S. L. Gay, and J. Hartung, “Implementation of DSP Applications Using the AT&T DSP32C Compiler and Application Library,” *Proceedings of the International Symposium on Circuits and Systems*, Espoo, Finland, June, 1988.

[Vaid90]

P. P. Vaidyanathan, “Fundamentals of Multidimensional Multirate Digital Signal Processing,” *Sadhana*, vol. 15, pp. 157-176, Nov. 1990.

[Vaid93]

P. P. Vaidyanathan, *Multirate Systems and Filter Banks*, Prentice Hall, 1993.

[Veig90]

M. Veiga, J. Parera, and J. Santos, “Programming DSP Systems on Multiprocessor Architectures,” *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Albuquerque, April, 1990.

[Verk96]

D. Verkest, K. Van Rompaey, I. Bolsens, H. De Man, “POPE — A Design Environment for Heterogeneous Hardware/Software Systems,” to appear, *Design Automation for Embedded Systems*, 1996.

[Visc91]

E. Viscito, J. P. Allebach, “The Analysis and Design of Multidimensional FIR Perfect Reconstruction Filterbanks for Arbitrary Sampling Lattices”, *IEEE Transactions on Circuits and Systems*, **Vol. CAS-38**, January 1991.

[Watl95]

J. A. Watlington, V. M. Bove Jr., “Stream-based Computing and Future Television”, Proceedings of the 137th SMPTE Technical conference, September 1995.

[Yates93]

R. K. Yates, “Networks of Real-Time Processes,” in Concur ‘93, *Proc. of the 4th Int. Conf. on Concurrency Theory,* E. Best, ed., Springer-Verlag LNCS 715, 1993.

[Yu93]

K. H. Yu and Y. H. Hu, “Optimized Code Generation for Programmable Digital Signal Processors,” *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Minneapolis, April, 1993

[Zima90]

H.Zima and B.Chapman, *Supercompilers for Parallel and Vector Computers*, ACM Press, 1990.

[Ziss87]

M. A. Zissman, G. C. O’Leary, and D. H. Johnson, “A Block Diagram Compiler for a Digital Signal Processing MIMD Computer,” *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Dallas, April, 1987.

[Zivo95]

V. Zivojnovic, H. Schraut, M. Willems, and R. Schoenen, “DSPs, GPPs, and Multimedia Applications — An Evaluation Using DSPStone,” *Proceedings of ICSPAT*, November, 1995.