S. S. Bhattacharyya. *Compiling Dataflow Programs for Digital Signal Processing*. PhD thesis, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, July 1994

Abstract

COMPILING DATAFLOW PROGRAMS FOR DIGITAL SIGNAL PROCESSING

by

Shuvra Shikhar Bhattacharyya

Doctor of Philosophy in Electrical Engineering

Professor Edward A. Lee, Chair

The synchronous dataflow (SDF) model has proven efficient for representing an important class of digital signal processing algorithms. The main property of this model is that the number of data values produced and consumed by each computation is fixed and known at compile-time. This thesis develops techniques to compile SDF-based graphical programs for embedded signal processing applications into efficient uniprocessor implementations on microprocessors or programmable digital signal processors. The main problems that we address are the minimization of code size and the minimization of the execution time and storage cost required to buffer intermediate results.

The minimization of code size is an important problem since only limited amounts of memory are feasible under the speed and cost constraints of typical embedded system applications. We develop a class of scheduling algorithms that minimize code space requirements without sacrificing the efficiency of inline code. This is achieved through the careful organization of loops in the target pro-

gram. Our scheduling framework provably synthesizes the most compact looping structures for a certain class of SDF graphs, and from our preliminary observations this class appears to subsume most practical SDF graphs. Also, by modularizing different components of the scheduling framework and establishing their independence, we demonstrate how two additional scheduling objectives — decreasing the memory required for data buffering and increasing the amount of buffering that occurs through registers — can be incorporated in a manner that does not conflict with the goal of code size compactness. We carry out these additional optimization objectives through graph clustering techniques that avoid deadlock and that fully preserve the compact loop structures offered by the original graph.

We also present compile-time techniques for improving the efficiency of buffering for a given uniprocessor schedule. The optimizations include dataflow analysis techniques to statically determine buffer addressing patterns; examination of the loop structures in a schedule to provide flexibility for overlaying buffer memory; and techniques to optimize the management of circular buffers, which are useful for implementing dataflow links that have delay and for reducing memory requirements.

Edward A. Lee, Thesis Committee Chairman

Table of Contents

S. S. Bhattacharyya. *Compiling Dataflow Programs for Digital Signal Processing*. PhD thesis, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, July 1994

1

INTRODUCTION

Algorithms for digital signal processing (DSP) are often most naturally described by block diagrams in which computational blocks are interconnected by links that represent sequences of data values. Due to the emergence of low cost workstations and personal computing systems with graphics capabilities, it has become feasible for designers of signal processing systems to acquire graphical block diagram programming environments, and as a result, there has been a proliferation of such programming environments in recent years, both from industrial sources and from research and educational institutions.

The synchronous dataflow (SDF) model, whose fundamental theories were developed by Karp and Miller in [Karp66] and by Lee and Messerschmitt [Lee87], has proven efficient for representing an important class of digital signal processing algorithms, and has been used as the basis for numerous DSP programming environments, such as those described in [Lauw90, Lee89, Ohal91, Prin92, Ritz92, Veig90]. The main property of the SDF model is that the number of data values produced and consumed by each functional component is fixed and known at com-

pile time. This thesis develops techniques for compiling block diagram programs based on the SDF model into efficient object code for microprocessors and programmable digital signal processors, which are specialized microprocessors for DSP applications [Lee88b].

Block diagram programming of DSP systems dates back at least to the early 1960s, when a group at Bell Telephone Laboratories developed a block diagram compiler for simulating signal processing systems developed for visual and acoustic research [Kell61]. In [Covi87], Covington presents a graphical programming environment for designing digital filters based on only two types of computational blocks — adders and constant gains. At Advanced Micro Devices Corporation, a graphical tool was developed for mapping signal processing algorithms onto a two dimensional array of programmable digital signal processors [Ziss87]. Similarly, at Carnegie-Mellon University, a hierarchical block diagram format was used to represent signal processing algorithms for compilation onto the iWarp multicomputer [Ohal91]. Currently, several graphical programming environments for DSP are also available commercially, such as the Signal Processing Worksystem, developed by Comdisco Systems, which is now the Alta Group of Cadence Design Systems [Barr91]; COSSAP, developed by Cadis and by Heinrik Meyer's group at the Aachen University of Technology [Ritz92]; and the DSP Station, developed by Mentor Graphics. See [Lee89] for a large number of additional references to graphical programming and simulation environments for DSP.

At the University of California at Berkeley, there has been a large effort in developing efficient and elaborate graphical design environments. This work is rooted in the BLOSIM simulation system developed by Messerschmitt [Mess84]. Further exploration with BLOSIM inspired the development of the SDF model [Lee87]; soon afterwards, Ho developed the first compiler for pure SDF semantics

[Ho88b], targetted to the Motorola 56000 programmable digital signal processor, and this compiler formed the foundation for the Gabriel design environment [Lee89]. The successor to BLOSIM and Gabriel is the Ptolemy project [Buck92], an object-oriented framework for simulation, prototyping, and software synthesis of heterogeneous systems. Unlike Gabriel, which is based on a single model of computation — the SDF model, Ptolemy allows a system to consist of multiple subsystems that are specified with different models of computation, and Ptolemy allows the user to define new models of computation and to interface a newly-defined model with the existing models. For example, dynamic dataflow, discrete-event, and communicating processes, are some of the models of computation that are supported by Ptolemy in addition to SDF. The Ptolemy framework together with a block diagram programming interface have been used to develop DSP simulation capabilities [Buck91], as well as compilers for the Motorola 56000 [Pino94] and the Sproc microprocessor, developed by Star Semiconductor Corporation [Murt93].

As mentioned above, a primary advantage of graphical programming environments for DSP is that DSP algorithms are often most naturally represented as hierarchies of block diagrams. Two additional advantages are the support for software reuse (*modularity*) and the support for efficient compilation. 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, usually it is 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 development effort. For example, since each function is defined only once, for frequently used functions it becomes economical to spend a large effort to hand-optimize the function definition for efficiency.

An alternative means of attaining modularity that has been explored in DSP design environments is the use of libraries of subroutines that can be called from high level language programs [Egol93, Tow88]. Here, once the library is in place, the programmer has the convenience of programming in a high level language, such as C or FORTRAN, while exploiting the efficiency of hand-optimized functions written in assembly language.

There have been widespread reports on the inability of high-level language compilers to deliver satisfactory code for time-critical DSP applications [Geni89, Tow88, Yu93]. The throughput requirements of such applications are often extremely severe, and designers typically must 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. The use of optimized subroutine libraries, as described above, 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]. 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 on this method show promise.

The alternative that we pursue in this thesis is the use of graphical or textual block diagram languages based on the SDF model in conjunction with hand-optimized block libraries. As we will discuss precisely in Chapter 2, the SDF

model allows us to schedule all of the computations at compile-time and thus eliminates 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, and 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]. 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.

In this thesis, we develop techniques for compiling general SDF programs for multirate DSP systems into efficient uniprocessor implementations. 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. For example, the Motorola DSP56001 has an on-chip capacity of 512 instruction and 512 data words, and Star Semiconductor's SPROC can store 1k instructions and 1k data. In the Motorola DSP56001, one on-chip instruction and two on-chip data words can be accessed in

parallel, while there is only one external memory interface. Thus, there is a speed penalty for accessing off-chip memory regardless of how fast the external memory is. Moreover, off-chip memory typically needs to be static, increasing the system cost considerably. In this thesis, we develop techniques to minimize the code size when compiling an SDF program, and we combine these techniques with techniques for minimizing the amount of memory required to buffer data between computational blocks.

As we will discuss in the sequel, large sample rate changes result in an explosion of code size requirements if naive compilation techniques are used. In this thesis, we develop a class of scheduling algorithms that minimizes code space requirements through the careful organization of loops in the target code. This scheduling framework provably synthesizes the most compact looping structures for a certain class of SDF graphs, and from our preliminary observations, this class appears to subsume most practical SDF graphs. Also, by modularizing different components of the scheduling framework and establishing their independence, we show that other scheduling objectives can be incorporated in a manner that does not conflict with the goal of code compactness, and we demonstrate this for two specific additional objectives — decreasing the amount of memory required for data storage and increasing the amount of data transfers that occur through registers rather than through memory. Finally, we present techniques to improve the efficiency of data buffering between the computational blocks in an SDF program.

It should be noted that there have been significant efforts to improve the efficiency of code generated from high level language programs of DSP applications, such as those described in [Hart88, Kafk90, Yu93], and the success of these efforts indicates that the range of applications that are adequately supported by high level language compilers is increasing. However we emphasize that the effi-

ciency of the compiled code is not the only advantage of block diagram programming and the SDF model — block diagram environments often provide the most natural specification format for signal processing algorithms, and they promote the recycling of software, expertise and development effort. All of these advantages motivate the solutions developed in this thesis.

1.1 Dataflow

The principles of dataflow and their application to the development of computer architectures and programming languages were pioneered by Dennis [Denn75]. A central objective of the dataflow concept is to facilitate the exploitation of parallelism from a program. In dataflow, a program is represented as a directed graph, called a *dataflow graph*, in which the vertices, called *actors*, represent computations and the edges represent FIFO channels that queue data values, encapsulated in objects called *tokens*, as they are passed from the output of one computation to the input of another. A key requirement of the computation corresponding to a dataflow actor is that it be *functional*; that is, each output value of an invocation of the computation is determined uniquely by the input values to that invocation.

A dataflow representation of a computation differs fundamentally from a corresponding representation in a procedural language such as C or FORTRAN in that it specifies the function being computed rather than specifying a step-by-step procedure to compute it. This distinction between *definitional* approaches to programming, such as dataflow, and *operational* approaches, such as C or FORTRAN is explored in depth in [Ambl92]. A major disadvantage of operational approaches is that they leave the programmer responsible for a difficult task, namely ordering

the computations, that is often critical to the speed and memory requirements of the target implementation. Of course, the compiler can attempt to deduce the dependencies between computations from an operational specification and then reorder the computations in a more efficient way, but this endeavor is often made extremely difficult or impossible by side effects, aliasing, or unstructured control-flow. Functional languages, such as pure Lisp and Haskell, are exceptions. In these languages, in which computations are specified through compositions of functions, programs can, in principle, be easily converted into equivalent dataflow representations [Acke82]. In [Lee94], Lee explores several more subtle relationships between functional languages and dataflow-based graphical programming frameworks.

Dennis applied the concepts of dataflow to pioneer a form of computer architecture; computers that are based on this form of architecture are called *dataflow computers*. Unlike conventional von Neumann computers in which the execution of instructions is controlled by a program counter, computations in a dataflow computer are driven by the availability of data. This is achieved by maintaining, at the machine level, a representation of the program as a dataflow graph, and by providing capabilities in hardware to detect which actors have sufficient data to fire, to execute the corresponding instructions and to route the output values to the appropriate actor inputs.

There are two basic types of dataflow computers — *static* dataflow computers and *dynamic*, or *tagged-token*, dataflow computers. The original dataflow computer architecture, the MIT Static Dataflow Architecture [Denn80], was of the static variety. In a static dataflow computer, at most one data value can be queued on an edge at one time. This restriction allows the storage for the edges to be allocated at compile-time, and it is enforced by adding feedback edges, called

acknowledgment arcs, directed between the sink and source actors of the edges in the original dataflow graph. In the MIT Static Dataflow Computer, the dataflow graph is maintained at the machine level as a collection of *activity templates*, which correspond to actor invocations. Each activity template consists of an opcode that specifies the associated machine instruction, locations to hold the operands, and pointers to the appropriate operand slots of the activity templates that must receive the output value. Each time an instruction is executed, each activity template referenced by the associated destination address pointers is updated by the *Update Unit* to contain the new output value in the appropriate operand slot. For each activity template that it modifies, the Update Unit checks whether that last vacant operand slot has been filled, and if so, it forwards a reference to the activity template to the *Instruction Queue*. Entries in this queue are processed by the *Fetch Unit*, which looks up each corresponding activity template in the activity store, sends an operation packet to the *Execution Unit*, and resets the activity template.

Since the rate at which instructions are executed is limited mainly by the rate at which the Execution Unit performs computations and by the rate at which the Instruction Queue is filled, which in turn depends on the matching of operand values to activity templates, the problems that arise in conventional von-Neumann processors due to memory latencies and synchronization are mitigated. Rather than handling interprocessor synchronization and processor-memory synchronization by wasteful idle-waiting or by expensive context switches, data dependencies are enforced by the hardware for each individual instruction, and independent operations are automatically detected and exploited.

A major shortcoming of the static dataflow computer arises from the restriction that only one data value can be queued on an edge at a given time,

which implies that multiple invocations of a given actor cannot be executed in parallel. This severely limits the parallelism that can be exploited from loops and precludes executing multiple invocations of a subroutine in parallel. To overcome this shortcoming, Arvind and Nikhil at MIT [Arvi90], and Gurd et al. at Manchester University [Gurd85] independently developed and explored the *tagged-token* concept, which permits an arbitrary number of invocations of the same actor to execute concurrently. In a tagged-token dataflow computer, an identifying tag is carried around with each token. This tag designates the subroutine invocation number, loop iteration number, and the instruction number. For example, in the MIT Tagged-Token Dataflow Machine, the *Waiting-Matching Unit* removes unprocessed tokens buffered in a *Token Queue*, and compares the tag of each token it removes with the tags of all tokens that are in the Waiting-Matching unit at that time. If a matching tag is not found, then the token is stored in the Waiting-Matching unit until a matching token arrives. Otherwise the matching token pair is forwarded to the *Instruction-Fetch Unit*, which accesses program memory to determine the appropriate machine instruction and constructs an operation packet consisting of the instruction and its operands. This operation packet is forwarded to the *ALU*, and simultaneously the operation is executed and the tag for the result token is computed. The result token and its tag are then combined and entered in the *Token Queue*.

Although dataflow computers succeed in attacking the problems of synchronization and memory latency, challenges remain in coping with the resource requirements of unpredictable and unbounded amounts of parallelism, and in amortizing the overhead incurred on sequential code. These issues continue to be an active research area; for example, see [Arvi91]. However dataflow computer technology has not yet matured to the point of being commercially advantageous,

and thus there are no commercially available dataflow computers to this date, although some commercially available processors have incorporated dataflow concepts to a limited degree [Chas84, Schm91].

In this thesis, we do not apply dataflow computers; instead, we apply the concepts of dataflow as they relate to program representation. Another aspect in which our use of dataflow differs from dataflow computers is in the complexity of the actors — we apply a *mixed grain* dataflow model, meaning that actors can represent operations of arbitrary complexity, whereas dataflow computers operate on *fine grain*, or *atomic*, dataflow graphs, where the complexity of the actors is at the level of individual machine instructions. In the SDF-based design environments to which this thesis applies, dataflow actors typically range in complexity from basic operations such as addition or subtraction to signal processing subsystems such as FFT units and adaptive filters. Finally, our use of dataflow is limited by the granularity of each actor: we use dataflow to describe the interaction between actors, but the functionality of each actor can be specified in any programming language, such as C, as in [Ritz92]; LISP, as in [Karj88]; or a LISP/assembly language hybrid as in [Lee89], where a high level language is used to customize assembly language code blocks according to compile-time parameters.

1.2 Synchronous Dataflow

Synchronous dataflow is a restricted version of dataflow in which the number of tokens produced (consumed) by an actor on each output (input) edge is a fixed number that is known at compile time. Each edge in an SDF graph also has a non-negative integer delay associated with it, which corresponds to the number of initial tokens on the edge. The application of the SDF model to mixed-grain data-

flow programming of multirate DSP systems was pioneered by Lee and Messerschmitt in the mid 1980s [Lee87]. In this section, we informally outline important theoretical developments on the SDF model and their application to block diagram programming of DSP algorithms. These principles will be reviewed rigorously early in Chapter 2, and they will form much of the theoretical basis for the remainder of the thesis.

Important foundations for the SDF model were laid by the definition and exploration of *computation graphs* by Karp and Miller roughly two decades before the development of SDF [Karp66]. The computation graph model is equivalent to SDF graphs, except that in addition to production and consumption parameters, an additional *threshold* parameter is associated with each edge. This threshold parameter, which must be greater than or equal to the corresponding consumption parameter, determines the minimum number of tokens that must be queued on the edge before the sink actor can be fired. Thus, an SDF graph is a computation graph in which the threshold parameter of each edge equals the number of tokens consumed from the edge per sink invocation.

Karp and Miller established that computation graphs are *determinate*, which means that each computation graph uniquely determines the sequence of data values produced on the edges in the graph; these sequences do not depend on the *schedule* of actor executions — that is, on the order in which the actors are invoked. Also, they developed topological and algebraic conditions to determine which subgraphs in a computation graph become deadlocked. For the problems that computation graphs were designed to represent, only graphs that terminate — that is, reach a deadlocked state — are correct, and thus, the results of Karp and Miller do not lead to solutions for constructing efficient infinite schedules, although the underlying concept of determinacy applies both to infinite and finite

schedules.

However, in DSP applications, we are often concerned with operations that are applied repeatedly to samples in an indefinitely long sequence of input data, and thus when applying a dataflow representation, it is mandatory that we support infinite sequences of actor executions. For example, consider the block diagram program shown in Figure 1.1, which is taken from a snapshot of a session with the Ptolemy system [Buck92]. This program specifies a sample rate conversion system developed by Thomas Parks, a graduate student at U. C. Berkeley, to interface a digital audio tape (DAT) player to a compact disc (CD) player. The sample rates of CD players and DAT players are, respectively, 44.1kHz and 48kHz, and the system in Figure 1.1 shows a multistage implementation of the conversion between these

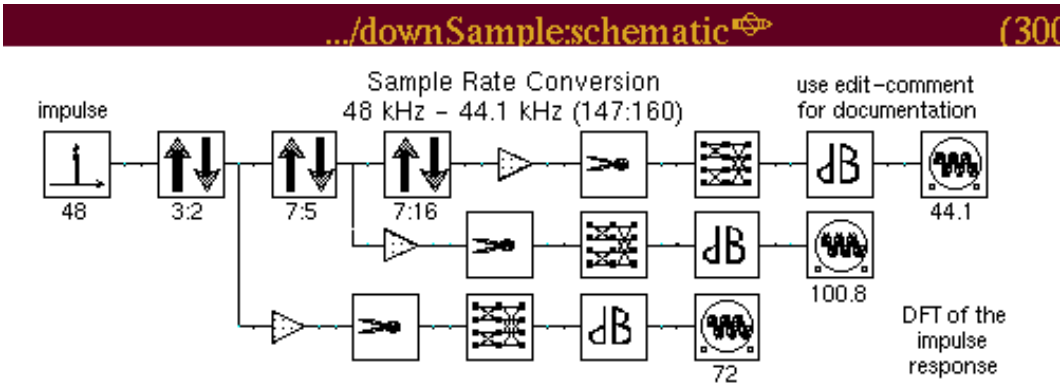


Figure 1.1. A snapshot of a session with the Ptolemy system [Buck92] that shows a sample rate conversion system for interfacing between a digital audio tape player and a compact disc player.

rates. The sample rate conversion is performed by three polyphase FIR filters that respectively perform 3:2, 7:5 and 7:18 rate conversions, and the cascade of blocks rooted at each filter's output simply scales the corresponding signal and displays

its frequency content.

Now the system represented in Figure 1.1 would normally receive input continuously from the DAT player. Each rate-changing FIR filter is applied repeatedly to successive data items that emerge from the output of the previous stage of the chain. In just 10 minutes, this system must process over 28 million input samples, and we see that it makes sense to model the input data sequence as a *semi-infinite* sequence that starts at some fixed time (the time when the system is activated) and extends to infinity. Correspondingly, we model the computation represented in Figure 1.1 as an infinite sequence of actor executions.

Three important issues emerge when attempting to derive an implementation of an infinite schedule from a dataflow graph. First, infinite schedules have the potential of requiring unbounded amounts of memory to buffer tokens as they are queued along the graph edges. Second, if deadlock arises, no more executions are possible and the infinite schedule cannot be carried out; similarly, if a subsystem becomes deadlocked, no more actors in that subsystem can be executed (even though it may be possible to continue executing actors outside the subsystem). In either case, if we are attempting to implement a system in which all operations are applied repeatedly on conceptually infinite data, then deadlock indicates an error.

Finally, we must provide a mechanism to sequence the actor executions in accordance with the given schedule. One option is to implement a software kernel that dynamically detects which actors have sufficient data on their inputs to be fired and determines when these actors are executed. However, the run-time overhead of this scheme is undesirable, particularly when a significant percentage of the invocations requires low computation time. An alternative is to store the schedule in memory as an infinite loop, thereby achieving *static scheduling*, and clearly

this is only feasible if the schedule is *periodic*.

Lee and Messerschmitt resolved these issues for SDF graphs by providing efficient techniques to determine at compile-time whether or not an arbitrary SDF graph has a periodic schedule that neither deadlocks nor requires unbounded buffer sizes [Lee87]. They also defined a general and efficient framework for constructing such a periodic schedule whenever one exists. The suitability of SDF for describing a large class of useful signal processing applications and the facility for achieving the advantages of static scheduling have motivated the use of SDF and closely related models in numerous design environments for DSP [Lauw90, Lee89, Ohal91, Prin92, Ritz92, Veig90]. A large part of this thesis is devoted to constructing static periodic schedules in such a way that the resulting target program is optimized.

A number of generalizations of the SDF model have been studied. In these new models, the methods for analyzing SDF graphs were extended or combined with additional techniques to incorporate actors that are more general than SDF, along with, in most cases, new techniques for constructing schedules. The objectives were to maintain at least a significant part of the compile-time predictability of SDF while broadening the range of applications that can be represented, and possibly, allowing representations that expose more optimization opportunities to a compiler. An example is the *token flow model,* which was defined by Lee in [Lee91] and explored further by Buck in [Buck93]. 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-null sequence of conditional actor invocations, where each actor is either invoked unconditionally or invoked conditionally based on the value of boolean tokens, that produces 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. Buck presents techniques for finding finite complete cycles whenever they exist, and heuristic techniques are developed to efficiently deal with graphs that don't have finite complete cycles or cannot be implemented with bounded memory.

In [Gao92], Gao et al. have studied a programming model in which non-SDF actors are used only as part of predefined constructs. 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. However, Gao's model, although more general than SDF, has significantly less expressive power than the token flow model of Buck.

Third, 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 can be specified to produce a 2×3 grid consisting of six tokens each time it is invoked. Lee demonstrated that in addition to substantially improving the expressive power of the unidimensional SDF model, multidimensional SDF also exposes parallelism more effectively than unidimensional SDF.

Also, in [Lauw94], Lauwereins et al. have proposed a minor but very useful generalization of the SDF model, called *cyclo-static dataflow*. In cyclo-static

dataflow, the number of tokens produced and consumed by an actor can vary between firings as long as the variations form a certain type of periodic pattern. For example, consider a *distributor* operator, which routes data received from a single input to each of two outputs, out_1 and out_2, in alternation. In cyclo-static dataflow, 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, … (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 out_1, and according to complementary pattern 0, 1, 0, 1, … on the edge corresponding to out_2. A general cyclo-static dataflow graph can be compiled as a cyclic pattern of pure SDF graphs, and static periodic schedules can be constructed in this manner. A major advantage of cyclo-static dataflow is that it can eliminate large amounts of token traffic arising from the need to generate dummy tokens in corresponding SDF representations [Lauw94]. This leads to lower memory requirements and fewer run-time operations.

The techniques of this thesis are developed for pure (unidimensional) SDF graphs. Due to the close relation between SDF and Lee's multidimensional SDF, they can easily be extended work with multidimensional SDF. However, how the techniques are best extended to the other models described above is not obvious and calls for further investigation.

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 for 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, per-

haps 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.

Although the techniques in this thesis are presented in the context of block diagram programming, they can be applied to other DSP design environments. Many of the programming languages used for DSP, such as Lucid[Asch75], SISAL[McGr83] and Silage[Geni90] are based on or closely related to dataflow semantics. In these languages, the compiler can easily extract a view of the program as a hierarchy of dataflow graphs. A coarse level view of part of this hierarchy may reveal SDF behavior, while the local behavior of the macro-blocks involved are not SDF. Knowledge of the high-level synchrony can be used to apply "global" optimizations such as those described in this thesis, and the local subgraphs can be examined for finer SDF components. For example, in [Denn92], Dennis shows how recursive stream functions in SISAL-2 can be converted into SDF graphs. In signal processing, usually a significant fraction of the overall computation can be represented with SDF semantics, so it is important to recognize and exploit SDF behavior as much as possible.

1.3 Compilation Model

Figure 1.2 outlines the process of compiling an SDF block diagram program that is used in the Gabriel [Ho88a] and Ptolemy [Pino94] systems. This is the compilation model that the techniques in this thesis are geared towards. The compilation begins with an SDF representation of the block diagram program specification and from this SDF graph, a periodic schedule is constructed. A *code generator* steps through this schedule and for each actor instance that it encoun-

ters, it generates a sequence of machine instructions, obtained from a predefined library of actor code blocks, that implements the actor. The sequence of code

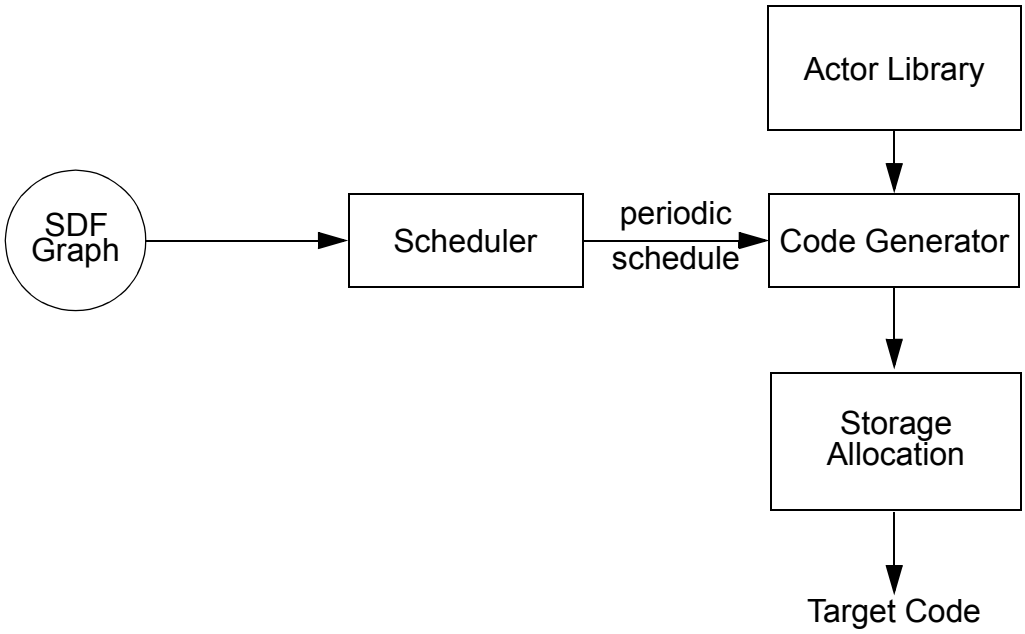


Figure 1.2. Compiling an SDF graph.

blocks output by the code generator is processed by a storage allocation phase that inserts the necessary instructions to route the data appropriately between actors and assigns variables to memory locations. The output of this storage allocation phase is the target program.

This form of block diagram compilation is referred to as *threading* [Bier93] since the target program is formed by linking together predefined code blocks. An alternative approach, called *synthesis*, involves first translating the block diagram to an intermediate language — possibly by threading code blocks that are defined the intermediate language — and then compiling the intermediate language into C

or assembly language. Examples of code generation systems that use the synthesis approach are the GOSPL [Covi87] and QuickSig [Karj88] systems, which first translate the block diagram to LISP, and the Mentor Graphics DSP Station. Most of the techniques developed in this thesis can be applied to synthesis; however, for clarity, we consistently use the threading model throughout the thesis.

In our application of threading, we perform strictly inline code generation. An alternative would be to define a subroutine for each actor and map the periodic schedule into a list of subroutine calls. However, each subroutine call induces run-time overhead. The principal components of the subroutine overhead come from saving the return address, passing arguments, allocating and deallocating local variable storage, branching to the subroutine, retrieving the return address, returning control from the subroutine, and saving and restoring the state of machine registers. Clearly if subroutines are used, the total subroutine overhead can be very detrimental if there are many actors of small granularity. The main reason that we prefer inline code over subroutines is to avoid subroutine overhead.

There is a danger, however, in using inline code, particularly for embedded system implementations, which typically can afford only very limited amounts of memory. The danger is that unmanageably large code size can result from actors that are invoked multiple times in the periodic schedule. For example, if an actor is invoked 100 times in the schedule, a straightforward inline implementation of the schedule will require 100 copies of the actor's code block to be inserted in the target code. Clearly, such code duplication can consume enormous amounts of memory, especially if complex actors having large code blocks are involved or if high invocation counts are involved.

Generally, the only mechanism to combat code size explosion while maintaining inline code is the use of loops in the target code. Clearly, if an actor's code

block is encapsulated by a loop, then multiple invocations of that actor can be carried out without any code duplication. For example, for the system in Figure 1.1, as it is represented in Ptolemy, over 9000 actor code blocks are required in the target code if inline code generation is applied with out employing any looping, while by carefully applying loops, the target code can be reduced to only 70 code blocks. A large part of this thesis is devoted to the construction of efficient loop structures from SDF graphs to allow the advantages of inline code generation under stringent memory constraints. We will elaborate on this problem informally in the following section, and then present it formally in Chapter 2.

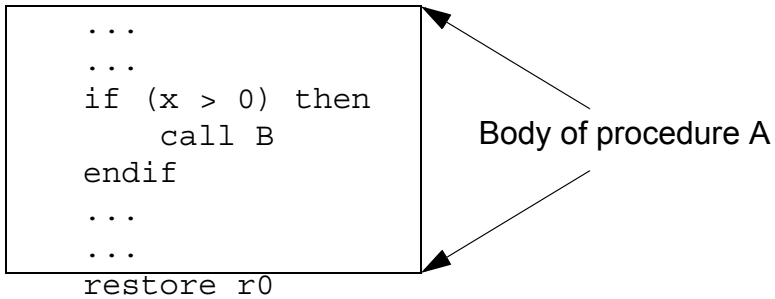
Until recently, it was widely believed that increased code size was the root cause of all aspects of the subroutine/inline code trade-off that favor the use of subroutines. However, experimental and analytical studies performed by Davidson revealed that inlining can also have a negative impact on register allocation [Davi92]. These effects however are largely artifacts of code generation conventions in modern compilers. For example, consider the conventional *callee-save* method of maintaining the integrity of registers across subroutine calls. In this convention, the values in the registers used by a subroutine are saved (stored to memory) upon entry to the subroutine, and the saved values are restored in the corresponding registers just before returning from the subroutine.

Figure 1.3 shows an example of how this convention can cause inlining to increase the amount of register-memory traffic in a program. Figure 1.3(a) shows an outline of the compiled code for two procedures A and B, where B is called by A. Here, x is a global variable, and the *save* and *restore* operations represent the register-memory and memory-register transfers involved in saving and restoring the registers used by a procedure. Also, we assume that B contains no subroutine calls, and the only subroutine call in A is the call to B that is shown. If procedure

(a)

```
procedure A
    save r0
    save r1
    ...
    ...
    if (x > 0) then
        call B
    endif
    ...
    ...
    restore r0
    restore r1

procedure B
    save r2
    save r3
    ...
    body of procedure B
    ...
    restore r2
    restore r3
```



(b)

```
procedure A
    save r0
    save r1
    save r2
    save r3
    if (x > 0) then
        ...
        body of procedure B
        ...
    endif
    restore r0
    restore r1
    restore r2
    restore r3
```

Figure 1.3. An example of how inlining can increase register-memory traffic under a callee-save register save/restore convention.

A is called 10 times, x is positive exactly 50% of the time, and B is not inlined in A, then it is easily verified that the calls to A result in a total of 30 register save operations and 30 restore operations. On the other hand, if B is inlined in A, as shown in Figure 1.3(b), then under the callee-save convention, the save/restore operations of B are moved to a location where they must be executed more frequently, and the 10 calls to A now result in 40 save operations and 40 restore operations.

In [Davi92] it is explained that inlining can also degrade performance with a *caller-save* convention, in which the registers used by the calling subroutine are saved by the caller just before transferring control to the callee, and the caller restores its registers just after control returns. It is also explained that the possible penalties for using inlining with the callee-save or caller-save conventions can be eliminated entirely through the application of dataflow-analysis. This has been demonstrated for callee-save systems in [Chow88] and for caller-save systems in [Davi89].

There is however one aspect of the negative interaction between inlining and register allocation that is not simply an artifact of typical compiler implementations. This is that variables of a subroutine that are placed in registers can be displaced to memory in inlined versions of the subroutine. This can lead to inefficient register allocation if frequently used variables are involved. Theoretically, this problem can be avoided since register assignments in inline code can be customized according to the context at the inlining boundaries, and thus, better register allocation is possible with inlined coded than with noninlned code. However, efficiently exploiting these opportunities for improvement is difficult, and it remains a challenge to systematically perform register allocation of inlined code in such a

way that an improvement is consistently obtained over the register allocation of corresponding noninlined code [Davi92].

An important conclusion from Davidson's study is that even if the code size increase of a particular inlining application does not lead to an increase in execution time, it is not guaranteed that the inlining will not decrease performance. This refutes the prior notion that the only detrimental affects of inlining are related to increases in code size. However, Davidson's study also shows that when the code size increase is not a factor, inlining is advantageous most of the time. Our use of inline code generation is motivated by this premise that if the code size increase is tolerable, then inline code generation is *usually* more efficient than heavy use of subroutines, and it is a main purpose of this thesis to examine the limits to which we can exploit inline code generation under strict memory constraints when compiling SDF programs.

1.4 Scheduling

1.4.1 Constructing Efficient Periodic Schedules

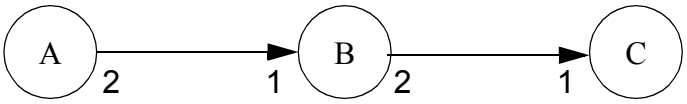
This section informally outlines the interaction between the construction of periodic schedules for SDF graphs and the memory requirements of the compiled code; also we review related work, particularly those efforts that involve interaction between scheduling and memory requirements in other contexts. In Section 3.4, we will elaborate in detail on the research efforts that are most closely related to the techniques developed in this thesis.

To understand the problem of scheduling SDF graphs to minimize code size, it is useful to examine closely the mechanism by which iteration is specified in SDF. In an SDF graph, iteration of actors in a periodic schedule arises whenever

the production and consumption parameters along an edge in the graph differ [Lee88a]. For example, consider the SDF graph in Figure 1.4(a), which contains three actors, labeled A, B and C. Each edge is annotated with the number of tokens produced and consumed by the incident actors; for example, actor A produces two tokens each time it is invoked and B consumes one token. The 2-to-1 mismatch on the left edge implies that within a periodic schedule, B must be invoked twice for every invocation of A. Similarly, the mismatch on the right edge implies that we must invoke C twice for every invocation of B.

Figure 1.4(b) shows four possible periodic schedules that we could use to implement Figure 1.4(a). For example, the first schedule specifies that first we are to invoke A, followed by B, followed by C, followed by B again, followed by three consecutive invocations of C. The parenthesized terms in schedules 2, 3 and 4 are used to highlight repetitive invocation patterns in these schedules. For example, the term $(2BC)$ in schedule 4 represents a loop whose iteration count is 2 and whose body is the invocation sequence BC; thus, $(2BC)$ represents the firing sequence $BCBC$. Similarly, the term $(2B(2C))$ represents the invocation sequence $BCCBCC$. Clearly, in addition to providing a convenient shorthand, these parenthesized loop terms, called *schedule loops*, present the code generator with opportunities to organize loops in the target program, and we see that schedule 2 corresponds to a nested loop, while schedules 3 and 4 correspond to cascades of loops. For example, if each schedule loop is implemented as a loop in the target program, the code generated from schedule 4 would have the structure shown in Figure 1.4(c).

We see that if each schedule loop is converted to a loop in the target code,



(a)

Periodic Schedules

(1). ABCBCCC

(2). A(2 B(2 C))

(3). A(2 B)(4 C)

(4). A(2 BC)(2 C)

(b)

```
for (i=0; i<2; i++) {
      code block for B
      code block for C
}
for (i=0; i<2; i++) {
      code block for C
}
```

(c)

Figure 1.4. An example used to illustrate the problem of scheduling SDF graphs to minimize code size.

then each *appearance* of an actor in the schedule corresponds to a code block in the target program. Thus, since actor C appears twice in schedule 4 of Figure 1.4(b), we must duplicate the code block for C in the target program. Similarly, we see that the implementation of schedule 1, which corresponds to the same invocation sequence as schedule 4 with no looping applied, requires seven code blocks. In contrast, in schedules 2 and 3, each actor appears only once, and thus no code duplication is required across multiple invocations of the same actor. We refer to such schedules as *single appearance* schedules, and we see that neglecting the code size overhead associated with the loops, any single appearance schedule yields an optimally compact inline implementation of an SDF graph with regard to code size. Typically the loop overhead is small, particularly in many programmable DSPs, which usually have provisions to manage loop indices and perform the loop test in hardware, without explicit software control. A large part of this thesis is devoted to studying properties of single appearance schedules, determining when single appearance schedules exist, and systematically constructing single appearance schedules whenever they exist. Additionally, we analyze the interaction between the use of schedule loops to construct compact schedules and the efficiency of buffering (the management of the FIFO queues corresponding to each edge in the SDF graph), and we present techniques to construct schedules that simultaneously minimize code size and support efficient buffering.

1.4.2 Related Work

Numerous research efforts have focused on constructing efficient parallel schedules from SDF graphs. These efforts operate on *homogeneous* SDF graphs; that is, SDF graphs in which each actor produces a single token on each output edge and consumes a single token from each input edge. Since iteration within a

periodic schedule, as defined in Subsection 1.4.1, does not arise in homogeneous SDF graphs, scheduling techniques for homogenous SDF graphs do not encounter the central problem addressed in this thesis — the management of code size and buffering when large amounts of iteration are present in general SDF graphs.

However, a number of the parallel scheduling techniques for homogeneous SDF graphs have important implications on code size. Many of these connections are related to the *unfolding factor* of a parallel schedule. The unfolding factor of a given periodic schedule S is the largest common factor (greatest common divisor) of the actor invocation counts in S (by the invocation count of an actor in S, we simply mean the number of times that the actor is invoked in S). The unfolding factor can also be viewed as the number of *minimal* periodic schedules that exist in the schedule. For example, a minimal periodic schedule for Figure 1.4(a) consists of 1 invocation of A, 2 invocations of B, and 4 invocations of C. Any schedule that invokes A, B and C U, $2U$ and $4U$ times, respectively, for some positive integer U, is also a periodic schedule, and U is referred to as the *unfolding factor* of the schedule. For example, the unfolding factor of the schedule $A(2B)A(2B)(8C)$ is 2.

A related term, which we will use extensively in this thesis, is the *blocking factor* of a periodic schedule. The blocking factor of a periodic schedule is simply the unfolding factor of a *blocked schedule*, which is an infinite repetition of a periodic schedule in which each cycle of the schedule must complete before the next cycle is begun. The distinctions between blocked and non-blocked periodic schedules are only relevant in a parallel scheduling context, and for parallel schedules, the increased flexibility offered by a non-blocked schedule can often provide more throughput than is possible with any blocked schedule. For example, in [Lee86],

Lee presents a homogeneous SDF graph that can be executed at a throughput of 0.5 minimal schedule periods per time unit (assuming that each actor takes unit time to execute) with a non-blocked schedule, and Lee shows that the best throughput attainable with a blocked schedule for this graph is $\frac{U}{2U+1}$, where U is the blocking factor. Thus, for Lee's example, a blocked schedule cannot match the performance of the given unblocked schedule for any finite blocking factor.

If inline code generation is performed and no looping is applied within a period of the periodic schedule, then the total amount of code space (across all processors) required to implement a general parallel periodic schedule is roughly proportional to the unfolding factor.

Also, given a representation of a computation as a homogeneous SDF graph, there is a fundamental upper bound on the throughput. This upper bound, which was established by Reiter in [Reit68], can be computed as the minimum over all directed cycles of the number of delays in a cycle divided by the sum of the computation times of all actors in the cycle. A multiprocessor schedule is called *rate-optimal* if it attains this throughput bound, and the reciprocal of the rate-optimal throughput is called the *iteration period bound*.

Thus, for a given homogeneous SDF graph, it is natural to ask if a rate-optimal schedule is attainable with a finite unfolding factor, and if so, what is the minimum unfolding factor that achieves the optimum throughput? In [Parh91], Parhi established that if we allow non-blocked schedules, the answer to the first question is always affirmative and provided a systematic technique for constructing finitely-unfolded rate-optimal schedules. The required unfolding factor for Parhi's scheme is the least common multiple of the number of delays in each directed cycle. In [Chao93] Chao found techniques to achieve rate-optimal sched-

ules with lower unfolding factors, and hence lower code size. Chao showed that a rate-optimal schedule can efficiently be constructed for an unfolding factor equal to the denominator of the reduced-fraction form of the iteration period bound, and that this is the minimum unfolding factor for which a rate-optimal schedule exists. Thus, Chao's techniques determine the rate-optimal schedule that has minimum code size. A related problem has been addressed by Murthy in [Murt94b] for the more restricted class of blocked schedules. It is shown that for a given homogenous SDF graph, we can determine in a finite number of steps whether or not there is a finite blocking factor for which a rate-optimal blocked schedule exists, and when such a blocking factor exists, we can determine in a finite number of steps the minimum blocking factor for which rate-optimal blocked schedules exist.

In contrast to our primary objective of minimum code size, many compilers for procedural languages apply transformations that deliberately increase the code size. One example is the inlining of subroutines, which we discussed in Section 1.3. Second, in *trace scheduling*, compile-time branch prediction is performed to estimate the most likely execution path through a program, and this path, called a *trace*, is scheduled as if it were a single basic block [Fish84]. This permits exploitation of the most abundant source of instruction-level parallelism — reordering code across basic block boundaries. For each instruction that moves across a basic block boundary, recovery code may have to be inserted just off the trace. For example, if along the trace, it is assumed that a particular conditional branch will be taken, then each instruction migrated from after the branch to a point before the branch may have to be "undone" if the branch is not taken. The insertion of such recovery instructions increases the code size. Once the most likely trace has been selected and reorganized, the next most likely trace is selected, and the process is repeated for any desired number of traces.

In *loop unrolling*, which is analogous to the unfolding of SDF graphs, the body of a loop is replicated to cover more than one iteration, and the iteration count is modified and possibly a prologue or epilogue is generated to guarantee that the unrolled version executes the correct number of iterations of the original loop [Dong79]. As with unfolding, unrolling facilitates the exploitation of inter-iteration parallelism at the expense of a roughly linear increase in code size.

A fourth example of a code-increasing program transformation is the duplication of code to eliminate unconditional branches [Muel92]. A significant number of unconditional jumps is generated by typical compilers. For example, when generating code for an if-then-else construct, compilers often place an unconditional branch at the end of the *then* section that skips over the else section. Here, the unconditional branch can be eliminated by appending to the then section a duplicate copy of the code at the target of the branch. The benefits of such code replication include fewer instructions executed, better program locality and increased opportunities for common subexpression elimination [Muel92].

In the application domain that we are concerned with in thesis — the domain of embedded real-time digital signal processing systems — the price paid for neglecting opportunities such as trace scheduling, loop unrolling, subroutine inlining or unconditional branch elimination is usually dominated by the penalty incurred when the target program does not fit within the on-chip memory limits. Given an SDF graph, we would like to first generate an efficient compact implementation, and then, if there is any remaining on-chip program memory, we can expand the code in a controlled manner to utilize it. In this thesis, we focus largely on the first part of this process — generating an efficient uniprocessor implementation with a minimal amount of code space.

The problem of scheduling SDF graphs to minimize the code size expan-

sion of inline code generation was first addressed by How [How90] in the context of the Gabriel project. How proposed a heuristic that involved consolidating subsystems of actors that were iterated the same number of times in a periodic schedule. Although How demonstrated that this approach often produced compact schedules, the technique did not adequately exploit looping opportunities that occur across subsystems that are iterated at different rates, and no systematic method was provided for avoiding or recovering from consolidations that produced deadlock. How's technique was subsequently extended to overcome these shortcomings [Bhat93], and the resulting scheduler, implemented in Ptolemy, was significantly more thorough in extracting looping opportunities. However, due to its use of a data structure that could grow exponentially with the size of the SDF graph, this scheduler became inefficient for graphs having large sample rate changes. An alternative loop scheduling algorithm was developed by Buck [Buck93]. This algorithm, which was in some ways an extension of How's scheme, was designed to be more time and space efficient that the technique of [Bhat93] while exploiting looping opportunities almost as thoroughly.

At the Aachen University of Technology, as part of the COSSAP design environment, the construction of compact schedules for SDF graphs was studied in the context of *minimum activation schedules* [Ritz93]. A major objective in this work was the minimization of context-switches that occur when distinct actors are executed in succession, and it was found that single appearance schedules are beneficial for this purpose. As one would expect, there are similarities between the techniques developed in this work on minimum activation schedules and the techniques developed in this thesis. The parallels between the work on minimum activation schedules and the techniques of this thesis are similar to the relationships between two major approaches to the compilation of nested loop procedural code

for vector computers. In Section 3.4, we will discuss in detail the problems and techniques involved in minimum activation schedules and in compiling nested loops for vector computers. We will also elaborate on the loop scheduling algorithms developed by How and Buck. Finally, we will also examine *thresholds*, a technique primarily used to compile procedural code for vector machines in which vector instructions on short vectors are cost-effective. The problems addressed by thresholds are closely related to issues encountered when scheduling loops from SDF graphs, particularly the issues discussed in [Ritz93] on constructing minimum activation schedules.

1.5 An Overview of the Remaining Chapters

In this introductory chapter, we have described the use of synchronous dataflow as an underlying model for block diagram programming of embedded digital signal processing applications. We have also defined a compilation model for synthesizing software from SDF-based graphical programs, we have discussed how scheduling plays a central role in this compilation process, and we have defined the class of single appearance schedules, which minimize code size under inline code generation.

In the following chapter, we formally review the basic concepts introduced casually in this chapter, and we build on the fundamental principles of SDF to develop a formal framework for constructing and manipulating schedules that contain loops. This framework is used first to present a technique, called *factoring*, for transforming a schedule into an alternative schedule that carries out the same computation, but with lower memory cost to implement the FIFO buffers corresponding to the graph edges. The concept of factoring is then applied to define a form a

single appearance schedule, called a *fully reduced schedule*, which is roughly a single appearance schedule that results when the factoring transformation is applied to a given single appearance schedule until no more opportunities exist for applying the factoring transformation. It is shown that a fully reduced single appearance schedule can be constructed from any legitimate single appearance schedule, and that under certain assumptions, the memory required for buffering by the fully reduced schedule is less than or equal to the memory required for buffering by the schedule from which it is derived.

We also show that any fully reduced schedule has unit block factor. Since a fully reduced single appearance schedule can be derived from any single appearance schedule, it follows that the existence of a single appearance implies the existence of a single appearance schedule that has unit blocking factor. We discuss the implications that this fundamental property has on code generation, and later we apply this property to help establish a recursive necessary and sufficient condition for the existence of a single appearance schedule. To develop this condition, we also apply a special form of precedence independence, called *subindependence*, for strongly connected SDF subgraphs.

In Chapter 3, we apply the concept of subindependence and our condition for the existence of single appearance schedules to develop a general class of scheduling algorithms, and we establish that all algorithms in this class guarantee certain useful properties of code size compactness. We also demonstrate how algorithms in this class can be tailored to additional scheduling objectives while maintaining the properties of compact code size. Two specific additional objectives are addressed: increasing the amount of buffering performed in registers and minimizing the amount of memory required for buffering. The scheduling framework defined in Chapter 3 has been implemented in Ptolemy, a design environment for

simulation, prototyping, and software synthesis of heterogeneous systems [Buck92]. A large part of the implementation in Ptolemy was performed by Joseph Buck, a graduate student colleague at the time and now with Synopsys Inc., and Soonhoi Ha, a post-doctoral fellow at U.C. Berkeley at the time and now a lecturer at Seoul National University.

In Chapter 3, for a restricted class of SDF graphs, we also present a technique that computes the single appearance that minimizes the memory required for buffering over all single appearance schedules. This work was done jointly with Praveen Murthy, a fellow graduate student at U. C. Berkeley.

In Chapter 4, we present techniques for improving the efficiency of buffering for a given uniprocessor schedule. The optimizations include compile-time dataflow analysis techniques to determine as much as possible about addressing patterns; analysis of the loop structures in a schedule to provide flexibility for overlaying buffer memory to a storage allocator; and techniques to optimize the management of *circular*, or *modulo*, buffers, which are useful for implementing dataflow edges that have delay. Finally, in Chapter 5, we discuss directions for related future work.

2

LOOPED SCHEDULES

2.1 Background

For reference, the definitions behind much of the terminology and notation that is introduced in this and subsequent chapters can be located by using the index at the end of the thesis.

2.1.1 Mathematical Terms and Notation

We adopt the convention of indexing vectors and matrices using functional notation rather than subscripts or superscripts. Thus, for example $\mathbf{x}(3)$ represents the third component of the vector $\mathbf{x}$, and $M(i,j)$ represents the value corresponding to the ith row and jth column of the two-dimensional matrix M. We denote the transpose of the vector $\mathbf{x}$ by $\mathbf{x}^T$.

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. Given a fraction $f = \frac{a}{b}$, we define $numer(f) = a$ and $denom(f) = b$; and given a positive rational number q, by $ReducedFraction(q)$ we denote that unique fraction f for which $numer(f)$ and $denom(f)$ are positive and mutually coprime, and $\frac{numer(f)}{denom(f)} = q$. 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 $\lceil r \rceil$. Finally, given two arbitrary sets S_1 and S_2, we define the difference of these two sets by $S_1 - S_2 \equiv \{s \in S_1 \mid s \notin S_2\}$, and we denote the number of elements in a finite set S by $|S|$.

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, and $f(x)$ is $\Theta(g(x))$ if it is both $O(g(x))$ and $\Omega(g(x))$.

2.1.2 Graph Concepts

This section introduces the basic graph-theoretic concepts that will be applied in this thesis. For elaboration on any of these concepts, the reader is referred to [Corm90].

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 $source(e)$ and $sink(e)$ such that $source(e), sink(e) \in V$. Each member of V is called a **vertex** of the directed multigraph and each member of E is called an **edge**. We say that a directed multigraph is **trivial** if it contains only one vertex. If e is an edge in a directed multigraph, we say that $source(e)$ is the **source** vertex of e; $sink(e)$ is the **sink** vertex of e; e is **directed from** $source(e)$ **to** $sink(e)$; e is an **output edge** of $source(e)$; and e is an **input edge** of $sink(e)$. We represent a directed multigraph pictorially by drawing a circle for each vertex, and for each edge e, drawing a directed line segment from the circle corresponding to $source(e)$ to the circle corresponding to $sink(e)$. For example, the directed multigraph depicted in Figure 2.1 consists of vertices v_1, v_2, v_3, v_4, and edges e_1, e_2, e_3, e_4, e_5, where $source(e_1) = v_1$, $sink(e_1) = v_2$, $source(e_2) = v_2$, $sink(e_2) = v_3$, $source(e_3) = v_1$, $sink(e_3) = v_3$, $source(e_4) = v_1$, $sink(e_4) = v_3$, $source(e_5) = v_4$, and $sink(e_5) = v_4$.

Given two not necessarily distinct vertices v_1 and v_2 in a directed multi-

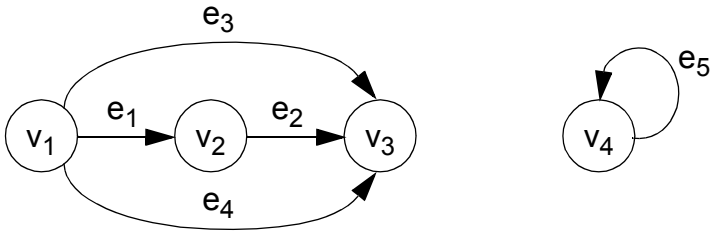


Figure 2.1. A directed multigraph.

graph (V,E), we say that v_1 is a **predecessor** of v_2 if there exists $e \in E$ such that $source(e) = v_1$ and $sink(e) = v_2$; 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. 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 $\{e \in E | (source(e), sink(e) \in V')\}$. 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 $sink(e_1) = source(e_2)$, $sink(e_2) = source(e_3)$, $\ldots$. We say that the path $p = e_1, e_2, e_3, \ldots$ **passes through** each member of $Z_p = \left(\bigcup_i \{source(e_i)\}\right) \cup \left(\bigcup_i \{sink(e_i)\}\right)$, and we refer to the SDF 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 actors are members of Z_p. Given a finite path $p = e_1, e_2, \ldots, e_n$, we say that p is *directed from* $source(e_1)$ *to* $sink(e_n)$. 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 $source(e)$ to $sink(e)$, then

we occasionally denote e by $source(e) \rightarrow sink(e)$; thus, in Figure 2.1, $v_1 \rightarrow v_2$ represents the edge e_1, whereas $v_1 \rightarrow v_3$ cannot represent e_3 because e_4 also has the same source and sink vertices.

We say that a directed multigraph is **connected** if for each distinct pair of vertices v_1, v_2, there is a path directed from v_1 to v_2 or there is a path directed from v_2 to v_1. Thus, the directed multigraph in Figure 2.1 is not connected, while the subgraph associated with $\{v_1, v_2, v_3\}$ is connected. 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 $1 \leq i \leq n$, $subgraph(V_i)$ is connected; and for each $e \in E$, $source(e), sink(e) \in V_j$ 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. For example, the connected components of the directed multigraph in Figure 2.1 are $\{v_1, v_2, v_3\}$ and $\{v_4\}$. 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 *and* there is a path directed from v_2 to v_1. 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'. For example, the directed multigraph

in Figure 2.2 has three strongly connected components — $\{v_1, v_2\}$,

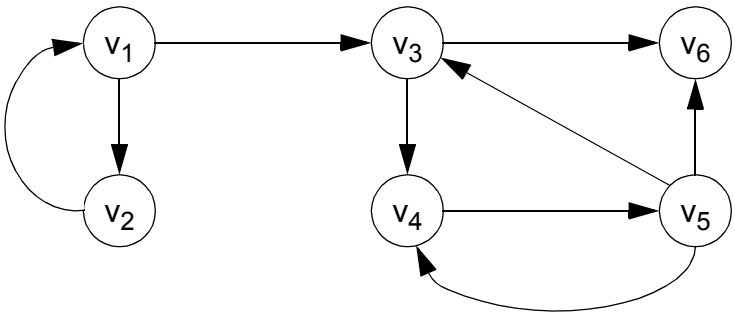


Figure 2.2. A directed multigraph that has three strongly connected components — $\{v_1, v_2\}$, $\{v_3, v_4, v_5\}$, and $\{v_6\}$.

$\{v_3, v_4, v_5\}$, and $\{v_6\}$. The strongly connected components of a directed multigraph can be determined in linear time by an algorithm developed by Tarjan [Tarj72].

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 $sink(e) = v$, and a **root strongly connected component** of (V, E) is a strongly connected component V' of (V, E) such that $\{e \in E \mid (source(e) \notin V' \textbf{ and } sink(e) \in V')\} = \varnothing$. For example, in Figure 2.2 there are no root vertices, and there is one root strongly connected component — $\{v_1, v_2\}$. 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 $e \in E$, $((source(e) = v_i)$ **and** $(sink(e) = v_j) \Rightarrow (i < j))$; that is, the source vertex of each edge occurs earlier in the ordering than the sink vertex. Thus, in Figure 2.2, $subgraph(v_1, v_3, v_4, v_6)$ has two distinct topological sorts — (v_1, v_3, v_4, v_6) and (v_1, v_3, v_6, v_4). 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}. For example, in Figure 2.2, $subgraph(v_1, v_3, v_6)$ is chain-structured, while $subgraph(v_3, v_4, v_6)$ is neither chain-structured nor well-ordered; and in Figure 2.1, $subgraph(v_1, v_2, v_3)$ is well-ordered but not chain-structured.

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

2.1.3 Synchronous Dataflow

Formally, an SDF graph is a directed multigraph in which each edge α has three properties in addition to $source(\alpha)$ and $sink(\alpha)$ — $delay(\alpha)$, which is a nonnegative integer that gives the number of initial data values associated with α; $produced(\alpha)$, a positive integer that indicates the number of data values, called **tokens**, produced onto the channel corresponding to α by each execution of the

computation corresponding to $source(\alpha)$; and $consumed(\alpha)$, a positive integer that represents the number of tokens consumed from α by each execution of $sink(\alpha)$. 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 $\alpha \in edges(G)$, $produced(\alpha) = consumed(\alpha) = 1$, then we say that G is a **homogeneous** SDF graph.

Conceptually, each edge in G, corresponds to a FIFO 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 $consumed(\alpha)$ tokens from the head of the buffer for each input edge α, and appending $produced(\beta)$ 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 $consumed(\alpha)$ 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 $S = A_1A_2A_3\ldots$ of actors in G. Each term A_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**. An **admissable schedule** for G is a schedule $A_1A_2A_3\ldots$ for G such that each

invocation A_i is fireable immediately after $A_1, A_2, \ldots, A_{i-i}$ have fired in succession. The process of successively firing the invocations in an admissable 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.

Consider the simple SDF graph in Figure 2.3. Each edge is annotated with

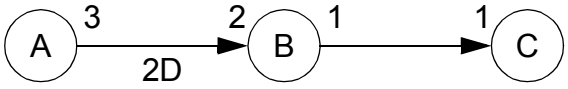


Figure 2.3. A simple SDF graph.

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 $A \rightarrow B$ and the number of tokens on the edge $B \rightarrow C$. Then, since there is a delay of 2 on the left side edge, the initial state of the system is $(2,0)$. Thus, the first invocation of the schedule — B_1 — 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 the corresponding actor has no input edges, the second invocation of the schedule is fireable, and its

firing leads to the state $(3,1)$. It is easily verified that the remaining three invocations in the schedule are fireable and the sequence of buffer states that results from these remaining firings is $(3,0), (1,1), (4,1)$. Thus, $BACBA$ is an admissable schedule for the SDF graph in Figure 2.3. In contrast, the slightly different schedule $BACBB$ is not admissable, since only one token resides on the input edge of B prior to invocation B_3.

If $S = A_1A_2A_3\ldots$ is not an admissable 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) $sink(\alpha) = A_i$ and (2) the buffer associated with $sink(\alpha)$ contains less than $consumed(\alpha)$ 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 admissable if and only if it does not terminate on any edge.

We say that a schedule S is a **periodic schedule** if it invokes each actor at least once and produces no net change in the system state — for each edge α, (the number of times $source(\alpha)$ is fired in $S) \times produced(\alpha)$ = (the number of times $sink(\alpha)$ is fired in $S) \times consumed(\alpha)$. For example for the SDF graph in Figure 2.3, 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 the left-side edge and $+1$ token on the right-side edge, 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; we will elaborate on this in Section 2.3.

By definition, a schedule that invokes each actor A $\mathbf{b}(A)$ times is a periodic schedule if and only if

$$\text{for each edge } \alpha \text{ in } G,$$
$$\mathbf{b}(source(\alpha)) \times produced(\alpha) = \mathbf{b}(sink(\alpha)) \times consumed(\alpha) \quad . \qquad (2\text{-}1)$$

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

$$\Gamma \mathbf{b} = \mathbf{0}, \qquad (2\text{-}2)$$

where Γ, called the **topology matrix** of G, is a two-dimensional 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[2]

$$\Gamma(\alpha, A) = \begin{cases} produced(\alpha), & \text{if } A = source(\alpha) \\ -consumed(\alpha), & \text{if } A = sink(\alpha) \\ 0, & \text{otherwise} \end{cases} \qquad (2\text{-}3)$$

Thus, G has a periodic schedule only if it's topology matrix does not have

1. Recall that in our definition of *periodic schedule*, we do not require admissability — a periodic schedule need not be admissable.
2. This formulation assumes that G does not contain any *self-loops*, edges whose source and sink vertices are identical, such as edge e_5 in Figure 2.1. In an SDF graph, a self-loop edge α precludes the existence a periodic schedule if $produced(\alpha) \neq consumed(\alpha)$; otherwise it has no effect on the existence of a periodic schedule, and thus it can be ignored in this analysis.

full rank. Furthermore, in [Lee87], Lee shows that the rank of Γ is always either n or $n-1$, where n denotes the number of actors in G, and that whenever the rank is $n-1$, a positive-integer vector exists that satisfies the balance equations. Thus, when a periodic schedule exists, the null space of Γ has dimension 1, and there is a unique minimum positive integer vector that satisfies (2-2). This unique minimum positive-integer vector is called the **repetitions vector** of G, and we denote this vector by $\mathbf{q}_G$, or simply by $\mathbf{q}$ if G is understood from context. Clearly, any positive integer multiple of the repetitions vector also solves the balance equations, and since the null space of Γ is of dimension $n-(n-1) = 1$, every positive-integer vector that solves the balance equations is a positive-integer multiple of the repetitions vector. Note that 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 the main properties that follow from the definition of the repetitions vector.

Fact 2.1: A positive-integer vector is the repetitions vector of a connected SDF graph if and only if its components are coprime and it satisfies the balance equations.

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 $\mathbf{q}_G$ exists and there exists a positive integer J_0 such that S invokes each $A \in actors(G)$ 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 2.4. The topology matrix for this SDF graph is

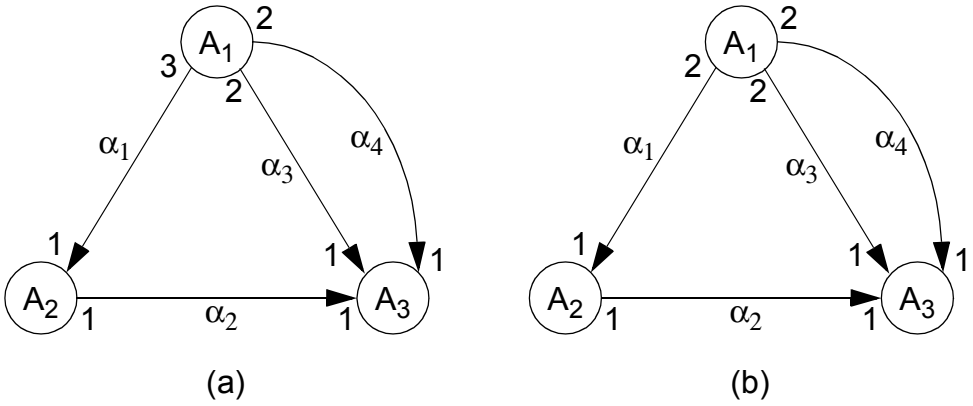


Figure 2.4. (a). An SDF graph that does not have a periodic schedule.
(b). A slightly modified version that has a periodic schedule.

$$\Gamma = \begin{bmatrix} 3 & -1 & 0 \\ 0 & 1 & -1 \\ 2 & 0 & -1 \\ 2 & 0 & -1 \end{bmatrix}, \tag{2-4}$$

where each α_i corresponds to the ith row and each A_i 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_1, three firings of A_2 are required to return edge α_1 to its initial state of

having no tokens queued in its buffer, and then three firings of A_3 are required to return α_2 to its initial state. However, since α_3 is an input edge of A_3 and A_1 produces only two tokens per firing on α_3, only two firings of A_3 are possible for each firing of A_1. Thus, any infinite admissable sequence of firings for this graph will produce an unbounded token accumulation on α_1, α_2, or both.

If we change $produced(\alpha_1)$ to 2, the resulting SDF graph, shown in Figure 2.4(b), has a periodic schedule. The topology matrix of this new SDF graph is

$$\Gamma' = \begin{bmatrix} 2 & -1 & 0 \\ 0 & 1 & -1 \\ 2 & 0 & -1 \\ 2 & 0 & -1 \end{bmatrix}. \tag{2-5}$$

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 2.4(b) is given by

$$\mathbf{q}(A_1, A_2, A_3) = (1, 2, 2)^T . \tag{2-6}$$

From (2-6), we see that $A_1A_2A_3A_2A_3$ and $A_1A_2A_2A_3A_3$ are minimal periodic schedules, and $A_1A_2A_1A_2A_2A_3A_2A_3A_3A_3$ is a periodic schedule having blocking factor 2. All three of these schedules are admissable.

In this thesis we are primarily concerned with schedules that are both periodic and admissable, 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 admissable 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 2.5. The repetitions vector for this graph is given by

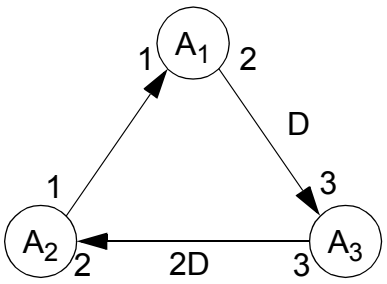


Figure 2.5. An SDF graph that has a repetitions vector but does not have an admissable schedule.

$$\mathbf{q}(A_1, A_2, A_3) = (3, 3, 2)^T , \tag{2-7}$$

and thus periodic schedules exits. However, one can easily verify that there are only five possible non-null admissable schedules for this SDF graph — A_2, A_2A_1, $A_2A_1A_3$, $A_2A_1A_3A_2$, and $A_2A_1A_3A_2A_1$. Since none of these five schedules con-

tains 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_1 from one to two, valid schedules, such as the schedule $A_2A_1A_3A_2A_1A_3A_2A_1$, exist.

Associated with any connected, consistent SDF graph G, and 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 [Lee87] throughout J successive minimal schedule periods for G. Each vertex of the APG corresponds to an actor invocation and 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 2.6 below shows the APG for Figure 2.3 and blocking factor 1. See [Sih91] for an efficient algorithm that systematically constructs the APG.

We say that two SDF graphs G_1 and G_2 are **isomorphic** if there exist

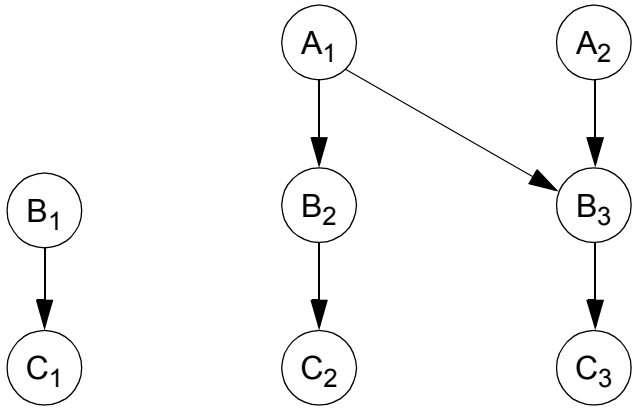


Figure 2.6. The acyclic precedence graph for Figure 2.3 and unity blocking factor.

bijective mappings $f_1 : actors(G_1) \rightarrow actors(G_2)$ and $f_2 : edges(G_1) \rightarrow edges(G_2)$ such that for each $\alpha \in edges(G_1)$, $source(f_2(\alpha)) = f_1(source(\alpha))$, $sink(f_2(\alpha)) = f_1(sink(\alpha))$, $delay(f_2(\alpha)) = delay(\alpha)$, $produced(f_2(\alpha)) = produced(\alpha)$, and $consumed(f_2(\alpha)) = consumed(\alpha)$. Intuitively, two SDF graphs are isomorphic if they differ only by a relabeling of the actors and/or edges. For example, $subgraph(\{A_2, A_3\})$ in Figure 2.5 and $subgraph(\{A, B\})$ in Figure 2.3 are isomorphic.

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 $sink(\alpha)$ in a minimal schedule period by $total\_consumed(\alpha, G)$; that is $total\_consumed(\alpha, G) = \mathbf{q}_G(sink(\alpha)) \times consumed(\alpha)$. Since in a periodic schedule, the number of tokens produced on an edge equals the number of tokens consumed, we also have that $total\_consumed(\alpha, G) = \mathbf{q}_G(source(\alpha)) \times produced(\alpha)$. If, G is understood from context, we may suppress the second argument and write $total\_consumed(\alpha)$ in place of $total\_consumed(\alpha, G)$.

2.1.4 Computing the Repetitions Vector

The repetitions vector can be computed efficiently by applying depth-first search. An algorithm based on the one that is used in the Gabriel [Lee89] and Ptolemy [Buck92] systems is described by the pseudocode segment below. In this algorithm, we maintain an array of fractions called *reps*. At the end of the algo-

rithm, $numer(reps(A)) = \mathbf{q}_G(A)$, for each actor A in the input SDF graph G, if G has a repetitions vector.

procedure ComputeRepetitions (G)
  for each $A \in actors(G)$, initialize $reps(A)$ to zero
  select an actor $A' \in actors(G)$
  SetReps (A', 1)
  compute $x = lcm(\{denom(reps(A)) \mid A \in actors(G)\})$
  for each $A \in actors(G)$, $reps(A) = x \times reps(A)$
  for each edge $\alpha \in edges(G)$
    if $(reps(source(\alpha)) \times produced(\alpha)) \neq (reps(sink(\alpha)) \times consumed(\alpha))$
      error: inconsistent graph
      exit

procedure SetReps (A, n)
  $reps(A) = n$
  for each output edge α of A
    if $reps(sink(\alpha)) = 0$
      SetReps($sink(\alpha)$,
        $ReducedFraction((n produced(\alpha)) / consumed(\alpha))$)
  for each input edge α of A
    if $reps(source(\alpha)) = 0$
      SetReps($source(\alpha)$,
        $ReducedFraction((n consumed(\alpha)) / produced(\alpha))$)

Assuming the production and consumption parameters on the edges are bounded — so that computing the least common multiple of two numbers is an elementary operation — this algorithm has time complexity that is linear in the number of actors and edges in the input SDF graph.

2.1.5 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 defines a class of scheduling algorithms, called *class-S algorithms*, that construct valid schedules given a positive integer multiple of the repetitions vector $\mathbf{r}$. 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 shows that a class-S algorithm constructs a valid schedule if and only if the SDF graph in question is consistent [Lee87].

One specific class-S scheduling algorithm is given by procedure *ConstructValidSchedule* in Figure 2.7. It is easily verified that if we assume that the number of input and output edges for a given actor is bounded by a constant, which is a reasonable assumption in practice, then the time complexity of *ConstructValidSchedule* is $O(\sum_{A \in actors(G)} \mathbf{r}(A))$, where G is the input SDF graph.

2.2 Looped Schedule Terminology and Notation

In Section 2.1, we reviewed relevant mathematical background, and we summarized several important developments in [Lee86]. In this section, we begin presenting the contributions of this thesis. We start by introducing some basic concepts and terminology pertaining to uniprocessor scheduling that will be used

procedure ConstructValidSchedule (G, $\mathbf{r}$)

- define *ready* to be a queue of actors
- define *queued* and *scheduled* to be vectors of non-negative integers indexed by the actors in G
- define S to be a schedule; initialize S to be the null schedule.

for each edge α

$\mathbf{b}(\alpha) = delay(\alpha)$

for each actor A in G

$r = \mathbf{r}(A)$

for each input edge α of A

$r = min(\{r, \lfloor delay(\alpha)/consumed(\alpha) \rfloor\})$

if $(r > 0)$

append A to the *ready* queue

$queued(A) = r$

$scheduled(A) = 0$

repeat until *ready* is empty

remove the actor A at the head of *ready*

append $queued(A)$ successive invocations of A to S

$scheduled(A) = scheduled(A) + queued(A)$

$n = queued(A)$; $queued(A) = 0$

for each input edge α of A

$\mathbf{b}(\alpha) = \mathbf{b}(\alpha) - (n \times consumed(\alpha))$

for each output edge α of A

$\mathbf{b}(\alpha) = \mathbf{b}(\alpha) + (n \times produced(\alpha))$

for each output edge α of A

$r = \mathbf{r}(sink(\alpha)) - scheduled(sink(\alpha))$

for each input edge β of $sink(\alpha)$

$r = min(\{r, \lfloor \mathbf{b}(\beta)/consumed(\beta) \rfloor\})$

if $(r > queued(sink(\alpha)))$

append $sink(\alpha)$ to *ready*

$queued(A) = r$

for each actor A in G

if $(scheduled(A) \neq \mathbf{r}(A))$

error: inconsistent graph

exit

output S

Figure 2.7. An algorithm for constructing a valid schedule for a connected SDF graph.

heavily throughout the remainder of the thesis.

Definition 2.1: 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 corre-

spond 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, $actors((2(2B)(5A))) = \{A, B\}$, $actors(X(2Y(3Z)X)) = \{X, Y, Z\}$, $appearances(C, (3CA)(4BC)) = 2$, and $appearances(A, (2ABAC)(3A)) = 3$. Given a looped schedule S and an actor A, we define $inv(A, S)$ to be the number of times that S invokes A. Similarly, if S_0 is a subschedule, we define $inv(S_0, S)$ to be the number of times that S invokes S_0. For example, if $S = A(2(3BA)C)BA(2B)$, then $inv(B, S) = 9$, $inv((3BA), S) = 2$, and $inv(\text{first appearance of } BA, S) = 6$. 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 2.4. 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 2.4: 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

$$\forall(A \in actors(S)),\ appearances(A, S') = appearances(A, S) \quad .$$

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 $source(\alpha)$ that precede the ith invocation of $sink(\alpha)$ in S; and we define $T(\alpha, i, S)$ to denote the number of tokens on α just prior to the ith invocation of $sink(\alpha)$ in an execution of S. For example, consider the SDF graph in Figure 2.3 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 $T(\alpha, 2, BC(2ABC)) = 1$.

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. 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 $buffer\_memory(S) = \sum_{\alpha \in edges(G)} max\_tokens(\alpha, S)$. In Figure 2.3, if $S = BC(2ABC)$, then $max\_tokens(A \rightarrow B, S) = 4$, $max\_tokens(B \rightarrow C, S) = 1$, and $buffer\_memory(S) = 4 + 1 = 5$.

Our model of buffering here — each is buffer mapped to a contiguous and independent block of memory — 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, Pino94, Ritz92]. However, perfectly valid target programs can be generated without these restrictions. In this and the following chapter, we examine the interaction of scheduling and memory requirements under the assumption that each buffer is mapped to a separate, independent block of contiguous memory. Developing scheduling techniques that take advantage of more flexible buffer implementations is a topic for future work; although some of the pertinent issues are explained in Chapter 4, which discusses how to increase the efficiency of buffering for a given schedule.

2.3 Non-connected SDF Graphs

The fundamentals of SDF were introduced in terms of connected SDF graphs. In this section, we extend some basic principles of SDF to non-connected SDF graphs. We begin with a definition.

Definition 2.2: Suppose that S is a looped schedule for an SDF graph G and $Z \subseteq actors(G)$. If we remove from S all actors that are not in Z and then we repeatedly remove all null loops until no null loops remain, we obtain another looped schedule, which we call the **projection** of S onto Z, and which we denote by $projection(S, Z)$. For example,

$$projection((2(2B)(5A)), \{A, C\}) = (2(5A)) \quad ,$$

and $projection((5C), \{A, B\})$ is the null schedule. If G' is a subgraph of G, we define $projection(S, G') \equiv projection(S, actors(G'))$.

We will use the following fact, which follows immediately from Definition 2.2 and the definitions introduced in the previous section.

Fact 2.5: If S and S' are valid looped schedules for an SDF graph G, α is an edge in G, and i is a positive integer such that $1 \le i \le inv(sink(\alpha), S)$ and $1 \le i \le inv(sink(\alpha), S')$, then

(a). $(P(\alpha, i, S) = P(\alpha, i, S')) \Leftrightarrow (T(\alpha, i, S) = T(\alpha, i, S'))$;

(b). $P(\alpha, i, S) = P(\alpha, i, projection(S, \{source(\alpha), sink(\alpha)\}))$; and

(c). $max\_tokens(\alpha, S) = max(\{T(\alpha, i, S) | (1 \le i \le inv(sink(\alpha), S))\})$.

The projection of an admissable schedule S onto a subset of actors Z fully specifies the sequence of token populations occurring on each edge in the corresponding subgraph. More precisely, for any actor $A \in Z$, any positive integer i such that $1 \le i \le inv(A, S)$, and any input edge α of A contained in $subgraph(Z)$, the number of tokens queued on α just prior to the ith invocation

of A in S equals the number of tokens queued on α just prior to the ith invocation of A in an execution of $projection(S, Z)$. Thus, we have the following fact.

Fact 2.6: If S is a schedule for an SDF graph G, G' is a subgraph of G, and α is an edge in G', then

(a). S is valid (periodic) implies that $projection(S, G')$ is a valid (periodic) schedule for G'; and

(b). S terminates on α implies that $projection(S, G')$ terminates on α.

The concept of blocking factor does not apply directly to SDF graphs that are not connected. For example in Figure 2.8 the minimal numbers of repetitions

Figure 2.8. A simple non-connected SDF graph.

for a periodic schedule are given by $\rho(A, B, C, D) = (1, 1, 1, 1)^T$ [1]. The schedule $A(2C)B(2D)$ is a valid schedule for this example, but this schedule corresponds to a blocking factor of 1 for $subgraph(\{A, B\})$ and a blocking factor of 2 for $subgraph(\{C, D\})$ — there is no single *scalar* blocking factor associated with $A(2C)B(2D)$.

Now suppose that S is a valid schedule for an arbitrary SDF graph G. By Fact 2.6, for each connected component C of G, we have that $projection(S, C)$

1. Note that this vector is not a repetitions vector, and thus it is not represented by $\mathbf{q}$, because the associated graph is not connected. By definition, only connected SDF graphs can have repetitions vectors.

is a valid schedule for $subgraph(C, G)$. Thus, associated with S, there is a vector of positive integers $\mathbf{J}_S$, indexed by the connected components of G, such that

$$\text{for each connected component } C \text{ of } G,$$
$$A \in C \Rightarrow inv(A, S) = \mathbf{J}_S(C)\mathbf{q}_{subgraph(C)}(A) \quad . \tag{2-8}$$

We call $\mathbf{J}_S$ the **blocking vector** of S. For example, if $S = A(2C)B(2D)$ for Figure 2.8, then $\mathbf{J}_S(\{A, B\}) = 1$, and $\mathbf{J}_S(\{C, D\}) = 2$. On the other hand, if S is connected, then $\mathbf{J}_S$ has only one component, which is the blocking factor of S, $J(S)$. We refer to any vector of positive integers indexed by the connected components of G as a blocking vector for G.

It is often convenient to view a part of an SDF graph as a subsystem that is invoked as a single unit. The invocation of a subsystem corresponds to invoking a minimal valid schedule for the associated subgraph. If this subgraph is connected, its repetitions vector gives the minimum number of invocations required for a periodic schedule. However, if the subgraph is not connected, then the minimum number of invocations involved in a periodic schedule is not necessarily obtained by concatenating the repetitions vectors associated with the connected components of the subgraph. This is because the full SDF graph may contain connections between the non-connected components of the subgraph.

For example, let G denote the SDF graph in Figure 2.9(a) and consider the subsystem $subgraph(\{A, B, C, D\})$ in this graph. It is easily verified that $\mathbf{q}_G(A, B, C, D, E) = (2, 2, 4, 4, 1)^T$. Thus in a periodic schedule, the actors in $subgraph(\{C, D\})$ must be invoked twice as frequently as those in

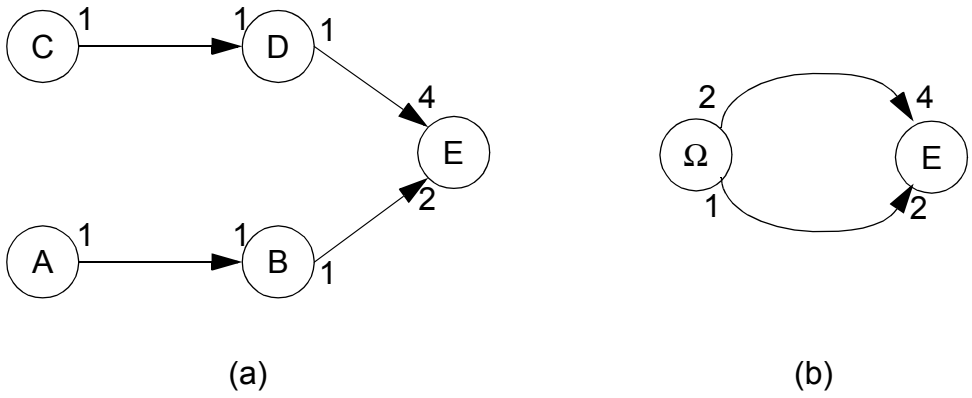


Figure 2.9. An example of clustering a subgraph in an SDF graph.

$subgraph(\{A, B\})$. We see that for a periodic schedule, the minimum numbers of repetitions for $subgraph(\{A, B, C, D\})$ *as a subgraph of* the original graph are given by $\rho(A, B, C, D) = (1, 1, 2, 2)^T$, which can be obtained by dividing each corresponding component in $\mathbf{q}_G$ by

$$gcd(\{\mathbf{q}_G(A), \mathbf{q}_G(B), \mathbf{q}_G(C), \mathbf{q}_G(D)\}) = 2 \quad .$$

On the other hand, concatenating the repetitions vectors of $subgraph(\{A, B\})$ and $subgraph(\{C, D\})$ yields the repetition counts $\rho'(A, B, C, D) = (1, 1, 1, 1)^T$. However, repeatedly invoking the subsystem with these relative repetition rates can never lead to a periodic schedule for G. We have motivated the following definition.

Definition 2.3: Let G be a connected SDF graph, suppose that Z is a subset of $actors(G)$, and let $R = subgraph(Z)$. We define

$q_G(Z) \equiv gcd(\{\mathbf{q}_G(A) \mid A \in Z\})$, and we define $\mathbf{q}_{R/G}$ to be the vector of positive integers indexed by the members of Z that is defined by $\mathbf{q}_{R/G}(A) \equiv \mathbf{q}_G(A)/q_G(Z)$, for each $A \in Z$. We can view $q_G(Z)$ as *the number of times a minimal periodic schedule for* G *invokes the subgraph* R, and we refer to $\mathbf{q}_{R/G}$ as the *repetitions vector of* R *as a subgraph of* G. For example, in Figure 2.9(a), if $R = subgraph(A, B, C, D)$, then $q_G(actors(R)) = 2$, and $\mathbf{q}_{R/G} = \mathbf{q}_{R/G}(A, B, C, D) = (1, 1, 2, 2)^T$.

The following fact establishes that for a connected SDF subgraph, its repetitions vector is the repetitions vector of itself as a subgraph of the enclosing graph.

Fact 2.7: If G is a connected SDF graph and R is a connected subgraph of G, then $\mathbf{q}_{R/G} = \mathbf{q}_R$. Thus, for a connected subgraph R, for each $A \in actors(R)$, $\mathbf{q}_G(A) = q_G(actors(R))\mathbf{q}_R(A)$.

Proof: Let S be any periodic schedule for G of unit blocking factor, and let $S' = projection(S, R)$. Then from Fact 2.6 and Fact 2.2, for all $A \in actors(R)$, we have $\mathbf{q}_G(A) = J(S')\mathbf{q}_R(A)$. From Fact 2.1, we know that the components of $\mathbf{q}_R$ are coprime, and it follows that

$$J(S') = gcd(\{\mathbf{q}_G(A') \mid A' \in actors(R)\}) = q_G(actors(R)) \quad .$$

Thus, for each actor A in R, $\mathbf{q}_R(A) = \mathbf{q}_G(A)/q_G(actors(R)) = \mathbf{q}_{R/G}(A)$. *QED.*

For example, in Figure 2.9(a), let $R = subgraph(\{A, B\})$. We have $\mathbf{q}_G(A, B, C, D, E) = (2, 2, 4, 4, 1)^T$, $\mathbf{q}_R(A, B) = (1, 1)^T$, and from Definition 2.3, $q_G(actors(R)) = gcd(2, 2) = 2$ and $\mathbf{q}_{R/G}(A, B) = (2, 2)^T/2 = (1, 1)^T$. As Fact 2.7 assures us, $\mathbf{q}_R = \mathbf{q}_{R/G}$.

Finally, we formalize the concept of **clustering** a subgraph of a connected SDF graph G, which as we discussed earlier, we use to organize hierarchy for scheduling purposes. This process is illustrated in Figure 2.9. Here $subgraph(\{A, B, C, D\})$ of Figure 2.9(a) is clustered into the hierarchical actor Ω, and the resulting SDF graph is shown in Figure 2.9(b). Each input edge α to a clustered subgraph R is replaced by an edge α' having

$$produced(\alpha') = produced(\alpha) \text{ , and}$$

$$consumed(\alpha') = consumed(\alpha) \times \mathbf{q}_{R/G}(sink(\alpha)) \text{ ,}$$

the number of tokens consumed from α in one invocation of R *as a subgraph of* G. Similarly, we replace each output edge β with β' such that

$$consumed(\beta') = consumed(\beta) \text{ , and}$$

$$produced(\beta') = produced(\beta) \times \mathbf{q}_{R/G}(source(\beta)) \text{ .}$$

We will use the following property of clustered subgraphs.

Fact 2.8: Suppose G is an SDF graph, R is a subgraph of G, G' is the SDF graph that results from clustering R into the hierarchical actor Ω, S' is a valid schedule for G', and S_R is a valid schedule for R such that for each actor A in R, $inv(A, S_R) = \mathbf{q}_{R/G}(A)$. Let $\hat{S}$ denote the schedule that results from replacing each instance of Ω in S' with S_R. Then $\hat{S}$ is a valid schedule for G.

As a simple example, consider Figure 2.9 again. Now, $(2\Omega)E$ is a valid schedule for the SDF graph in Figure 2.9(b), and $S = AB(2CD)$ is a valid schedule for $R = subgraph(\{A, B, C, D\})$ such that $(inv(A', S) = \mathbf{q}_{R/G}(A'))$, $\forall A'$. Thus Fact 2.8 guarantees that $(2AB(2CD))E$ is a valid schedule for Figure 2.9(a).

Proof of Fact 2.8: Given a schedule S and an SDF edge α, we define

$$\Delta(\alpha, S) \equiv (inv(source(\alpha), S) \times produced(\alpha)) - (inv(sink(\alpha), S) \times consumed(\alpha)) \quad . \tag{2-9}$$

Then S is periodic if and only if it invokes each actor and $(\Delta(\alpha, S) = 0)\forall\alpha$.

We can decompose S' into $s_1\Omega s_2\Omega\ldots s_{k-1}\Omega s_k$, where each s_j denotes the sequence of invocations between the $(j-1)$ th and j th invocations of Ω. Then $\hat{S} = s_1 S_R s_2 S_R \ldots S_R s_k$.

First, suppose that β is an edge in G such that $source(\beta), sink(\beta) \notin actors(R)$. Then S_R contains no occurrences of $source(\beta)$ nor $sink(\beta)$, so $P(\beta, i, \hat{S}) = P(\beta, i, S')$ for any invocation number i of $sink(\beta)$. Thus, since S' is admissable, $\hat{S}$ does not terminate on β. Also, $\Delta(\beta, \hat{S}) = \Delta(\beta, s_1 s_2 \ldots s_k) = \Delta(\beta, S') = 0$, since S' is periodic.

If $source(\beta), sink(\beta) \in actors(R)$, then none of the s_j's contain any occurrences of $source(\beta)$ or $sink(\beta)$. Thus, for any i, $P(\beta, i, \hat{S}) = P(\beta, i, \hat{S'})$, where $\hat{S'} = S_R S_R \ldots S_R$ denotes $\hat{S}$ with all of the s_j's removed. Since $\hat{S'}$ consists of successive invocations of a valid schedule, it fol-

lows that $\hat{S}$ does not terminate on β, and $\Delta(\beta, \hat{S}) = 0$.

Now suppose that $source(\beta) \in actors(R)$ and $sink(\beta) \notin actors(R)$. Then corresponding to β, there is an edge β' in G', such that $source(\beta') = \Omega$, $sink(\beta') = sink(\beta)$, $produced(\beta') = \mathbf{q}_{R/G}(source(\beta)) \times produced(\beta)$, and $consumed(\beta') = consumed(\beta)$. Now each invocation of S_R produces

$$inv(source(\beta), S_R) produced(\beta) = \mathbf{q}_{R/G}(source(\beta)) produced(\beta)$$

$$= produced(\beta')$$

tokens onto β. Since $consumed(\beta') = consumed(\beta)$ and S' is a valid schedule, it follows that $\Delta(\beta, \hat{S}) = 0$, and $\hat{S}$ does not terminate on β.

Similarly, if $source(\beta) \notin actors(R)$ and $sink(\beta) \in actors(R)$, we see that each invocation of S_R consumes the same number of tokens from β as Ω consumes from the corresponding edge in G', and thus $\Delta(\beta, \hat{S}) = 0$ and $\hat{S}$ does not terminate on β.

We conclude that $\hat{S}$ does not terminate on any edge in G, and thus, $\hat{S}$ is admissable. Furthermore, $\Delta(\alpha, \hat{S}) = 0$ for each edge α in G, and since S' and S_R are both periodic schedules, it is easily verified that $\hat{S}$ invokes each actor in G at least once, so we conclude that $\hat{S}$ is a periodic schedule. *QED*.

We conclude this section with a fact that relates the repetitions vector of an SDF graph obtained by clustering a subgraph to the repetitions vector of the original graph.

Fact 2.9: If G is a connected SDF graph, $Z \subseteq actors(G)$, and G' is the SDF graph obtained from G by clustering $subgraph(Z)$ into the actor Ω, then

$\mathbf{q}_{G'}(\Omega) = q_G(Z)$, and $\forall A \in (actors(G) - Z)$, $\mathbf{q}_{G'}(A) = \mathbf{q}_G(A)$.

Proof: Let $\mathbf{q}'$ denote the vector that we claim is the repetitions vector for G', and recall from Fact 2.1 that $\mathbf{q}' = \mathbf{q}_{G'}$ if and only if $\mathbf{q}'$ satisfies the balance equations for G' and the components of $\mathbf{q}'$ are coprime. From the definition of *clustering*, it can easily be verified that $\mathbf{q}'$ satisfies the balance equations for G'. Furthermore, from Fact 2.1, no positive integer greater than 1 can divide all members of

$$(\{\mathbf{q}_G(A) \mid (A \in actors(G) - Z)\} \cup \{gcd(\{\mathbf{q}_G(A) \mid (A \in Z)\})\}) \quad .$$

Since $\mathbf{q}'(\Omega) = gcd(\{\mathbf{q}_G(A) \mid (A \in Z)\})$, it follows that the components of $\mathbf{q}'$ are coprime. *QED.*

Fact 2.8 and Fact 2.9 imply that for scheduling purposes, a cluster in a connected SDF graph can be viewed as monolithic from the outside or as an SDF graph (possibly non-connected) from the inside, and that the SDF parameters of the monolith and the repetitions vector of the graph that it is contained in can be formally bound to the repetitions vector of the original SDF graph.

The concept of a cluster in a graph has been defined in and applied in many different contexts. In VLSI circuits, for example, a "cluster" is informally defined as a particularly dense or complex subcircuit, and the problem of detecting such clusters has been addressed to partition a circuit so that the number of connections crossing the partition are minimized [Garb90]. In multiprocessor scheduling, clustering is commonly used to group subsets of dataflow actors that are to be scheduled on the same processor [Gera92]. A third example arises in the context of dataflow/von Neumann hybrid architectures, which allow collections of data flow actors, called threads, to execute sequentially under the control of a program

counter, while the invocation of threads is carried out in a data-driven manner. Thus, the computation within a thread is performed in a von Neumann style, while the threads themselves are sequenced in a dataflow style. When compiling for a hybrid dataflow/von Neumann machine, clustering can be used to construct coarse-grain threads from a fine-grain dataflow representation of the program [Najj92].

2.4 Factoring Schedule Loops

In this section, we show that in a single appearance schedule, we can "factor" common terms from the iteration counts of inner loops into the iteration count of the enclosing loop. An important practical advantage of factoring is that it may significantly reduce the amount of memory required for buffering.

For example, consider the SDF graph in Figure 2.10. Here,

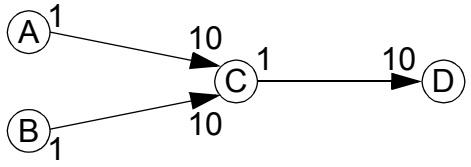


Figure 2.10. An SDF graph used to illustrate the factoring of loops.

$\mathbf{q}(A, B, C, D) = (100, 100, 10, 1)^T$, 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 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-startup 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 2.11 shows a simple SDF graph that can be used to demonstrate that unlike the factoring transformation, reverse factoring does not necessarily preserve the admissability 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. Further exploration of reverse factoring is beyond the scope of this thesis.

The factoring transformation is closely related to the **loop fusion** transformation, which has been used for decades in compilers for procedural languages. In the basic version of this transformation, two adjacent loops having the same iteration count are merged into a single loop by concatenating the bodies. It is well-known that loop fusion can reduce a program's memory requirements [Wolf89] just as the factoring transformation that we present in this section does. Also, loop fusion has been found to increase data locality, and hence to improve the exploitation of memory hierarchies [AbuS81]. In compilers for procedural languages, tests for the validity of loop fusion include analysis of array subscripts to determine whether or not for each iteration n of the (lexically) second loop, this iteration depends only on iterations $1, 2, \ldots n$ of the first loop [Zima90]. These tests are difficult to perform comprehensively due to the complexity of exact subscript analysis [Bane88], and due to complications such as data-dependent subscript values, conditional branches inside one or more of the loops, and input/output statements. In this section, we show that for single appearance schedules of SDF graphs, these

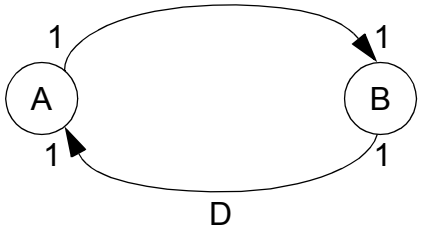


Figure 2.11. An example used to illustrate that reverse factoring is not always valid for single appearance schedules.

complications do not arise, and loop fusion, as well as our more general factoring transformation, is always a valid transformation. Thus, loop fusion is an additional example of the increased compile-time predictability that can be gained when restricting the computational model to SDF.

We will apply the following simple number-theoretic fact in the development of this section.

Fact 2.10: Suppose that x is a nonnegative integer, and a, b, and y are positive integers such that $x + ya \geq yb$. Then $\forall (i \in \{0, 1, \ldots, y\})$, $x + ia \geq ib$.

Proof: Suppose that $i \in \{0, 1, \ldots, y\}$, and first suppose that $a \leq b$. Then,

$$x + ya \geq yb \Rightarrow x \geq (b - a)y \Rightarrow x \geq (b - a)i \Rightarrow x + ia \geq ib \quad .$$

While, on the other hand if $a > b$, then $(b - a)i \leq 0$, and since $x \geq 0$, it follows that $(b - a)i \leq x$, and thus $x + ia \geq ib$. *QED.*

The following lemma establishes similarities between two looped schedules S and S' for a consistent SDF graph, where S' is obtained by replacing some subschedule S_0 in S with another schedule S_0' that invokes each actor the same number of times as S_0. For an SDF edge α, Lemma 2.1(a) states that if the mth invocation of $sink(\alpha)$ is not part of S_0' in S', then the number of invocations of $source(\alpha)$ that precede the mth invocation of $sink(\alpha)$ in S' equals the number of invocations of $source(\alpha)$ that precede the mth invocation of $sink(\alpha)$ in S; and Lemma 2.1(b) asserts the same conclusion whenever α is not contained in the subgraph associated with the set of actors invoked by S_0. We first state and prove the lemma, and present a corollary, and then we illustrate with an example.

Lemma 2.1: Suppose that G is a consistent SDF graph, S is a looped schedule for G, and S_0 is a subschedule of S. Suppose also that S_0' is any looped schedule such that $actors(S_0') = actors(S_0)$, and $inv(A, S_0') = inv(A, S_0)$, $\forall(A \in actors(S_0))$. Let S' denote the schedule obtained by replacing S_0 with S_0' in S. Then for any actor $N \in actors(G)$, any positive integer m such that $1 \le m \le inv(N, S)$, and any input edge α of N, we have

(a). If invocation N_m is not part of S_0' in S', then $P(\alpha, m, S') = P(\alpha, m, S)$; and

(b). If α is not contained in $subgraph(actors((S_0), G))$, then $P(\alpha, m, S') = P(\alpha, m, S)$.

Proof: The sequence of invocations in S can be decomposed into $s_1 b_1 s_2 b_2 \ldots b_n s_{n+1}$, where b_j denotes the sequence of invocations associated with the jth invocation of subschedule S_0, and s_j is the sequence of invocations between the $(j-1)$ th and jth invocations of S_0. Since S' is derived by rearranging the invocations in S_0, we can express S' similarly as $s_1 b_1' s_2 b_2' \ldots b_n' s_{n+1}$, where b_j' corresponds to the jth invocation of S_0' in S'.

For the proof of part (a), observe that N_m is part of some s_j, say s_k, in S'. Then $k-1$ invocations of S_0' precede N_m in S', and since for all j, $inv(N, b_j') = inv(N, b_j)$ we have that in S, N_m is also part of s_k. It follows that

$$P(\alpha, m, S) =$$
$$P(\alpha, m - (k-1)\,inv(N, S_0), s_1 s_2 \ldots s_k) + (k-1)\,inv(source(\alpha), S_0) \quad ,$$

and

$$P(\alpha, m, S') =$$

$$P(\alpha, m-(k-1)inv(N, S_0'), s_1 s_2 \ldots s_k) + (k-1)\, inv(source(\alpha), S_0') \quad .$$

But, by assumption, $inv(source(\alpha), S_0) = inv(source(\alpha), S_0')$, and $inv(N, S_0) = inv(N, S_0')$, so $P(\alpha, m, S) = P(\alpha, m, S')$, and the proof of part (a) is complete.

For the proof of part (b), first observe that if in S, N_m is part of one of the s_j's, then from part (a), we have immediately that $P(\alpha, m, S) = P(\alpha, m, S')$. On the other hand, if N_m is part of one of the b_j's, say b_p, in S, then $inv(N, S_0) = inv(N, S_0')$ implies that in S', N_m is part of b_p'. Also, by assumption α is not in $subgraph(actors((S_0), G))$, so since N_m is part of b_p, $N = sink(\alpha)$ is contained in $actors(S_0)$, and thus $source(\alpha) \notin actors(S_0)$. It follows that

$$P(\alpha, m, S) = inv(source(\alpha), s_1 s_2 \ldots s_p) = P(\alpha, m, S') \quad ,$$

and the proof of part (b) is complete. *QED*.

The following corollary follows immediately from Lemma 2.1. It implies that under the assumptions of Lemma 2.1 together with the additional assumption that S is admissable, if an invocation is not part of S_0' in S', then S' cannot terminate at that invocation, and if α is not contained in the subgraph associated with $actors(S_0)$, then S' cannot terminate on α.

Corollary 2.1: Assume the hypotheses of Lemma 2.1 with the additional assump-

tion that S is admissable. Then

(a). If invocation N_m is not part of S_0' in S', then S' does not terminate on α at N_m.

(b). If α is not contained in $subgraph(actors(S_0), G)$, then S' does not terminate on α.

Consider the example in Figure 2.12. Here, each d_j represents the number

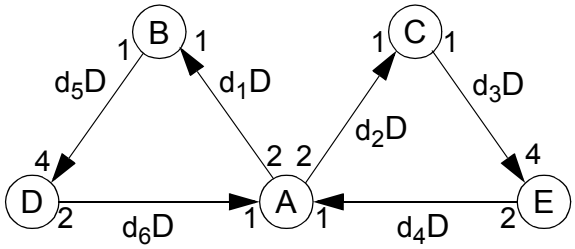


Figure 2.12. An example used to illustrate the application of Lemma 2.1.

of delays on the corresponding edge, and the repetitions vector is given by $\mathbf{q}(A, B, C, D, E) = (2, 4, 4, 1, 1)^T$. Suppose that the d_j's are such that $D(2A(2BC))E$ is an admissable schedule. Then Corollary 2.1(b) — with $S_0 = A(2BC)$ and $S_0' = BCABC$ — guarantees that the schedule $D(2BCABC)E$ does not terminate on the edges $B \rightarrow D$, $D \rightarrow A$, $E \rightarrow A$, or $C \rightarrow E$; and Corollary 2.1(a) insures that this schedule does not terminate at invocation D_1 or E_1.

The following lemma establishes a simple sufficient condition for valid application of the loop fusion transformation to a looped schedule. It states that

given a valid looped schedule, two adjacent loops having the same iteration count can be fused to yield another valid schedule if the sets of actors invoked by the fused loops are mutually disjoint.

Lemma 2.2: Suppose that S is a valid schedule for an SDF graph G, and suppose that S contains a subschedule $S_0 = L_1 L_2$, where $L_1 = (nB_1)$ and $L_2 = (nB_2)$ are two schedule loops having identical iteration counts and arbitrary bodies B_1 and B_2, respectively. Assume also that $actors(B_1) \cap actors(B_2) = \varnothing$. Then replacing S_0 with the schedule loop $S_0' = (nB_1B_2)$ in S results in a valid schedule for G.

As a counter-example that illustrates the need for the assumption that $actors(B_1) \cap actors(B_2) = \varnothing$, consider the SDF graph in Figure 2.13. One can

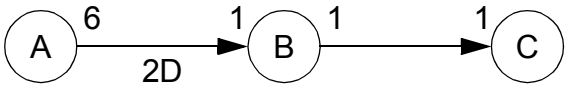


Figure 2.13. An SDF graph used to illustrate that the fusion of two adjacent schedule loops in a valid looped schedule is not always a legitimate transformation

easily verify that the looped schedule $A(2B)(2CCBB)CC$ is a valid schedule for this SDF graph. Observe that although the two schedule loops in this schedule have a common iteration count, they both contain instances of the actor B, and thus these loops do not satisfy the hypotheses of Lemma 2.2. If we fuse these two loops, we obtain the schedule $A(2BCCBB)CC$. The invocation sequence generated by this new schedule, $ABCCBBBCCBBCC$, terminates on the edge $B \rightarrow C$

at invocation C_2. Thus, the fusion of the schedule loops $(2B)$ and $(2CCBB)$ converts a valid schedule into a schedule that is not valid.

Proof of Lemma 2.2: Let S' denote the schedule that results from replacing S_0 with S_0' in S. By construction of S', we have that for all actors A in G, $inv(A, S') = inv(A, S)$. Since S is valid, and hence periodic, it follows that S' is also periodic. It remains to be shown that S' is admissable.

Clearly S' is admissable if for each edge α in G, S' does not terminate on α. There are four cases to consider here:

1. $(source(\alpha), sink(\alpha) \in actors(B_1))$ **or** $(source(\alpha), sink(\alpha) \in actors(B_2))$

2. $(source(\alpha) \notin (actors(B_1) \cup actors(B_2)))$ **or** $(sink(\alpha) \notin (actors(B_1) \cup actors(B_2)))$

3. $(source(\alpha) \in actors(B_1))$ **and** $(sink(\alpha) \in actors(B_2))$

4. $(source(\alpha) \in actors(B_2))$ **and** $(sink(\alpha) \in actors(B_1))$

Case 1:

$(source(\alpha), sink(\alpha) \in actors(B_1))$ **or** $(source(\alpha), sink(\alpha) \in actors(B_2))$

.

Let i be that member of $\{1, 2\}$ such that $source(\alpha), sink(\alpha) \in actors(B_i)$, and observe that since $actors(B_1) \cap actors(B_2) = \varnothing$,

$projection(S_0, actors(B_i)) = (nB_i) = projection(S_0', actors(B_i))$, and thus

$$projection(S', actors(B_i)) = projection(S, actors(B_i)) \quad . \qquad (2\text{-}10)$$

Now, (S' terminates on α)

$\Rightarrow$($projection(S', actors(B_i))$ terminates on α) (by Fact 2.6b)

$\Rightarrow$($projection(S, actors(B_i))$ terminates on α) (by 2-10)

$\Rightarrow$(S is not valid) (by Fact 2.6a).

By contraposition, it follows that under the assumptions of Lemma 2.2, S' does not terminate on α.

Case 2: $(source(\alpha) \notin (actors(B_1) \cup actors(B_2)))$ **or** $(sink(\alpha) \notin (actors(B_1) \cup actors(B_2)))$.

By Corollary 2.1(b), S' does not terminate on α.

Case 3: $(source(\alpha) \in actors(B_1))$ **and** $(sink(\alpha) \in actors(B_2))$.

Let r be any positive integer such that $1 \le r \le inv(S_0, S)$; let t_r denote the number of tokens on α just prior to the rth invocation of S_0 in S; and observe that since S is admissable, we must have

$$t_r + n(inv(source(\alpha), B_1))produced(\alpha)$$

$$\ge n(inv(sink(\alpha), B_2)consumed(\alpha)) \quad . \tag{2-11}$$

Now, by construction of S', we have that

$$\text{for all actors } A \in actors(G), \quad inv(A, S_0') = inv(A, S_0) \quad . \tag{2-12}$$

Thus, the number of tokens on α just prior to the rth invocation of S_0' in S' is equal to t_r, and S' does not terminate on α during the rth invocation of S_0 if there is a sufficient number of tokens on α prior to each of the n invocations of

B_2 — that is, if

$$\forall(k \in \{1, 2, \ldots, n\})$$

$$t_r + k\,inv(source(\alpha), B_1)produced(\alpha) - (k-1)\,inv(sink(\alpha), B_2)consumed(\alpha)$$

$$\geq inv(sink(\alpha), B_2)consumed(\alpha) \quad .$$

which is equivalent to

$$\forall(k \in \{1, 2, \ldots, n\}), t_r + k(inv(source(\alpha), B_1))produced(\alpha)$$

$$\geq k(inv(sink(\alpha), B_2))consumed(\alpha) \quad . \qquad (2\text{-}13)$$

By Fact 2.10, (2-13) is guaranteed by (2-11), and since (2-11) holds for all invocation numbers r, it follows that S' does not terminate on α during an invocation of S_0'. Furthermore, from Corollary 2.1(a), S' cannot terminate at an invocation that is not part of S_0'. We conclude that S' does not terminate on α.

Case 4: $(source(\alpha) \in actors(B_2))$ **and** $(sink(\alpha) \in actors(B_1))$.

Again, let r be any positive integer such that $1 \leq r \leq inv(S_0, S)$, and let t_r denote the number of tokens on α just prior to the rth invocation of S_0 in S. Then since S is admissable,

$$t_r \geq n(inv(sink(\alpha), B_1))consumed(\alpha) \quad . \qquad (2\text{-}14)$$

Now clearly, S' does not terminate on α during the rth invocation of S_0' if

$\forall(k \in \{1, 2, \ldots, n\})$,

$$t_r + (k-1)\,inv(source(\alpha), B_2)produced(\alpha) - (k-1)\,inv(sink(\alpha), B_1)consumed(\alpha)$$

$\geq inv(sink(\alpha), B_1) consumed(\alpha)$,

which is equivalent to

$$\forall (k \in \{1, 2, \ldots, n\}), t_r + (k-1) inv(source(\alpha), B_2) produced(\alpha)$$
$$\geq k\, inv(sink(\alpha), B_1) consumed(\alpha) \quad . \tag{2-15}$$

Now, it is easily seen that (2-14) implies (2-15). Since this analysis holds for any choice of r, S' does not terminate on α during an invocation of S_0', and from Corollary 2.1(a), S' cannot terminate at an invocation that is not part of S_0', so we conclude that S' does not terminate on α.

Our treatment of cases 1-4 shows that for any edge α contained in G, S' does not terminate on α. *QED*.

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 2.1 is that factoring cannot convert a valid single appearance schedule into a schedule that is not valid.

Theorem 2.1: Suppose that S is a valid schedule for an SDF graph G, and suppose that $L = (m(n_1 S_1)(n_2 S_2)\ldots(n_k S_k))$ is a schedule loop in S of any nesting depth such that $(1 \leq i < j \leq k) \Rightarrow actors(S_i) \cap actors(S_j) = \varnothing$. Suppose also that γ is any positive integer that divides $n_1, n_2, \ldots, n_k$, and let L' denote the schedule loop $(\gamma m(\gamma^{-1} n_1 S_1)(\gamma^{-1} n_2 S_2)\ldots(\gamma^{-1} n_k S_k))$. Then the schedule that

results from replacing L with L' in S is a valid schedule for G.

Proof: We will prove this theorem by induction on k.

First, let S' denote the schedule that results from replacing L with L' in S, and observe that for $k = 1$, L and L' generate the same invocation sequence, and thus S and S' generate the same invocation sequence. We conclude that S' is valid for $k = 1$, and thus Theorem 2.1 holds for $k = 1$.

Second, consider the case $k = 2$. Then $L = (m(n_1S_1)(n_2S_2))$ and $L' = (\gamma m(\gamma^{-1}n_1S_1)(\gamma^{-1}n_2S_2))$. Now, observe that L generates the same invocation sequence as the schedule loop $\hat{L} = (m(\gamma(\gamma^{-1}n_1S_1))(\gamma(\gamma^{-1}n_2S_2)))$, so replacing L with $\hat{L}$ in the valid schedule S yields another valid schedule $\hat{S}$. Since, by assumption $actors(S_1) \cap actors(S_2) = \varnothing$, Lemma 2.2 guarantees that replacing $\hat{L}$ with $\hat{L}' = (m(\gamma(\gamma^{-1}n_1S_1)(\gamma^{-1}n_2S_2)))$ in $\hat{S}$ yields a third valid schedule $\hat{S}'$. But, clearly $\hat{L}'$ and L' generate the same invocation sequence, so replacing $\hat{L}'$ with L' in $\hat{S}'$ results in a valid schedule $\hat{S}''$. But by our construction, $\hat{S}'' = S'$, and thus S' is a valid schedule for G. We conclude that Theorem 2.1 holds for $k = 2$.

Now suppose that Theorem 2.1 holds whenever $k \le k'$, for some $k' \ge 2$. We will show that this implies the validity of Theorem 2.1 for $k \le (k' + 1)$. For $k = k' + 1$,

$$L = (m(n_1S_1)(n_2S_2)\ldots(n_{k'+1}S_{k'+1})), \text{ and}$$

$$L' = (\gamma m(\gamma^{-1}n_1S_1)(\gamma^{-1}n_2S_2)\ldots(\gamma^{-1}n_{k'+1}S_{k'+1})).$$

Let S_a denote the schedule that results from replacing L with the schedule loop $L_a = (m(1(n_1S_1)(n_2S_2)\ldots(n_{k'}S_{k'}))(n_{k'+1}S_{k'+1}))$ in S. Since L_a and L generate the same invocation sequence, S_a generates the same invocation sequence as S. Now Theorem 2.1 for $k = k'$ guarantees that replacing $(1(n_1S_1)(n_2S_2)\ldots(n_{k'}S_{k'}))$ with $(\gamma(\gamma^{-1}n_1S_1)(\gamma^{-1}n_2S_2)\ldots(\gamma^{-1}n_{k'}S_{k'}))$ in S_a results in a valid schedule S_b.

Observe that S_b is the schedule S with L replaced by

$$L_b = (m(\gamma(\gamma^{-1}n_1S_1)(\gamma^{-1}n_2S_2)\ldots(\gamma^{-1}n_{k'}S_{k'}))(n_{k'+1}S_{k'+1})) \quad .$$

Theorem 2.1 for $k = 2$ guarantees that replacing L_b with

$$L_c = (\gamma m(1(\gamma^{-1}n_1S_1)(\gamma^{-1}n_2S_2)\ldots(\gamma^{-1}n_{k'}S_{k'}))(\gamma^{-1}n_{k'+1}S_{k'+1}))$$

yields another valid schedule S_c. Now clearly L_c generates the same invocation sequence as L', so replacing L_c with L' in S_c yields a valid schedule S_d. But, by our construction, $S_d = S'$, so S' is a valid schedule for G.

We have shown that Theorem 2.1 holds for $k = 1$ and $k = 2$, and we have shown that if the result holds for $k \le k'$, then it holds $k \le (k' + 1)$. We conclude that Theorem 2.1 holds for all k. *QED*.

We have demonstrated that factoring may decrease the buffer memory requirement for a schedule. Although the transformation is not guaranteed to always decrease the buffer memory requirement, factoring never increases the buffer memory requirement. This is established by the following theorem.

Theorem 2.2: As in Theorem 2.1, assume that S is a valid schedule for an SDF

graph G; $L = (m(n_1S_1)(n_2S_2)\ldots(n_kS_k))$ is a schedule loop in S of any nesting depth such that $(1 \le i < j \le k) \Rightarrow actors(S_i) \cap actors(S_j) = \varnothing$; and γ is a positive integer that divides $n_1, n_2, \ldots, n_k$. Let L' denote the schedule loop $(\gamma m(\gamma^{-1}n_1S_1)(\gamma^{-1}n_2S_2)\ldots(\gamma^{-1}n_kS_k))$, and let S' denote the schedule that results from replacing L with L' in S. Then $buffer\_memory(S') \le buffer\_memory(S)$.

Proof: We show that for each edge α in G, $max\_tokens(\alpha, S') \le max\_tokens(\alpha, S)$, which clearly implies the desired result. We consider three cases.

Case 1: $(source(\alpha) \notin actors(L))$ **or** $(sink(\alpha) \notin actors(L))$. From Lemma 2.1(b), we have that

$$\forall(i \in \{1, 2, \ldots, inv(sink(\alpha), S)\}), P(\alpha, i, S') = P(\alpha, i, S),$$

and from Fact 2.5(a), it follows that

$$\forall(i \in \{1, 2, \ldots, inv(sink(\alpha), S)\}), T(\alpha, i, S') = T(\alpha, i, S).$$

Thus, from Fact 2.5(c), we have $max\_tokens(\alpha, S') = max\_tokens(\alpha, S)$.

Case 2: For some $j \in \{1, 2, \ldots, k\}$, $source(\alpha), sink(\alpha) \in actors(S_j)$. Then since $actors(S_1) \cap actors(S_2) \cap \ldots \cap actors(S_k) = \varnothing$, it is easily verified from the construction of S', that $projection(S, \{source(\alpha), sink(\alpha)\})$ generates the same invocation sequence as $projection(S', \{source(\alpha), sink(\alpha)\})$. From Fact 2.5(b) and Fact 2.5(a), $T(\alpha, i, S') = T(\alpha, i, S)$, and thus $max\_tokens(\alpha, S') = max\_tokens(\alpha, S)$.

Case 3: $(source(\alpha) \in actors(S_p))$ **and** $(sink(\alpha) \in actors(S_q))$,

where $p, q \in \{1, 2, \ldots, k\}$ and $p \neq q$. We define

$$r_1 \equiv max(\{T(\alpha, i, S) | sink(\alpha)_i \text{ is part of } L\}) \quad ;$$

$$r_1' \equiv max(\{T(\alpha, i, S') | sink(\alpha)_i \text{ is part of } L'\}) \quad ;$$

$$r_2 \equiv max(\{T(\alpha, i, S) | sink(\alpha)_i \text{ is not part of } L\}) \quad ;$$

$$r_2' \equiv max(\{T(\alpha, i, S') | sink(\alpha)_i \text{ is not part of } L'\}) \quad .$$

Then clearly,

$$max\_tokens(\alpha, S) = max(\{r_1, r_2\}) \quad \text{and}$$

$$max\_tokens(\alpha, S') = max(\{r_1', r_2'\}) \quad . \tag{2-16}$$

Now, if in S, $sink(\alpha)_i$ is not part of L, then clearly by the construction of L', $sink(\alpha)_i$ is not part of L' in S', and from Lemma 2.1(a) and Fact 2.5(a), we have that $T(\alpha, i, S') = T(\alpha, i, S)$, and thus

$$r_2 = r_2' . \tag{2-17}$$

On the other hand, if $sink(\alpha)_i$ is part of L in S, and thus $sink(\alpha)_i$ is part of L' in S', we define

$$\Delta \equiv (n_p \times inv(source(\alpha), S_p) \times produced(\alpha)) - \quad ;$$
$$(n_q \times inv(sink(\alpha), S_q) \times consumed(\alpha))$$

$M(\alpha, j) \equiv max(\{T(\alpha, i, S) | sink(\alpha)_i$ is part of the jth invocation of $L\})$; and

$M'(\alpha, j) \equiv max(\{T(\alpha, i, S') | sink(\alpha)_i$ is part of the jth invocation of $L'\})$.

Also, we define x_j to denote the number of tokens on α just before the j th invocation of L (L') in S (S'), and we define y_j to denote the number of tokens on α just after the j th invocation of L (L') in S (S').

Clearly, if $\Delta \geq 0$, then during a particular invocation of L in an execution of S, the maximum number of tokens on α is attained just after the last invocation of $(n_p S_p)$. Similarly in an execution of S', the maximum number of tokens on α during the jth invocation of L' is attained just after the last invocation of $(\gamma^{-1} n_p S_p)$. Thus, if $p < q$, then

$$M(\alpha, j) = x_j + m n_p inv(source(\alpha), S_p) produced(\alpha) - (m-1) n_q inv(sink(\alpha), S_q) consumed(\alpha)$$

$$= x_j + m\Delta + n_q inv(sink(\alpha), S_q) consumed(\alpha) \quad ,$$

and similarly,

$$M'(\alpha, j) = x_j + (\gamma m) \frac{n_p}{\gamma} inv(source(\alpha), S_p) produced(\alpha) - (\gamma m - 1) \frac{n_q}{\gamma} inv(sink(\alpha), S_q) consumed(\alpha)$$

$$= x_j + m\Delta + \frac{n_q}{\gamma} inv(sink(\alpha), S_q) consumed(\alpha) \quad .$$

Thus, since $\gamma \geq 1$, we have that $M'(\alpha, j) \leq M(\alpha, j)$, and since this holds for all j, r_1' cannot exceed r_1. From (2-16) and (2-17), it follows that

$max\_tokens(\alpha, S') \le max\_tokens(\alpha, S)$.

If $(\Delta \ge 0)$ and $(p > q)$, then clearly y_j cannot be less than $M(\alpha, j)$ nor $M'(\alpha, j)$ for any j . Thus,

$$max\_tokens(\alpha, S) = max(\{r_2, y_1, y_2, \ldots, y_{inv(L, S)}\}) = max\_tokens(\alpha, S') \quad .$$

If $(\Delta < 0)$ and $(p < q)$, then for any j ,

$$M(\alpha, j) = x_j + n_p inv(source(\alpha), S_p) produced(\alpha) \quad \text{, and}$$

$$M'(\alpha, j) = x_j + \frac{n_p}{\gamma} inv(source(\alpha), S_p) produced(\alpha) \quad .$$

Thus, $M'(\alpha, j) \le M(\alpha, j)$ for all j , and we have $r_1' \le r_1$. From (2-16) and (2-17), we conclude that $max\_tokens(\alpha, S') \le max\_tokens(\alpha, S)$.

Finally, if $(\Delta < 0)$ and $(p > q)$, then clearly $M(\alpha, j) = M'(\alpha, j) = x_j$ for all j .Thus $r_1' = r_1$, and (2-16) and (2-17) yield that $max\_tokens(\alpha, S') = max\_tokens(\alpha, S)$.

Any edge α in G must fall into the domain of case 1 , case 2 or case 3 , and in each of these cases, we have established that $max\_tokens(\alpha, S') \le max\_tokens(\alpha, S)$. *QED*.

Recall that our definition of *buffer memory requirement* assumes that each buffer is implemented as a separate, contiguous block of storage, and thus Theorem 2.2 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 Chapter 4, we will discuss shared buffers and buffers that do not necessarily reside in contiguous memory locations.

2.5 Reduced Single Appearance Schedules

Definition 2.4: Suppose that Λ is either a schedule loop or a looped schedule. We say that Λ is **non-coprime** if all iterands of Λ are schedule loops and there exists an integer $j > 1$ that divides all of the iteration counts of the iterands of Λ. If Λ is not non-coprime, we say that Λ is **coprime**.

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 our 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 2.5: 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.

In this section, we show 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. First, however, we show that any fully reduced schedule has unit blocking factor. 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.

Theorem 2.3: Suppose that G is a connected SDF graph and S is a valid fully

reduced single appearance schedule for G. Then $J(S) = 1$.

Proof: First, suppose that not all iterands of S are schedule loops. Then some actor A is an iterand of S. Since A is not enclosed by a loop in S, and since S is a single appearance schedule, $inv(A, S) = 1$, and thus $J(S) = 1$.

Now suppose that all iterands of S are schedule loops and suppose that j is an arbitrary integer that is greater than one. Then since S is fully reduced, j does not divide at least one of the iteration counts associated with the iterands of S. Define $i_0 = 1$ and let L_1 denote one of the iterands of S whose iteration count i_1 is not divisible by $j = j/gcd(\{j, i_0\})$. Again, since S is fully reduced, if all iterands of L_1 are schedule loops, then there exists and iterand L_2 of L_1 such that $j/gcd(\{j, i_0 i_1\})$ does not divide the iteration count i_2 of L_2. Similarly, if all iterands of L_2 are schedule loops, there exists an iterand L_3 of L_2 whose iteration count i_3 is not divisible by $j/gcd(\{j, i_0 i_1 i_2\})$.

Continuing in this manner, we generate a sequence $L_1, L_2, L_3, \ldots$ such that the iteration count i_k of each L_k is not divisible by $j/(gcd(\{j, i_0 i_1 \ldots i_{k-1}\}))$. Since G contains a finite number of actors, we cannot continue this process indefinitely— for some $m \geq 1$, not all iterands of L_m are schedule loops. Thus, there is an actor A that is an iterand of L_m. Since S is a single appearance schedule,

$inv(A, S)$

$$= inv(L_1, S)\,inv(L_2, L_1)\,inv(L_3, L_2)\ldots inv(L_m, L_{m-1})\,inv(A, L_m)$$

$$= i_0 i_1 \ldots i_m . \qquad (2\text{-}18)$$

By our selection of the L_k's, $j \not/ (gcd(\{j, i_0 i_1 \ldots i_{m-1}\}))$ does not divide i_m, and thus from (2-18), j does not divide $inv(A, S)$.

We have shown that given any integer $j > 1$, there exists an actor A in G, such that $inv(A, S)$ is not divisible by j. If follows that the blocking factor of S is one. *QED*.

Theorem 2.4: Suppose that G is a consistent SDF graph and S is a valid single appearance schedule for G. Then there exists a valid single appearance schedule S' for G such that S' is fully reduced and $buffer\_memory(S') \leq buffer\_memory(S)$.

Proof: We prove this theorem by construction. This construction process can easily be automated to yield an efficient algorithm for synthesizing a valid fully reduced schedule from an arbitrary valid single appearance schedule.

Given a looped schedule Ψ, we denote the set of schedule loops in Ψ that are not coprime by $non\text{-}coprime(\Psi)$. Now suppose that S is a valid single appearance schedule for G, and let $\lambda_1 = (m(n_1\Psi_1)(n_2\Psi_2)\ldots(n_k\Psi_k))$ be any innermost member of $non\text{-}coprime(S)$ — that is, λ_1 is non-coprime, but every schedule loop nested within λ_1 is coprime. From Theorem 2.1, replacing λ_1 with

$\lambda_1' = (\gamma m(\gamma^{-1} n_1\Psi_1)(\gamma^{-1} n_2\Psi_2)\ldots(\gamma^{-1} n_k\Psi_k))$, where

$\gamma = gcd(\{n_1, n_2, \ldots, n_k\})$, yields another valid single appearance schedule S_1, and from Theorem 2.2, $buffer\_memory(S_1) \leq buffer\_memory(S)$. Furthermore, λ_1' is coprime, and since every schedule loop nested within λ_1 is coprime, every loop nested within λ_1' is coprime as well. Now let λ_2 be any innermost member of $non\text{-}coprime(S_1)$, and observe that λ_2 cannot equal λ_1'. Theorem 2.1 guarantees a replacement λ_2' for λ_2 in S_1 that leads to another valid single appearance schedule S_2, and Theorem 2.2 guarantees that $buffer\_memory(S_2) \leq buffer\_memory(S)$. If we continue this process, it is clear that no replacement loop λ_k' ever replaces one of the previous replacement loops $\lambda_1', \lambda_2', \ldots, \lambda_{k-1}'$, since these loops and the loops nested within these loops are already coprime. Also, no replacement changes the total number of schedule loops in the schedule. It follows that we can continue this process only a finite number of times — eventually, we will arrive at an S_n such that $non\text{-}coprime(S_n)$ is empty.

Now if S_n is a coprime looped schedule, we are done. Otherwise, S_n is of the form $(p_1T_1)(p_2T_2)\ldots(p_mT_m)$, where $\gamma' \equiv gcd(\{p_1, p_2, \ldots, p_m\}) > 1$. Applying Theorem 2.1 to the schedule $(1S_n) = (1(p_1T_1)(p_2T_2)\ldots(p_mT_m))$, we have that

$$(\gamma'((\gamma')^{-1}p_1T_1)((\gamma')^{-1}p_2T_2)\ldots((\gamma')^{-1}p_mT_m))$$

is a valid schedule for G. From the definition of a valid schedule, it follows that

$$S_n' \equiv ((\gamma')^{-1}p_1T_1)((\gamma')^{-1}p_2T_2)\ldots((\gamma')^{-1}p_mT_m)$$

is also a valid schedule, and by our construction of S_n and S_n', S_n' is a coprime single appearance schedule, and all schedule loops in S_n' are coprime. Thus, S_n' is a valid fully reduced single appearance schedule for G. Furthermore, since $(1S_n)$ generates the same invocation sequence as S_n clearly $buffer\_memory((1S_n)) = buffer\_memory(S_n)$. From Theorem *2.2*, $buffer\_memory(S_n') \leq buffer\_memory((1S_n))$, and thus $buffer\_memory(S_n') \leq buffer\_memory(S)$. *QED*.

2.6 Subindependence

Since valid single appearance schedules implement the full repetition inherent in an SDF graph without requiring subroutines or code duplication, we examine the topological conditions required for such schedules to exist. First, suppose that G is a connected, consistent acyclic SDF graph containing n actors. Then we can take some root actor R_1 of G and fire all $\mathbf{q}_G(R_1)$ invocations of R_1 in succession. After all invocations of R_1 have fired, we can remove R_1 from G, pick a root actor R_2 of the new acyclic SDF graph, and schedule its $\mathbf{q}_G(R_2)$ repetitions in succession. Clearly, we can repeat this process until no actors are left to obtain the single appearance schedule

$$(\mathbf{q}_G(R_1)R_1)(\mathbf{q}_G(R_2)R_2)\ldots(\mathbf{q}_G(R_n)R_n)$$

for G. Thus, we see that any consistent acyclic SDF graph has a valid single appearance schedule.

Also, observe that if G is an arbitrary connected, consistent SDF graph,

then we can cluster the subgraph associated with each nontrivial strongly connected component of G. Clustering a strongly connected component into a single actor Ω never results in deadlock since there can be no cycle containing Ω. Since clustering all strongly connected components yields an acyclic graph, it follows from Fact 2.6 and Fact 2.8 that *G has a valid single appearance schedule if and only if each strongly connected component has a valid single appearance schedule*.

Observe that we must, in general, analyze a strongly connected component subgraph Θ as a separate entity since G may have a valid single appearance schedule even if there is an actor A in Θ for which we cannot fire all $\mathbf{q}_G(A)$ invocations in succession. The key is that $\mathbf{q}_\Theta(A)$ may be less than $\mathbf{q}_G(A)$, so we may be able to generate a single appearance subschedule for Θ; for example, we may be able to schedule A $\mathbf{q}_\Theta(A)$ times in succession. Since we can schedule G so that the subschedule for Θ appears only once, this will translate into a single appearance schedule for G. For example, in Figure 2.14(a), it can be verified that

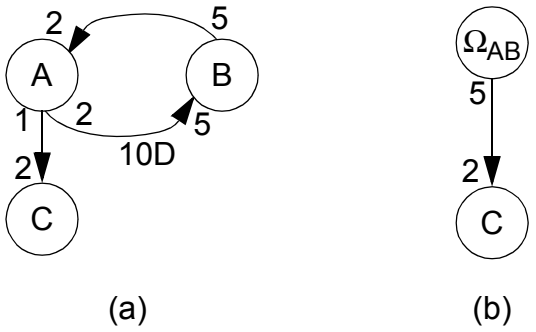


Figure 2.14. An example of how clustering strongly connected components can aid in generating compact looped schedules.

$\mathbf{q}(A, B, C) = (10, 4, 5)^T$, but we cannot fire so many invocations of A , B , nor C in succession. However, consider the strongly connected component subgraph $\Theta' \equiv subgraph(\{A, B\})$. Then we obtain $\mathbf{q}_{\Theta'}(A) = 5$ and $\mathbf{q}_{\Theta'}(B) = 2$, and we immediately see that $\mathbf{q}_{\Theta'}(B)$ invocations of B can be fired in succession to yield a subschedule for Θ'. The SDF graph that results from clustering Θ' is shown in Figure 2.14(b). This leads to the valid single appearance schedule $(2(2B)(5A))(5C)$.

Theorem 2.5: Suppose that G is a connected SDF graph and suppose that G has a valid single appearance schedule for some arbitrary blocking factor. Then G has valid single appearance schedules for all blocking factors.

Proof: Clearly, any valid schedule S of unity blocking factor can be converted into a valid schedule of arbitrary blocking factor j simply by encapsulating S inside a schedule loop having iteration count j. Thus, it suffices to show that G has a valid single appearance schedule of unity blocking factor. Now, Theorem 2.4 guarantees that G has a valid fully reduced single appearance schedule, and Theorem 2.3 guarantees that the blocking factor of this schedule is unity. *QED*.

Corollary 2.2: Suppose that G is an SDF graph that has a valid single appearance schedule (G need not be connected). Then G has a valid single appearance schedule for all blocking vectors.

Proof: Suppose that S is a valid single appearance schedule for G, let $\kappa_1, \kappa_2, \ldots, \kappa_n$ denote the connected components of G, let

$\mathbf{J}'(\kappa_1, \kappa_2, \ldots, \kappa_n) \equiv (z_1, z_2, \ldots, z_n)$ be an arbitrary blocking vector for G, and for $1 \le i \le n$, let S_i denote the projection of S onto κ_i. Then from Fact 2.6, each S_i is a valid single appearance schedule for the corresponding $subgraph(\kappa_i)$. From Theorem 2.5, for $1 \le i \le n$, there exists a valid single appearance schedule S_i' of blocking factor z_i for $subgraph(\kappa_i, G)$. Since the κ_i's are mutually disjoint and non-adjacent, it follows that $S_1'S_2'\ldots S_n'$ is a valid single appearance schedule of blocking vector $\mathbf{J}'$ for G. *QED*.

The condition for the existence of a valid single appearance schedule can be expressed in terms of a form of precedence independence, which is specified in the following definition.

Definition 2.6: 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 $source(\alpha) \in Z_2$ and $sink(\alpha) \in Z_1$, we have $delay(\alpha) \ge total\_consumed(\alpha, G)$. 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.

In other words, Z_1 is subindependent of Z_2 if given a minimal periodic schedule for G, data produced by Z_2 is never consumed by Z_1 in the same schedule period in which it is produced. Thus, at the beginning of each schedule period,

all of the data required by Z_1 from Z_2 for that schedule period is available at the inputs of Z_1. For example, let G denote the SDF graph in Figure 2.15. Here

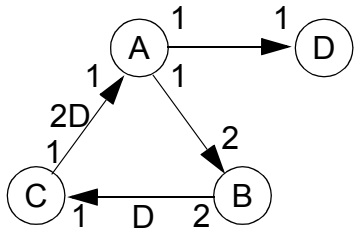


Figure 2.15. An example used to illustrate the concept of subindependence.

$\mathbf{q}(A, B, C, D) = (2, 1, 2, 2)^T$, and we see that $\{A\}$ is subindependent of $\{C\}$; $\{A, D\}$ and $\{B, C\}$ form a subindependent partition of G; and trivially, $\{A, B, C\}$ is subindependent of $\{D\}$.

The following properties of subindependence follow immediately from Definition 2.6.

Fact 2.11: Suppose that G is a connected, sample rate consistent SDF graph, and X, Y and Z are disjoint, nonempty subsets of $actors(G)$. Then

(a). (X is subindependent of Z) **and** (Y is subindependent of Z) $\Rightarrow$ ($X \cup Y$) is subindependent of Z

(b). (X is subindependent of Y) **and** (X is subindependent of Z) $\Rightarrow X$ is subindependent of $(Y \cup Z)$

Recall that an arbitrary consistent SDF graph has a valid single appearance schedule if and only if each strongly connected component has a single appearance schedule. The following theorem gives necessary and sufficient conditions for a strongly connected SDF graph to have a valid single appearance schedule.

Theorem 2.6: Suppose that G is a nontrivial, consistent, strongly connected SDF graph. Then G has a valid single appearance schedule if and only if there exists a nonempty proper subset $X \subset actors(G)$ such that

(1). X is subindependent of $(actors(G) - X)$ in G; and

(2). $subgraph(X, G)$ and $subgraph(actors(G) - X, G)$ both have valid single appearance schedules.

Proof: ($\Leftarrow$ direction). Let S and T denote valid single appearance schedules for $Y \equiv subgraph(X, G)$ and $Z \equiv subgraph((actors(G) - X), G)$, respectively; let $y_1, y_2, \ldots, y_k$ denote the connected components of Y; and let $z_1, z_2, \ldots, z_l$ denote the connected components of Z. From Corollary 2.2, we can assume without loss of generality that for $1 \le i \le k$, $\mathbf{J}_S(y_i) = q_G(y_i)$, and that for $1 \le i \le l$, $\mathbf{J}_T(z_i) = q_G(z_i)$. From Fact 2.7, it follows that S invokes each $A \in X$ $\mathbf{q}_G(A)$ times, and T invokes each $A \in (actors(G) - X)$ $\mathbf{q}_G(A)$ times, and since X is subindependent in G, it follows that ST, the schedule obtained by appending T to S, is a valid single appearance schedule (of blocking factor one) for G.

($\Rightarrow$ direction). Suppose that S is a valid single appearance schedule for G. From Theorem 2.5, we can assume without loss of generality that S has blocking factor one, and from Fact 2.4, there exists a valid single appearance schedule S' that has blocking factor one and contains no one-iteration loops. Then S' can be expressed as $S_a S_b$, where S_a and S_b are nonempty single appearance subschedules of S' that are not encompassed by a loop, since if S' is a schedule loop

$(n(\ldots)(\ldots)\ldots(\ldots))$, then $gcd(\{\mathbf{q}_G(A) \mid (A \in actors(G))\}) \geq n$ so S' does not have unity blocking factor — a contradiction. Since $S_a S_b$ is a minimal, valid single appearance schedule for G, every actor $A \in actors(S_a)$ is invoked $\mathbf{q}_G(A)$ times before any actor outside of $actors(S_a)$ is invoked. It follows that $actors(S_a)$ is subindependent of dent of $actors(S_b)$ in G. Also, by Fact 2.6, S_a is a valid single appearance schedule for $subgraph(actors(S_a))$ and S_b is a valid single appearance schedule for $subgraph(actors(S_b))$. *QED.*

Theorem 2.6 states that a strongly connected SDF graph G has a valid single appearance schedule only if we can find a subindependent partition Z_1, Z_2. If we can find such Z_1 and Z_2, then we can construct a valid single appearance schedule for G by constructing a valid single appearance schedule for all invocations associated with Z_1 and then concatenating a valid single appearance schedule for all invocations associated with Z_2. By repeatedly applying this type of decomposition, we can construct single appearance schedules whenever they exist, and we will elaborate on this extensively in the following chapter.

The following theorem presents a simple topological condition for the existence of a subindependent partition that leads to an efficient algorithm for finding a subindependent partition whenever one exists.

Theorem 2.7: Suppose that G is a nontrivial, strongly connected, consistent SDF graph. From G, remove all edges α for which $delay(\alpha) \geq total\_consumed(\alpha, G)$, and call the resulting SDF graph G'. Then

G has a subindependent partition if and only if G' is not strongly connected. Furthermore, if G' is not strongly connected, then any root strongly connected component Z of G' is subindependent of $(actors(G) - Z)$ in G.

Proof: First suppose that G' is not strongly connected, and let Z_1 be any root strongly connected component of G'. Thus, no edge in G that is directed from a member of $(actors(G) - Z_1)$ to a member of Z_1 is contained in G'. Thus, by the construction of G', for each edge α in G directed from a member of $(actors(G) - Z_1)$ to a member of Z_1, we have $delay(\alpha) \geq total\_consumed(\alpha, G)$. It follows that Z_1 is subindependent in G. Thus, since Z_1 is an arbitrary root strongly connected component of G', we have shown that if G' is not strongly connected, then G has a subindependent partition and any root strongly connected component of G' is subindependent in G.

To complete the proof, we show that whenever G has a subindependent partition, G' is not strongly connected. If G has a subindependent partition, then $actors(G)$ can be partitioned into Z_1 and Z_2 such that Z_1 is subindependent of Z_2 in G. By construction of G', there are no edges in G' directed from a member of Z_2 to a member of Z_1, so G' is not strongly connected. *QED*.

Theorem 2.7 establishes the validity of the following algorithm, which takes as input a nontrivial consistent, strongly connected SDF graph G, and finds a subindependent partition of G if one exists.

procedure SubindependentPartition(G)
Compute the repetitions vector $\mathbf{q}$ of G.
From G, remove each edge α for which
$delay(\alpha) \geq total\_consumed(\alpha, G)$.
Denote the resulting graph by G'.
Determine the strongly connected components of G'.
if G' consists of only one strongly connected component, $actors(G')$,
G' does not have a subindependent partition
else
for each strongly connected component Z
if no member of Z has an input edge α such that $source(\alpha) \notin Z$
Z is subindependent in G.

Let $m = max(\{|actors(G)|, |edges(G)|\})$. The algorithm presented in Subsection 2.1.4 computes the repetitions vector in time $O(m)$; it is obvious that the next step of algorithm *SubindependentPartition* — removing the edges with insufficient delay — can also be performed in $O(m)$ time; Tarjan's algorithm allows the determination of the strongly connected components in $O(m)$ time [Tarj72]; and the checks in the if-else segment are clearly $O(m)$ as well. Thus, the time complexity of algorithm *SubindependentPartition* is linear in the number of actors and edges in G.

The operation of algorithm *SubindependentPartition* is illustrated in Figure 2.16. For the strongly connected SDF graph on the left side of this figure, which we denote by G, $\mathbf{q}(A, B, C, D) = (1, 10, 2, 20)^T$. Thus, the delay on the edge directed from D to B (25) exceeds the total number of tokens consumed by B in a minimal schedule period of G (20). We remove this edge to obtain the new graph depicted on the right side of Figure 2.16. Since this new SDF graph is not

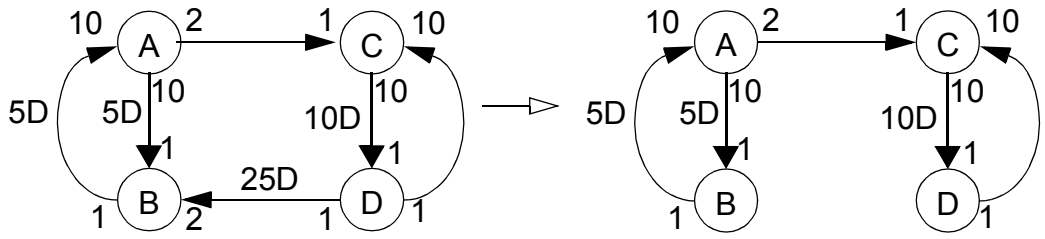


Figure 2.16. An illustration of algorithm *SubindependentPartition*.

strongly connected, a subindependent partition of G exists: the root strongly connected component $\{A, B\}$ is subindependent of the remaining actors $\{C, D\}$ in G.

S. S. Bhattacharyya. *Compiling Dataflow Programs for Digital Signal Processing*. PhD thesis, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, July 1994

3

SCHEDULING TO MINIMIZE CODE SIZE

In this chapter, we present systematic techniques for compiling SDF graphs into implementations that require minimum code size. We define a graph decomposition process that can be used to construct single appearance schedules whenever they exist. Based on this decomposition process, we define a general framework for developing scheduling algorithms, and we show that all scheduling algorithms that are constructed through this framework construct single appearance schedules whenever they exist. Also, we show that the code size optimality of the scheduling framework extends in a restricted way to SDF graphs that do not have single appearance schedules: the framework guarantees minimum code size for all actors that are not contained in subgraphs of a certain form, called *tightly interdependent subgraphs*.

In Section 3.2, we discuss considerations that must be addressed when incorporating clustering techniques into our scheduling framework, and we present a clustering technique that can be incorporated into the framework to increase the amount of buffering that occurs through registers. A large part of Section 3.2 is devoted to establishing that this clustering technique does not violate the code size minimization properties of the scheduling framework. In the following section, we

discuss the problem of constructing single appearance schedules that minimize the buffer memory requirement. Here, we focus on the class of chain-structured SDF graphs, and some extensions to more general graphs are given in Subsection 3.3.4. Finally, in Section 3.4 we describe in detail a number of research efforts that are closely related to the work presented in this section. These efforts include loop scheduling mechanisms in Gabriel, which were examined by How [How90]; a related loop scheduling technique described in [Buck93] for the Ptolemy system; the construction of uniprocessor schedules that minimize the number of context-switches, a problem that has been addressed in the COSSAP design environment [Ritz93]; and a number of techniques developed to compile procedural programs into efficient code for vector computers [Mura71, Alle87].

3.1 Loose Interdependence Algorithms

Definition 3.1: Suppose that G is a sample rate consistent, nontrivial strongly connected SDF graph. Then we say that G is **loosely interdependent** if G has a subindependent partition. We say that G is **tightly interdependent** if it is not loosely interdependent.

For example, consider the strongly connected SDF graph in figure 3.1. Here, the repetitions vector is $\mathbf{q}(A, B, C) = (3, 2, 1)^T$, and d_1, d_2 and d_3 represent the number of delays on the associated edges. From Definition 3.1, this SDF graph is loosely interdependent if and only if $(d_1 \geq 6)$ **or** $(d_2 \geq 2)$ **or** $(d_3 \geq 3)$; equivalently the graph is tightly interdependent if and only if $(d_1 < 6)$ **and** $(d_2 < 2)$ **and** $(d_3 < 3)$.

We will use the following fact, which follows immediately from the defini-

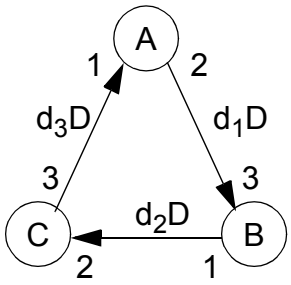


Figure 3.1. An example used to illustrate the concepts of loose and tight interdependence.

tion of loose interdependence.

Fact 3.1: If G_1 and G_2 are two isomorphic SDF graphs and G_1 is loosely interdependent, then G_2 is loosely interdependent.

Our code scheduling framework is based on the following definition, which decomposes the scheduling process into four distinct functions, and defines how algorithms for these functions can be combined to generate a class of scheduling algorithms.

Definition 3.2: Let ϑ_1 be any algorithm that takes as input a nontrivial strongly connected SDF graph G, determines whether G is loosely interdependent, and if so, finds a subindependent partition of G. Le ϑ_2 be any algorithm that finds the strongly connected components of a directed multigraph. Let ϑ_3 be any algorithm that takes an acyclic SDF graph and generates a valid single appearance schedule. Finally, let ϑ_4 be any algorithm that takes a tightly interdependent SDF graph and generates a valid looped schedule of blocking factor one. We define the algorithm

$L(\vartheta_1, \vartheta_2, \vartheta_3, \vartheta_4)$ by the sequence of steps shown in figure 3.2. This process for combining the algorithms ϑ_1, ϑ_2, ϑ_3, and ϑ_4 defines a family of algorithms $L(\bullet, \bullet, \bullet, \bullet)$, which we call **loose interdependence algorithms** because they exploit loose interdependence to decompose the input SDF graph. Given a loose interdependence algorithm $\zeta = L(\vartheta_1, \vartheta_2, \vartheta_3, \vartheta_4)$, we call the component algorithms ϑ_1, ϑ_2, ϑ_3, and ϑ_4 the **subindependence partitioning algorithm** of ζ, the **strongly connected components algorithm** of ζ, the **acyclic scheduling algorithm** of ζ, and the **tight scheduling algorithm** of ζ, respectively.

Since nested recursive calls decompose a graph into finer and finer strongly connected components, it easy to verify that a loose interdependence algorithm always terminates on a finite input graph. Also, since the *for*-loop in step 4 replaces each Ω_i in S' with a valid looped schedule for $subgraph(Z_i)$, we know from Fact 2.6 that these replacements yield a valid looped schedule for G, and thus that the output $S_L(G)$ of a loose interdependence algorithm is always a valid schedule.

We will also make use of the following observations in the remainder of this section.

Remark 3.1: Observe that step 4 does not insert or delete appearances of actors that are not contained in a nontrivial strongly connected component Z_k. Since ϑ_3 generates a single appearance schedule for G', we have that for ever actor A that is not contained in a nontrivial strongly connected component of G, $appearances(A, S_L(G)) = 1$.

procedure ScheduleLoops
input: a connected, consistent SDF graph G.
output: a valid unit blocking factor looped schedule $S_L(G)$ for G.

step 1: Use ϑ_2 to determine the nontrivial strongly connected components $Z_1, Z_2, \ldots, Z_s$.
step 2: Cluster $Z_1, Z_2, \ldots, Z_s$ into the actors $\Omega_1, \Omega_2, \ldots, \Omega_s$ respectively, and denote the resulting graph by G'. This is an acyclic graph.
step 3: Apply ϑ_3 to G', and denote the resulting schedule by S'.
step 4:

for $i = 1, 2, \ldots, s$

Let G_z denote $subgraph(Z_i)$.

Apply ϑ_1 to G_z.

if $X, Y \subseteq Z_i$ are found such that X is subindependent of Y in G_z.

- Let G_x denote $subgraph(X)$ and G_y denote $subgraph(Y)$.
- Determine the connected components $X_1, X_2, \ldots, X_v$ and $Y_1, Y_2, \ldots, Y_w$ of G_x and G_y, respectively.
- Recursively apply *ScheduleLoops* to construct

$$S_x = (q_{G_x}(X_1)S_L(subgraph(X_1)))\ldots(q_{G_x}(X_v)S_L(subgraph(X_v)))$$

and

$$S_y = (q_{G_y}(Y_1)S_L(subgraph(Y_1)))\ldots(q_{G_y}(Y_w)S_L(subgraph(Y_w))) \quad .$$

- Replace the single appearance of Ω_i in S' with $(q_{G_z}(X)S_x)(q_{G_z}(Y)S_y)$.

else ($subgraph(Z_i)$ is tightly interdependent)

- Apply ϑ_4 to obtain a valid schedule S_i for $subgraph(Z_i)$.
- Replace the single appearance of Ω_i in S' with S_i.

step 5: Output S' as $S_L(G)$.

Figure 3.2. The specification of how algorithms $\vartheta_1, \vartheta_2, \vartheta_3, \vartheta_4$ in Definition 3.2 are combined to form a *loose interdependence algorithm*.

Remark 3.2: If Z is a nontrivial strongly connected component of G and $A \in Z$, then since $S_L(G)$ is derived from $S'(G)$ by replacing the single appearance of each Ω_i, we have that

$$appearances(A, S_L(G)) = appearances(A, S_L(subgraph(Z))) \quad .$$

Remark 3.3: For each strongly connected component Z_k whose associated subgraph is loosely interdependent, L partitions Z_k into X and Y such that X is sub-independent of Y in $subgraph(Z_k)$, and replaces the single appearance of Ω_k in $S'(G)$ with $(q_{G_z}(X)S_x)(q_{G_z}(Y)S_y)$. If A is a member of the connected component X_i, then $A \notin Y$, so

$$appearances(A, (q_{G_z}(X)S_x)(q_{G_z}(Y)S_y))$$

$$= appearances(A, S_L(subgraph(X_i))) \quad .$$

Also, since A cannot be in any other strongly connected component besides Z_k, and since $S'(G)$ contains only one appearance of Ω_k, we have $appearances(A, S_L(G)) = appearances(A, (q_{G_z}(X)S_x)(q_{G_z}(Y)S_y))$. Thus, for $i = 1, 2, \ldots, v$,

$$(A \in X_i) \Rightarrow$$

$$appearances(A, S_L(G)) = appearances(A, S_L(subgraph(X_i))) \quad .$$

By a similar argument, we can show that for $i = 1, 2, \ldots, w$,

$$(A \in Y_i) \Rightarrow appearances(A, S_L(G)) = appearances(A, S_L(subgraph(Y_i)))$$

We will apply a loose interdependence algorithm to derive nonrecursive necessary and sufficient conditions for the existence of a valid single appearance schedule. First, we introduce two useful lemmas.

Lemma 3.1: Suppose G is a connected, consistent SDF graph; A is an actor in G that is not contained in any tightly interdependent subgraph of G; and ζ is a loose interdependence algorithm. Then A appears only once in $S_\zeta(G)$, the schedule generated by ζ.

Proof: From Remark 3.1, if A is not contained in a nontrivial strongly connected component of G, the result is obvious, so we assume, without loss of generality, that A is in some nontrivial strongly connected component Z_1 of G. From our assumptions, $subgraph(Z_1)$ must be loosely interdependent, so ζ partitions Z_1 into X and Y, where X is subindependent of Y in $subgraph(Z_1)$. Let Z_1' denote that connected component of $subgraph(X)$ or $subgraph(Y)$ that contains A. From Remark 3.3,

$$appearances(A, S_\zeta(G)) = appearances(A, S_\zeta(subgraph(Z_1'))) \quad .$$

From our assumptions, all nontrivial strongly connected subgraphs of $subgraph(Z_1')$ that contain A are loosely interdependent. Thus, if A is contained in a nontrivial strongly connected component Z_2 of $subgraph(Z_1')$, then ζ will partition Z_2, and we will obtain a proper subset Z_2' of Z_1' such that

$$appearances(A, S_\zeta(subgraph(Z_1')))$$
$$= appearances(A, S_\zeta(subgraph(Z_2'))) \quad .$$

Continuing in this manner, we get a sequence $Z_1', Z_2', \ldots$ of subsets of $actors(G)$ such that each Z_i' is a proper subset of Z_{i-1}', A is contained in each Z_i', and

$$appearances(A, S_\zeta(G)) = appearances(A, S_\zeta(subgraph(Z_1'))) = appearances(A, S_\zeta(subgraph(Z_2'))) = \ldots .$$

Since each Z_i' is a proper subset of its predecessor, we can continue this process only a finite number, say m, of times. Then $A \in Z_m'$, A is not contained in a non-trivial strongly connected component of $subgraph(Z_m')$, and

$$appearances(A, S_\zeta(G)) = appearances(A, S_\zeta(subgraph(Z_m'))) .$$

But from Remark 3.1, $S_\zeta(subgraph(Z_m'))$ contains only one appearance of A. *QED*.

Lemma 3.2: Suppose that G is a strongly connected, consistent SDF graph, $Y \subseteq actors(G)$ is subindependent in G, and Z is a strongly connected subset of $actors(G)$ such that $Y \cap Z \neq Z$ and $Y \cap Z \neq \varnothing$. Then $(Y \cap Z)$ is subindependent in $subgraph(Z)$.

Proof: Suppose that α is an edge directed from a member of $(Z - (Y \cap Z))$ to a member of $(Y \cap Z)$. By the subindependence of Y in G, $delay(\alpha) \geq consumed(\alpha)\mathbf{q}_G(sink(\alpha))$, and by Fact 2.7, $\mathbf{q}_G(sink(\alpha)) \geq \mathbf{q}_{subgraph(Z)}(sink(\alpha))$.

Thus, $delay(\alpha) \geq consumed(\alpha)\mathbf{q}_{subgraph(Z)}(sink(\alpha))$. Since this holds for any α directed from an actor in $(Z - (Y \cap Z))$ to an actor in $(Y \cap Z)$,

we conclude that $(Y \cap Z)$ is subindependent in $subgraph(Z)$. *QED*.

Corollary 3.1: Suppose that G is a strongly connected, consistent SDF graph, Z_1 and Z_2 are subsets of $actors(G)$ such that Z_1 is subindependent of Z_2 in G, and T is a tightly interdependent subgraph of G. Then $(actors(T) \subseteq Z_1)$ **or** $(actors(T) \subseteq Z_2)$.

Proof: (By contraposition). If $actors(T)$ has nonempty intersection with both Z_1 and Z_2, then from Lemma 3.2, $(actors(T) \cap Z_1)$ is subindependent in T, and thus, T is loosely interdependent. *QED*.

Theorem 3.1: 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.

Proof: ($\Leftarrow$ direction). Suppose that every nontrivial strongly connected subgraph of G is loosely interdependent, and let ζ be any loose interdependence algorithm. Since no actor in G is contained in a tightly interdependent subgraph, it follows from Lemma 3.1 that $S_\zeta(G)$ is a single appearance schedule for G.

($\Rightarrow$ direction). Suppose that G has a single appearance schedule and that Z is a strongly connected subset of $actors(G)$ such that $|Z| > 1$. Set $Z_0 = G$. From Theorem 2.6, there exist X_0, $Y_0 \subseteq Z_0$ such that X_0 is subindependent of Y_0 in $subgraph(Z_0)$, and $subgraph(X_0)$ and $subgraph(Y_0)$ both have single appearance schedules. If X_0 and Y_0 do not both intersect Z, then Z is completely

contained in some strongly connected component Z_1 of $subgraph(X_0)$ or $subgraph(Y_0)$. We can then apply Theorem 2.6 to partition Z_1 into X_1 and Y_1, and continue recursively in this manner until we obtain a strongly connected $Z_k \subseteq actors(G)$ with the following properties: there exist X_k, $Y_k \subseteq Z_k$ such that X_k is subindependent of Y_k in $subgraph(Z_k)$; $Z \subseteq Z_k$; and $(X_k \cap Z)$ and $(Y_k \cap Z)$ are both nonempty. From Lemma 3.2, $(X_k \cap Z)$ is subindependent in $subgraph(Z)$, so $subgraph(Z)$ must be loosely interdependent. *QED*.

Corollary 3.2: Given a connected, consistent SDF graph G, any loose interdependence algorithm will obtain a single appearance schedule if one exists.

Proof: If a single appearance schedule for G exists, then from Theorem 3.1, G contains no tightly interdependent subgraphs. In other words, no actor in G is contained in a tightly interdependent subgraph of G. From Lemma 3.1, the schedule resulting from any loose interdependence algorithm contains only one appearance of each actor in G. *QED*.

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. In general, however, there may be more than one way to decompose $actors(G)$ into two parts so that one of the parts is subindependent of the other in G. Thus, it is natural to ask the following question: Given two distinct partitions $\{Z_1, Z_2\}$ and $\{Z_1', Z_2'\}$ of $actors(G)$ such that Z_1 is subindependent of Z_2 in G, and Z_1' is subindependent of Z_2' in

G, is it possible that one of these partitions leads to a more compact schedule than the other? Fortunately, as we will show in the remainder of this section, the answer to this question is "No". In other words, any two loose interdependence algorithms that use the same tight scheduling algorithm always lead to equally compact schedules. The key reason is that tight interdependence is an additive property.

Lemma 3.3: Suppose that G is a connected, consistent SDF graph, Y and Z are distinct strongly connected subsets of $actors(G)$ such that $(Y \cap Z) \neq \varnothing$, and $subgraph(Y)$ and $subgraph(Z)$ are both tightly interdependent. Then $subgraph(Y \cup Z)$ is tightly interdependent.

Proof: (By contraposition). Let $H = (Y \cup Z)$, and suppose that $subgraph(H)$ is loosely interdependent. Then there exist H_1 and H_2 such that H_1 is subindependent of H_2 in $subgraph(H)$. From $H_1 \cup H_2 = H = Y \cup Z$, and $Y \cap Z \neq \varnothing$, it is easily seen that H_1 and H_2 both have a nonempty intersection with Y, or they both have a nonempty intersection with Z. Without loss of generality, assume that $H_1 \cap Y \neq \varnothing$ and $H_2 \cap Y \neq \varnothing$. From Lemma 3.2, $(H_1 \cap Y)$ is subindependent in $subgraph(Y)$, and thus $subgraph(Y)$ is not tightly interdependent. *QED*.

Lemma 3.3 implies that each SDF graph G has a unique set $\{T_1, T_2, \ldots, T_n\}$ of maximal tightly interdependent subgraphs such that $(i \neq j) \Rightarrow actors(T_i) \cap actors(T_j) = \varnothing$, and every tightly interdependent subgraph in G is contained in some T_i. We call each set $actors(T_i)$ a **tightly interdependent component** of G. It follows from Theorem 3.1 that G has a single

appearance schedule if and only if G has no tightly interdependent components. Furthermore, since the tightly interdependent components are unique, the performance of a loose interdependence algorithm, with regards to schedule compactness, is not dependent on the particular subindependence partitioning algorithm, the component algorithm used to partition the loosely interdependent subgraphs. The following theorem develops this result.

Theorem 3.2: Suppose that G is a connected, consistent SDF graph, A is an actor in G, and ζ is a loose interdependence algorithm.

(a). If A is not contained in a tightly interdependent component of G, then A appears only once in $S_\zeta(G)$; and

(b). If A is contained in a tightly interdependent component X, then

$$appearances(A, S_\zeta(G)) = appearances(A, S_\zeta(subgraph(X))) \quad —$$

the number of appearances of A is determined entirely by the tight scheduling algorithm of ζ.

Proof: If A is not contained in a tightly interdependent component of G, then A is not contained in any tightly interdependent subgraph. Then from Lemma 3.1, $appearances(A, S_\zeta(G)) = 1$. Thus the proof of part (a) is complete.

Now suppose that A is contained in some tightly interdependent component X of G. If $X = actors(G)$, we are done. Otherwise, set $M_0 = actors(G)$, and thus $X \neq M_0$; by definition, tightly interdependent graphs are strongly connected, so X is contained in some strongly connected component Z of $subgraph(M_0)$.

If X is a proper subset of Z, then $subgraph(Z)$ must be loosely interdependent, since otherwise $subgraph(X)$ would not be a maximal tightly interdependent subgraph. Thus ζ partitions Z into V and W such that V is subindependent of W in $subgraph(Z)$. We set M_1 to be that connected component of $subgraph(V)$ or $subgraph(W)$ that contains A. Since V and W partition Z, M_1 is a proper subset of M_0. Also from Remark 3.3,

$$\begin{aligned} &appearances(A, S_\zeta(subgraph(M_0))) \\ &= appearances(A, S_\zeta(subgraph(M_1))) \quad , \end{aligned} \tag{3-1}$$

and from Corollary 3.1, $X \subseteq M_1$.

On the other hand, if $X = Z$, then we set $M_1 = X$. Since $X \neq M_0$, M_1 is a proper subset of M_0; from Remark 3.2, (3-1) holds, and trivially, $X \subseteq M_1$.

If $X \neq M_1$, then we can repeat the above procedure to obtain a proper subset M_2 of M_1 such that

$$\begin{aligned} &appearances(A, S_\zeta(subgraph(M_1))) \\ &= appearances(A, S_\zeta(subgraph(M_2))) \quad , \end{aligned}$$

and $X \subseteq M_2$. Continuing this process, we get a sequence $M_0, M_1, M_2, \ldots$. Since for each $i > 1$, M_i is a proper subset of its predecessor M_{i-1}, we cannot repeat this process indefinitely — eventually, for some $k \geq 1$, we will have $X = M_k$. But, by construction

$$appearances(A, S_\zeta(G)) = appearances(A, S_\zeta(subgraph(M_0)))$$

$= appearances(A, S_\zeta(subgraph(M_1)))$

$= \ldots = appearances(A, S_\zeta(subgraph(M_k)))$;

and thus, $appearances(A, S_\zeta(G)) = appearances(A, S_\zeta(subgraph(X)))$. *QED*.

Theorem 3.2 states that the tight scheduling algorithm is independent of the subindependence partitioning algorithm and vice-versa. Any subindependence partitioning algorithm guarantees that there is only one appearance for each actor outside the tightly interdependent components, and the tight scheduling algorithm completely determines the number of appearances for actors inside the tightly interdependent components. For example, if we develop a new subindependence partitioning algorithm that is more efficient in some way (for example, it is faster or minimizes the memory required to implement buffering), we can replace it for any existing subindependence partitioning algorithm without changing the compactness of the resulting schedules — we don't need to analyze its interaction with the rest of the loose interdependence algorithm. Similarly, if we develop a new tight scheduling algorithm that schedules any tightly interdependent graph more compactly than the existing tight scheduling algorithm, we are guaranteed that using the new algorithm instead of the old one will lead to more compact schedules overall.

The complexity of a loose interdependence algorithm ζ depends on its subindependence partitioning algorithm ζ_{sp}, strongly connected components algorithm ζ_{sc}, acyclic scheduling algorithm ζ_{as}, and tight scheduling algorithm ζ_{ts}. From Definition 3.2, we see that ζ_{ts} is applied exactly once for each tightly interdependent component. For example, the algorithm specified in Subsection 2.1.5,

ConstructValidSchedule, can be used as the tight scheduling algorithm. If this algorithm is applied to a tightly interdependent component X, it runs in time that is linear in the total number of invocations in a minimal schedule period of $subgraph(X)$. That is, the running time is $O(I_X)$, where $I_X = \sum_{A \in X} \mathbf{q}_{subgraph(X)}(A)$. Thus, if ζ_{ts} is algorithm *ConstructValidSchedule* and ζ is applied to an SDF graph G, the total time that ζ_{ts} accounts for is $O(I_G)$, where $I_G = \sum_{A \in actors(G)} \mathbf{q}_G(A)$.

The other component algorithms, ζ_{sc}, ζ_{as}, and ζ_{sp}, are successively applied to decompose an SDF graph, and the process is repeated until all tightly interdependent components are found. In the worst case, each decomposition step isolates a single actor from the current n-actor subgraph, and the decomposition must be recursively applied to the remaining $(n-1)$-actor subgraph. Thus, if G denotes the input SDF graph, then ζ performs $|actors(G)|$ decomposition steps in the worst case. Tarjan's algorithm [Tarj72] allows the strongly connected components of G to be found in $O(m)$ time, where $m = max(\{|actors(G)|, |edges(G)|\})$. Hence ζ_{sc} can be chosen to be linear, and since at most $|actors(G)| \leq m$ decomposition steps are required, the total time that such a ζ_{sc} accounts for in ζ is $O(m^2)$. Finally, in Section 2.6 we described a simple linear-time algorithm that constructs a single appearance schedule for an acyclic graph. Thus ζ_{as} can also be chosen such that its total time is also $O(m^2)$.

We have specified ζ_{sp}, ζ_{sc}, ζ_{as}, and ζ_{ts} such that the resulting loose interdependence algorithm ζ has worst-case running time that is $O(m^2 + I)$, where $m = max(\{|actors(G)|, |edges(G)|\})$ and $I = \sum_{A \in actors(G)} \mathbf{q}_G(A)$. Note that our worst-case estimate is conservative — in practice, usually only a few decomposition steps are required to fully schedule a strongly connected subgraph, while our estimate assumes $|actors(G)|$ steps. Furthermore, a more accurate expression for the total time that the tight scheduling algorithm accounts for is $O\left(\sum_{i=1}^{p} \sum_{A \in actors(T_i)} \mathbf{q}_{T_i}(A)\right)$, where $T_1, T_2, \ldots, T_p$ are the subgraphs associated with the tightly interdependent components of G. When the tightly interdependent components form only a small part of G this bound will be much tighter than the $\sum_{A \in actors(G)} \mathbf{q}_G(A)$ bound.

3.2 Clustering in a Loose Interdependence Algorithm

As we discussed in 2.3, clustering subgraphs — grouping subgraphs so that they are invoked as single units — can be used to guide a scheduler toward more efficient schedules. However, certain clustering decisions conflict with code-space minimization goals, and thus if any clustering is to be incorporated into a loose interdependence algorithm, then the possible degradation on code-compaction potential should be considered. In this section, we present a useful clustering technique for increasing the frequency of data transfers that occur through machine registers rather than memory, and we prove that this technique does not interfere with the code compactness potential of a loose interdependence algorithm — this

clustering preserves the properties of loose interdependence algorithms discussed in the previous section.

Figure 3.3 illustrates two ways in which arbitrary clustering decisions can

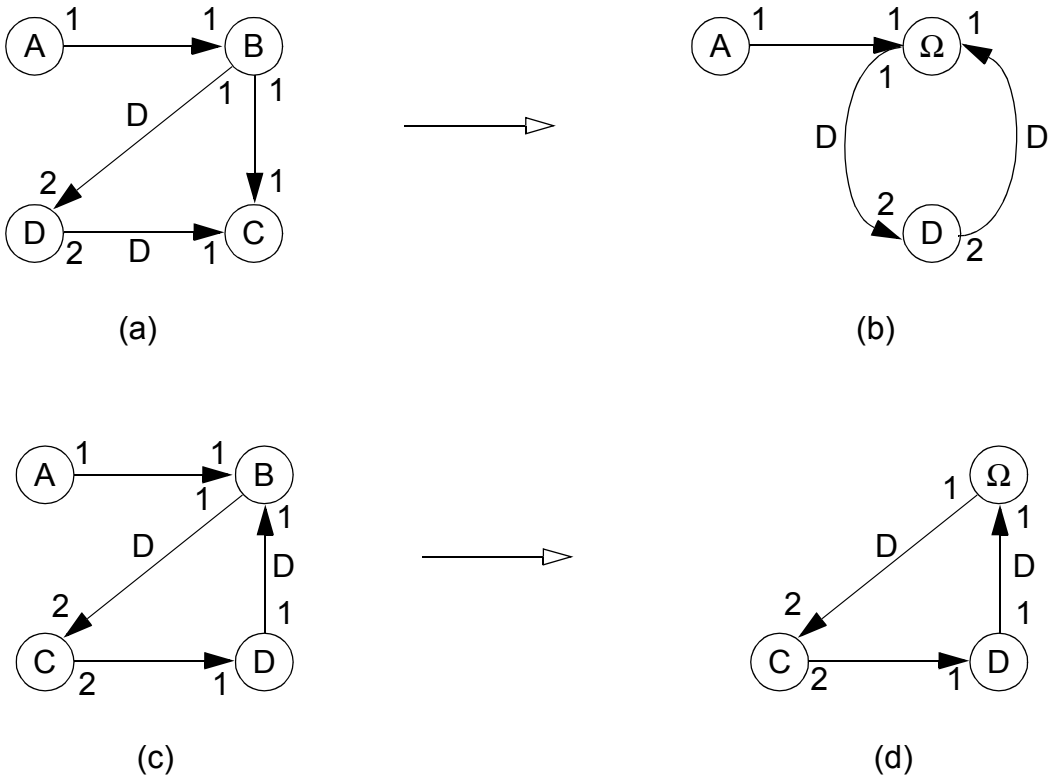


Figure 3.3. Examples of clustering decisions that conflict with code compactness goals.

conflict with code compactness objectives. Observe that the SDF graph in figure 3.3(a) is acyclic, so it must have a single appearance schedule. Figure 3.3(b) shows the hierarchical SDF graph that results from clustering actors B and C in figure 3.3(a) into the single actor Ω. It is easily verified that in figure 3.3(b), $subgraph(\{\Omega, D\})$ is tightly interdependent. Thus, the clustering of B and C in

figure 3.3(a) cancels the existence of a single appearance schedule.

In figure 3.3(c), $subgraph(\{B, C, D\})$ is a tightly interdependent component and actor A is not contained in any tightly interdependent subgraph. From Theorem 3.2, we know that any loose interdependence algorithm will schedule the graph of figure 3.3(c) in such a way that A appears only once. Now observe that the hierarchical SDF graph that results from clustering A and B, shown in figure 3.3(d), is a tightly interdependent graph. It can be verified that the most compact minimal periodic schedule for this graph is $\Omega C(2D)\Omega$, which leads to the schedule $ABC(2D)AB$ for figure 3.3(c). By increasing the extent of the tightly interdependent component $subgraph(\{B, C, D\})$ to subsume actor A, this clustering decision increases the minimum number of appearances of A in the final schedule.

Thus, we see that a clustering decision can conflict with optimal code compactness if it introduces a new tightly interdependent component or extends an existing tightly interdependent component. In this section, we present a clustering technique of practical use and prove that it neither extends nor introduces tight interdependence. Our clustering technique and its compatibility with loose interdependence algorithms is summarized by Fact 3.2 below. This fact is an immediate corollary of Theorem 3.3, which will be presented later in this section. Establishing Theorem 3.3 is the main topic of the remainder of this section.

Fact 3.2: Clustering two adjacent actors A and B in an SDF graph does not introduce or extend a tightly interdependent component if (a) Neither A nor B is contained in a tightly interdependent component; (b) At least one edge directed from A to B has zero delay; (c) A and B are invoked the same number of times in a periodic schedule; *and* (d) B has no predecessors other than A or B.

We motivate our clustering technique with the example shown in figure 3.4. The repetitions vector for the SDF graph in figure 3.4(a) is

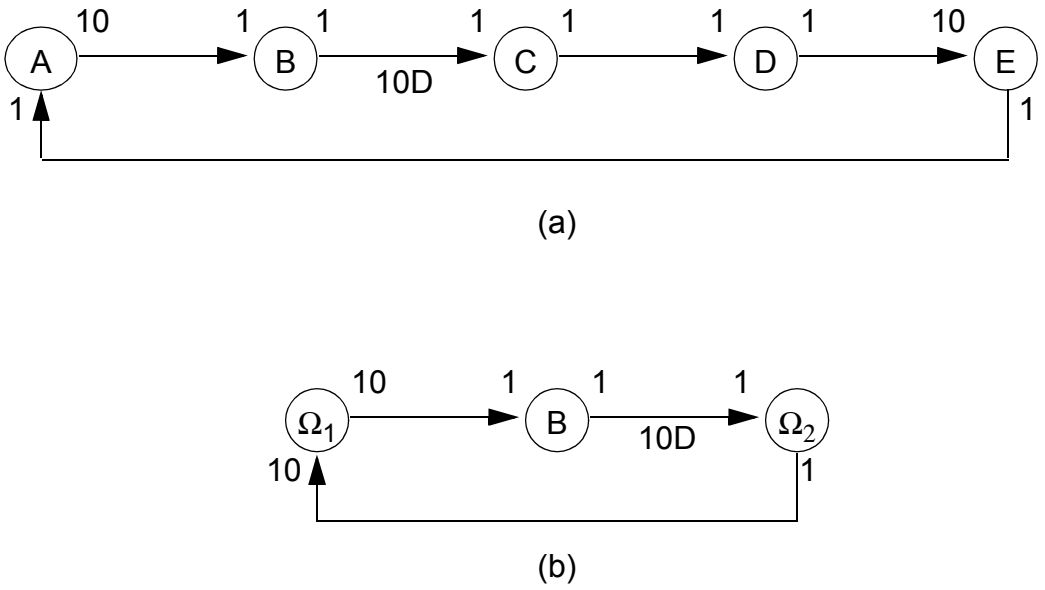


Figure 3.4. An example of clustering to increase the frequency of data transfers that occur through registers rather than memory.

$\mathbf{q}(A, B, C, D, E) = (1, 10, 10, 10, 1)^T$, and one valid single appearance schedule for this graph is $(10C)(10D)EA(10B)$. This schedule is inefficient with regards to buffering. Due to the schedule loop that specifies ten successive invocations of actor C, the data transfers between C and D cannot take place in machine registers and 10 units of memory are required to implement the edge $C \rightarrow D$. However, observe that the four conditions of Fact 3.2 all hold for the adjacent pairs $\{C, D\}$ and $\{A, E\}$. Thus, we can cluster these pairs without cancelling the existence of a single appearance schedule. The hierarchical SDF graph that results

from this clustering is shown in figure 3.4(b); this graph leads to the valid single appearance schedule

$$(10\Omega_2)\Omega_1(10B) \Rightarrow (10CD)EA(10B) \quad .$$

In this second schedule, each token produced by C is consumed by D in the same loop iteration, so all of the transfers between C and D can occur through a single machine register. Thus, the clustering of C and D saves 10 units of memory for the data transfers between C and D, and it allows these transfers to be performed through a register rather than memory, which will usually result in faster code.

When it is not ambiguous, we will the following additional notation in the development of this section.

Definition 3.3: Let G be an SDF graph and suppose that we cluster a subset W of actors in G. We will refer to the resulting hierarchical SDF graph as G', and we will refer to the actor in G' into which W has been clustered as Ω. For each edge α in G that is not contained in $subgraph(W, G)$, we denote the corresponding edge in G' by α'. Finally, if $X \subseteq actors(G)$, we denote the corresponding subset of $actors(G')$ as X'. That is, X' contains all members of X that are not in W, and if X contains one or more members of W, then X' also contains Ω.

For example if G is the SDF graph in figure 3.3(a), $W = \{B, C\}$, and α and β respectively denote $B \rightarrow D$ and $A \rightarrow B$, then we denote the graph in figure 3.3(b) by G', and in G', we denote $\Omega \rightarrow D$ by α' and $A \rightarrow \Omega$ by β'. Also, if $X = \{A, B\}$ then $X' = \{A, \Omega\}$.

Lemma 3.4: Suppose that G is a strongly connected, consistent SDF graph, and X_1 and X_2 form a partition of $actors(G)$ such that X_1 is subindependent of X_2 in G. Also, suppose that A and B are actors in G such that $A, B \in X_1$ or $A, B \in X_2$. If we cluster $W = \{A, B\}$, then the resulting SDF graph G' is loosely interdependent.

Proof: Let Φ denote the set of edges in G that are directed from an actor in X_2 to an actor in X_1, and let Φ' denote the set of edges in G' that are directed from an actor in X_2' to an actor in X_1'. Since $subgraph(\{A, B\})$ does not contain any edges in Φ, it follows that $\Phi' = \{\alpha' | \alpha \in \Phi\}$. From Fact 2.9, we have that for all α', $\mathbf{q}_{G'}(sink(\alpha'))consumed(\alpha') = \mathbf{q}_G(sink(\alpha))consumed(\alpha)$. Now since X_1 is subindependent of X_2 in G, for all $\alpha \in \Phi$, $delay(\alpha) \geq \mathbf{q}_G(sink(\alpha))consumed(\alpha)$. It follows that for all $\alpha' \in \Phi'$, $delay(\alpha') \geq \mathbf{q}_{G'}(sink(\alpha'))consumed(\alpha')$, and we conclude that X_1' is sub-independent of X_2' in G'. But, by construction, X_1' and X_2' partition $actors(G')$; thus, G' is loosely interdependent. *QED*.

Lemma 3.5: Suppose that G is a connected, consistent SDF graph, Z is a proper subset of $actors(G)$, $A_1 \in Z$, and A_2 is an actor that is contained in $actors(G)$ but not in Z such that

(1). A_2 is not adjacent to any member of $(Z - \{A_1\})$, and

(2). for some positive integer k, $\mathbf{q}_G(A_2) = k\mathbf{q}_G(A_1)$.

If we cluster $W = \{A_1, A_2\}$ in G, then $subgraph(Z - \{A_1\} + \{\Omega\}, G')$ is isomorphic to $subgraph(Z, G)$.

As a simple illustration, consider again the clustering example of figure 3.3(c) and figure 3.3(d). Let G and G' respectively denote the graphs of figure 3.3(c) and figure 3.3(d), and let $Z = \{B, C\}$, $A_1 = B$, and $A_2 = A$. Then $(Z - \{A_1\} + \{\Omega\}) = \{C, \Omega\}$, and clearly, $subgraph(\{C, \Omega\}, G')$ is isomorphic to $subgraph(\{B, C\}, G)$.

Proof of Lemma 3.5: Let $X = subgraph(Z - \{A_1\} + \{\Omega\}, G')$, let Φ denote the set of edges in $subgraph(Z, G)$, and let Φ' denote the set of edges in X. From (1), every edge in X has a corresponding edge in $subgraph(Z, G)$, and vice-versa, and thus $\Phi' = \{\alpha' | \alpha \in \Phi\}$. Now, from the definition of clustering a subgraph, we know that $produced(\alpha') = produced(\alpha)$ for any edge $\alpha \in \Phi$ such that $source(\alpha) \neq A_1$. If $source(\alpha) = A_1$ then α is replaced by α' with $source(\alpha') = \Omega$, and

$$produced(\alpha') = produced(\alpha)\mathbf{q}_G(A_1) / gcd(\{\mathbf{q}_G(A_1), \mathbf{q}_G(A_2)\}) \quad .$$

But, $gcd(\{\mathbf{q}_G(A_1), \mathbf{q}_G(A_2)\}) = gcd(\{\mathbf{q}_G(A_1), k\mathbf{q}_G(A_1)\}) = \mathbf{q}_G(A_1)$,

so $produced(\alpha') = produced(\alpha)$. Thus $produced(\alpha') = produced(\alpha)$ for all $\alpha \in \Phi$. Similarly, we can show that $consumed(\alpha') = consumed(\alpha)$ for all $\alpha \in \Phi$. Thus, the mappings $f_1 : Z \rightarrow actors(X)$ and $f_2 : \Phi \rightarrow \Phi'$ defined by

$$f_1(A) = A \text{ if } A \neq A_1,\ f_1(A_1) = \Omega \text{ ; and } \forall \alpha, f_2(\alpha) = \alpha'$$

demonstrate that $subgraph(Z, G)$ is isomorphic to X. *QED*.

Lemma 3.6: Suppose that G is a consistent, strongly connected SDF graph and Z is a strongly connected subset of actors in G such that $q_G(Z) = 1$. Suppose Z_1 and Z_2 form a partition of Z such that Z_1 is subindependent of Z_2 in $subgraph(Z, G)$. Then Z_1 is subindependent of Z_2 in G.

Proof: For each edge α directed from a member of Z_2 to a member of Z_1, we have $delay(\alpha) \geq \mathbf{q}_{subgraph(Z)}(sink(\alpha))consumed(\alpha)$. From Fact 2.7, $\mathbf{q}_{subgraph(Z)}(A) = \mathbf{q}_G(A)$ for all $A \in Z$. Thus, for all edges α in $subgraph(Z)$,

$$\mathbf{q}_{subgraph(Z)}(sink(\alpha))consumed(\alpha) = \mathbf{q}_G(sink(\alpha))consumed(\alpha),$$

and we conclude that Z_1 is subindependent of Z_2 in G. *QED*.

Lemma 3.7: Suppose that G is a consistent, strongly connected SDF graph, A and B are distinct actors in G, and $W = \{A, B\}$ forms a proper subset of $actors(G)$. Suppose also that the following four conditions all hold:

(1). Neither A nor B is contained in a tightly interdependent subgraph of G.

(2). There is at least one edge directed from A to B that has zero delay.

(3). B has no predecessors other than A or B.

(4). $\mathbf{q}_G(B) = k\mathbf{q}_G(C)$ for $k \in \{1, 2, 3, \ldots\}$, and for some $C \in actors(G)$ such that $C \neq B$.

Then the SDF graph G' that results from clustering W in G is loosely interdependent.

Proof: From (1), G must be loosely interdependent, so there exist subsets Z_1 and Z_2 of $actors(G)$ such that Z_1 and Z_2 partition $actors(G)$, and Z_1 is subindependent of Z_2 in G. If $A, B \in Z_1$ or $A, B \in Z_2$, then from Lemma 3.4, we are done. Now, condition (2) precludes the scenario $((B \in Z_1)$ **and** $(A \in Z_2))$, so the only remaining possibility is $((A \in Z_1)$ **and** $(B \in Z_2))$. There are two subcases to consider here:

(i). B is not the only member of Z_2. Then from (3), $(Z_1 + \{B\})$ is subindependent of $(Z_2 - \{B\})$. But $A, B \in Z_1 + \{B\}$, so Lemma 3.4 again guarantees that G' is loosely interdependent.

(ii). $Z_2 = \{B\}$. Thus, we have Z_1 is subindependent of $\{B\}$, so

$$\forall(\alpha \in \{\alpha \in edges(G) \,|\, sink(\alpha) \neq B\}) \quad ,$$

$$(source(\alpha) = B) \Rightarrow delay(\alpha) \geq total\_consumed(\alpha, G) \quad . \qquad (3\text{-}2)$$

Also, since $C \in Z_1$, we have from (4) that

$$q_G(Z_1) = gcd(\{\mathbf{q}_G(N) \,|\, (N \in Z_1)\})$$

$$= gcd(\{\mathbf{q}_G(N) \,|\, (N \in Z_1)\} \cup \{k\mathbf{q}_G(C)\})$$

$$= gcd(\{\mathbf{q}_G(N) \,|\, (N \in Z_1)\} \cup \{\mathbf{q}_G(B)\})$$

$$= gcd(\{\mathbf{q}_G(N) \mid N \in actors(G)\}) = 1 \quad .$$

That is,

$$q_G(Z_1) = 1 \ . \tag{3-3}$$

Now if Z_1 is not strongly connected, then it has a proper subset Y such that there are no edges directed from a member of $(Z_1 - Y)$ to a member of Y. Furthermore, from condition (3), $A \notin Y$. This is true because if Y contained A, then there would be no path directed from a member of $(Z_1 - Y)$ to B, and thus G would not be strongly connected. Thus, $A \in (Z_1 - Y)$, and there are no edges directed from a member of $(Z_1 - Y)$ to a member of Y. So all edges directed from a member of $(Z_1 - Y + \{B\})$ to Y have actor B as their source. From (3-2), it follows that Y is subindependent of $(Z_1 - Y + \{B\})$ in G. Now, $A, B \in (Z_1 - Y + \{B\})$, so applying Lemma 3.4, we conclude that G' is loosely interdependent.

If Z_1 is strongly connected, we know from condition (1) that there exists a partition X_1, X_2 of Z_1 such that X_1 is subindependent of X_2 in $subgraph(Z_1)$. From (3-3) and Lemma 3.6, X_1 is subindependent of X_2 in G. Now if $A \in X_1$, then from condition (3), $\{B\}$ is subindependent of X_2 in G, so from Fact 2.11(a), $(X_1 \cup \{B\})$ and X_2 constitute a subindependent partition of G. Applying Lemma 3.4, we see that G' is loosely interdependent. On the other hand, suppose that $A \in X_2$. Then from (3-2), we know that X_1 is subindependent of $\{B\}$ in G.

From Fact 2.11(b), it follows that X_1 and $(X_2 \cup \{B\})$ constitute a subindependent partition of G, so again we can apply Lemma 3.4 to conclude that G' is loosely interdependent. *QED*.

Theorem 3.3: Suppose that G is a consistent, connected SDF graph, A and B are distinct actors in G such that B is a successor of A, and $W = \{A, B\}$ is a proper subset of $actors(G)$. Suppose also that the following four conditions all hold:

(1). Neither A nor B is contained in a tightly interdependent component of G.

(2). At least one edge directed from A to B has zero delay.

(3). For some positive integer k, $\mathbf{q}_G(B) = k\mathbf{q}_G(A)$.

(4). Actor B has no predecessors other than A or B.

Then the tightly interdependent components of G' are the same as the tightly interdependent components of G.

Proof: Observe that all subgraphs in G that do not contain A nor B are not affected by the clustering of W, and thus it suffices to show that all strongly connected subgraphs in G' that contain Ω are loosely interdependent. So we suppose that Z' is a strongly connected subset of actors in G' that contains Ω, and we let Z denote the corresponding subset of actors in G; that is $Z = Z' - \{\Omega\} + \{A, B\}$. Now, in $subgraph(Z', G')$, suppose that there is a cycle consisting of Ω and two other actors, C and D. From condition (4), this implies that there is a cycle in G containing A, C, D, and possibly B. The two possible ways in which a cycle in G introduces a cycle consisting of Ω in G' are

illustrated in figure 3.5(a) and (b); the situation in figure 3.5(c) cannot arise because of condition (4).

Now in $subgraph(Z', G')$, if one or more of the cycles that pass through Ω correspond to figure 3.5(a), then Z must be a strongly connected subset in G. Otherwise, all of the cycles involving Ω correspond to figure 3.5(b), so $(Z - \{B\})$ is strongly connected, and from condition (4), no member of

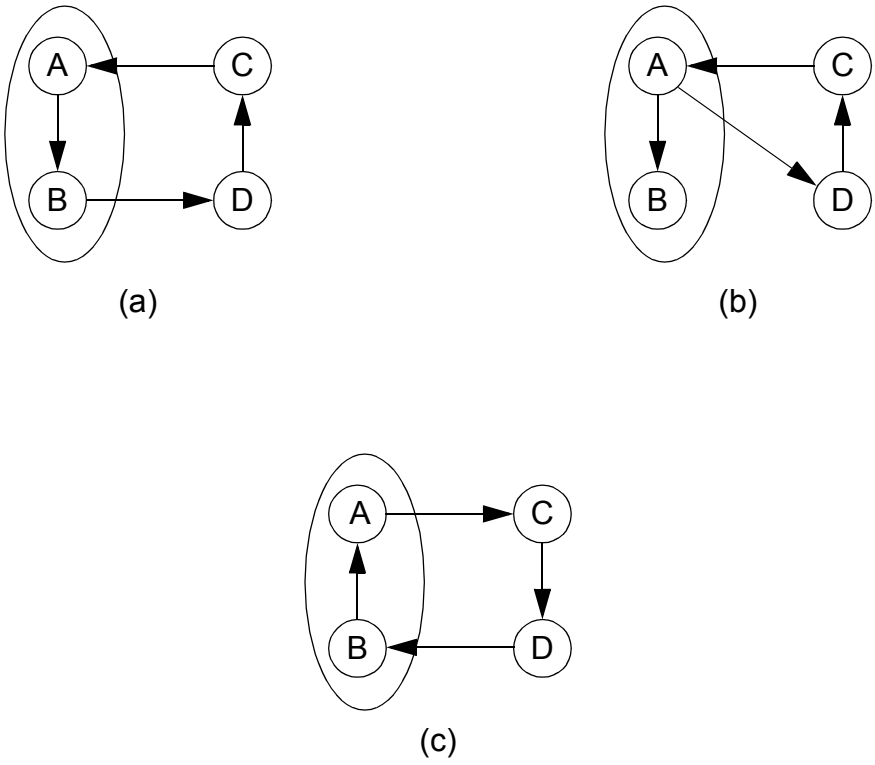


Figure 3.5. An illustration of how a cycle containing Ω originates in G' for Theorem 3.3. The two possible scenarios are shown in (a) and (b); (c) will not occur due to condition (4). SDF parameters on the edges have not been assigned because they are not relevant to the introduction of cycles.

$(Z - \{A, B\})$ is adjacent to B. In the former case, Lemma 3.7 immediately yields the loose interdependence of $subgraph(Z', G')$.

In the latter case, Lemma 3.5 guarantees that $subgraph(Z - \{B\}, G)$ is isomorphic to $subgraph(Z', G')$. Since $A \in (Z - \{B\})$, and since from condition (1), A is not contained in any tightly interdependent subgraph of G, it follows that $subgraph(Z', G')$ is loosely interdependent. *QED*.

If we assume that the input SDF graph has a single appearance schedule, then we can ignore condition (1). From our observations, this is a valid assumption for a large class of practical SDF graphs. Also, condition (3) can be verified by examining any single edge directed from A to B; if α is an edge directed from A to B, then condition (3) is equivalent to $produced(\alpha) = k\,consumed(\alpha)$. In our current implementation, we consider only the case $k = 1$ for condition (3) because in practice, this corresponds to most of the opportunities for efficiently using registers to implement the buffers for the edges in an SDF graph.

The following corollary assures us that when applying Theorem 3.3, no further checks are necessary to determine whether the clustering of A and B introduces deadlock.

Corollary 3.3: Assume the hypotheses of Theorem 3.3, including conditions (1) through (4). Then G' is not deadlocked.

Proof: (By contraposition). If G' is deadlocked, then there exists a fundamental cycle in G' whose associated graph G_f is deadlocked. By the definition of tight interdependence, G_f is tightly interdependent, so $actors(G_f)$ is contained in some tightly interdependent component X of G'. Thus, Theorem 3.3 guarantees

that $subgraph(X, G')$ is a tightly interdependent subgraph of G, and hence that the deadlocked graph G_f is contained in G. It follows that G is deadlocked, and G is not a consistent SDF graph. *QED*.

Under the assumption that the input SDF graph has a single appearance schedule, the clustering process defined by Theorem 3.3 requires only *local* data-flow information, and thus it can be implemented very efficiently. If our assumption that a single appearance schedule exists is wrong, then we can always undo our clustering decisions. Since the assumption is frequently valid, and since it leads to an efficient algorithm, this is the form in which we have implemented Theorem 3.3. Finally, in addition to making buffering more efficient, our clustering process provides a fast way to reduce the size of an SDF graph without cancelling the existence of a single appearance schedule. When used as a preprocessing technique, this can sharply reduce the execution time of a loose interdependence algorithm.

3.3 Minimizing Buffer Memory: Chain-Structured Graphs

In this section, we address the problem of constructing single appearance schedules that minimize the buffer memory requirement. The work presented in this section was done jointly with Praveen K. Murthy, a fellow graduate student at U. C. Berkeley [Murt94a].

Our model of buffering here is that discussed in Section 2.2 — each buffer is mapped to a contiguous and independent block of memory. Scheduling to minimize the amount of memory required for buffering while taking advantage of more flexible buffer implementations, a more difficult problem, is mainly beyond the scope of this thesis; one simple technique is given in Subsection 3.3.4, and some of

the pertinent issues are elaborated on in Section 4. Also, in this section, we focus on SDF graphs that are chain-structured; some extensions to more general graphs are discussed in Subsection 3.3.4.

In [Ade94], Ade develops upper bounds on the minimum buffer memory requirement for a number of restricted classes of SDF graphs. The graphs considered each consist of a chain-structured subgraph, together with zero or more edges directed between distinct actors in the chain-structured subgraph. For graphs that fall into the categories considered, Ade presents an efficiently computable upper bound on the minimum buffer memory required over all valid schedules, and Ade presents simulation data that demonstrates that on average, the computed bounds are close to the corresponding actual minima. Since Ade's bounds attempt to minimize over all valid schedules, and since single appearance schedules generally have much larger buffer memory requirements than schedules that are optimized for minimum buffer memory only, Ade's bounds cannot consistently give close estimates of the minimum buffer memory requirement for single appearance schedules.

In Section 2.6, we demonstrated that every consistent, acyclic SDF graph has a valid single appearance schedule since given a topological sort $A_1, A_2, \ldots A_n$ for a connected, consistent, acyclic SDF graph G, $(\mathbf{q}_G(A_1)A_1)(\mathbf{q}_G(A_2)A_2)\ldots(\mathbf{q}_G(A_n)A_n)$ is always a valid schedule. However single appearance schedules constructed from topological sorts in this way can be inefficient with regards buffer memory. For example, consider the SDF graph in figure 3.6. Here, $\mathbf{q}(A, B, C, D) = (9, 12, 12, 8)^T$, and there is only one topological sort — A, B, C, D. Thus, the approach outlined in Section 2.6 yields the valid single appearance schedule $S_1 \equiv (9A)(12B)(12C)(8D)$, and one can easily

verify that $buffer\_memory(S_1) = 36 + 12 + 24 = 72$. In contrast, $S_2 \equiv (3(3A)(4B))(4(3C)(2D))$ is an alternative single appearance schedule (with the same blocking factor as S_1 — unity) with a much lower buffer memory requirement: $buffer\_memory(S_2) = 12 + 12 + 6 = 30$.

As we will show in Subsection 3.3.1, 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. In this section, we show that the problem of finding the valid single appearance schedule that minimizes buffering memory for a chain-structured SDF graph is similar to the problem of most-efficiently multiplying a chain of matrices, for which a cubic-time dynamic programming algorithm exists [Godb73]. We show that this dynamic programming technique can be adapted to our problem to give an algorithm with time complexity $O(m^3)$, where m is the number of actors in the input chain-structured SDF graph. Finally, in Subsection 3.3.4, we discuss how the dynamic programming technique of Subsection 3.3.2 can be applied to other problems in the construction of efficient looped schedules.

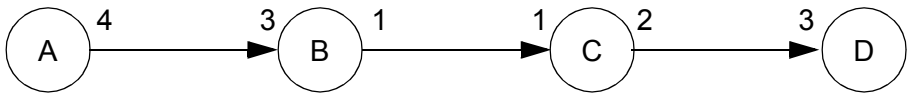


Figure 3.6. A chain-structured SDF graph.

For simplicity, in this section we assume that the edges in a chain-structured SDF graph have no delay; however, the techniques presented here can be extended to handle delays.

3.3.1 A Class of Recursively Constructed Schedules

Let G be a chain-structured SDF graph with actors $A_1, A_2, \ldots, A_m$ and edges $\alpha_1, \alpha_2, \ldots, \alpha_{m-1}$ such that each α_k is directed from A_k to A_{k+1}. In the trivial case, $m = 1$, we immediately obtain A_1 as a valid single appearance schedule for G. Otherwise, given any $i \in \{1, 2, \ldots, m-1\}$, define

$$left(i) \equiv subgraph(\{A_1, A_2, \ldots, A_i\}, G) \quad \text{, and}$$

$$right(i) \equiv subgraph(\{A_{i+1}, A_{i+2}, \ldots, A_m\}, G) \quad .$$

From Fact 2.7, if S_L and S_R are valid minimal single appearance schedules for $left(i)$ and $right(i)$, respectively, then $(q_L S_L)(q_R S_R)$ is a valid minimal single appearance schedule for G, where $q_L = gcd(\{\mathbf{q}_G(A_j) \,|\, 1 \le j \le i\})$ and $q_R = gcd(\{\mathbf{q}_G(A_j) \,|\, i < j \le m\})$.

For example, suppose that G is the SDF graph in figure 3.6 and suppose $i = 2$. It is easily verified that $\mathbf{q}_{left(i)}(A, B) = (3, 4)^T$ and $\mathbf{q}_{right(i)}(C, D) = (3, 2)^T$. Thus, $S_L = (3A)(4B)$ and $S_R = (3C)(2D)$ are valid minimal single appearance schedules for $left(i)$ and $right(i)$, and $(3(3A)(4B))(4(3C)(2D))$ is a valid minimal single appearance schedule

for figure 3.6.

We can recursively apply this procedure of decomposing a chain-structured SDF graph into left and right subgraphs to construct a schedule. However, different sequences of choices for i will in general lead to different schedules. For a given chain-structured SDF graph, we refer to the set of valid minimal single appearance schedules obtainable from this recursive scheduling process as the set of **R-schedules**.

We will use the following fact, which is easily verified from the definition of an R-schedule.

Fact 3.3: Suppose that G is a nontrivial chain-structured SDF graph, and $(delay(\alpha) = 0), \forall(\alpha \in edges(G))$. Then a valid single appearance schedule S for G is an R-schedule if and only if every schedule loop L contained in the schedule $(1S)$ satisfies the following property:

(a). L has a single iterand, which is an actor; that is, $L = (nA)$ for some positive integer n and some $A \in actors(G)$; *or*

(b). L has exactly two iterands, which are schedule loops having coprime iteration counts; that is, $L = (m(n_1S_1)(n_2S_2))$, where m, n_1 and n_2 are positive integers; $gcd(n_1, n_2) = 1$; and S_1 and S_2 are looped schedules.

If a schedule loop L satisfies condition (a) or condition (b) of Fact 3.3, we say that L is an **R-loop**. Thus, a valid single appearance schedule S is an R-schedule if and only if every schedule loop contained in $(1S)$ is an R-loop.

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 obtain-

able 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,

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

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

$$\varepsilon_n = \sum_{k=1}^{n-1} (\text{the number of R-schedules when } (i = k)) = \sum_{k=1}^{n-1} (\varepsilon_k \times \varepsilon_{n-k}) \quad . \text{(3-4)}$$

The sequence of positive integers generated by (3-4) 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+1}. It can be shown that the sequence generated by (3-4) is given by

$$\varepsilon_n = \frac{1}{n}\binom{2n-2}{n-1}, \text{ for } n = 1, 2, 3, \ldots, \qquad \text{(3-5)}$$

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 (3-5) is $\Omega(4^n/n)$ [Corm90].

For example, the chain-structured SDF graph in figure 3.6 consists of four actors, so (3-5) indicates that this graph has $\frac{1}{4}\binom{6}{3} = 5$ R-schedules. The R-schedules for figure 3.6 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 .

The following theorem establishes that the set of R-schedules always contains a schedule that achieves the minimum buffer memory requirement over all valid single appearance schedules.

Theorem 3.4: Suppose that G is a chain-structured SDF graph; $(delay(\alpha) = 0), \forall(\alpha \in edges(G))$; and S is a valid single appearance schedule for G. Then there exists an R-schedule S' for G such that $buffer\_memory(S') \leq buffer\_memory(S)$.

Proof: We prove this theorem by construction. We use the following notation here: given a schedule loop L and a looped schedule S, we define $nonR(S)$ to be the set of schedule loops in S that are not R-loops; we define $I(L)$ to be the number of iterands of L; and we define $\hat{I}(S) \equiv \sum_{L' \in nonR(S)} I(L')$.

First observe that all chain-structured SDF graphs are consistent so no fur-

ther assumptions are required to assure that valid schedules exist for G, and observe that from Theorem 2.4, there exists a valid fully reduced schedule S_0 for G such that $buffer\_memory(S_0) \le buffer\_memory(S)$.

Now let $L_0 = (nT_1T_2\ldots T_m)$ be an innermost non-R-loop in $(1S_0)$; that is, L_0 is not an R-loop, but every loop nested in L_0 is an R-loop. If $m = 1$ then since S_0 is fully reduced, $L_0 = (n(1T'))$, for some iterand T'. Let S_0' be the schedule that results from replacing L_0 with (nT') in $(1S_0)$. Then clearly, S_0' is also valid and fully reduced, and S_0' generates the same invocation sequence as S_0, so $buffer\_memory(S_0') = buffer\_memory(S_0)$. Also,

$nonR(S_0') = nonR((1S_0)) - \{L\}$, so $\hat{I}(S_0') < \hat{I}((1S_0))$.

If on the other hand $m \ge 2$, we define $S_a \equiv (1T_1)$ if T_1 is an actor and $S_a \equiv T_1$ otherwise (if T_1 is a schedule loop). Also, if T_2, T_3, …, T_m are all schedule loops, we define $S_b \equiv \left(\gamma\left(\frac{I(T_2)}{\gamma}B_2\right)\left(\frac{I(T_3)}{\gamma}B_3\right)\ldots\left(\frac{I(T_m)}{\gamma}B_m\right)\right)$,

where $\gamma = gcd(\{I(T_i) \mid (2 \le i \le m)\})$, and B_2, B_3, …, B_m are the bodies of the loops T_2, T_3, …, T_m, respectively; if T_2, T_3, …, T_m are not all schedule loops, we define $S_b \equiv (1T_2\ldots T_m)$. Let S_0' be the schedule that results from replacing L_0 with $L_0' = (nS_aS_b)$ in $(1S_0)$. It is easily verified that S_0' is a valid, fully reduced schedule and that L_0' is an R-loop, and with the aid of Theorem 2.2, it is also easily verified that $buffer\_memory(S_0') \le buffer\_memory(S_0)$. Finally,

observe that if $m = 3$, then $nonR(S_0') = nonR((1S_0)) - \{L_0\}$, while if $m \neq 3$, then $nonR(S_0') = nonR((1S_0)) - \{L_0\} + \{S_b\}$. Since $I(S_b) = I(L_0) - 1$, it follows that for any value of m, $\hat{I}(S_0') < \hat{I}((1S_0))$.

Thus, from $(1S_0)$, we have constructed a valid, fully reduced schedule S_0' such that $buffer\_memory(S_0') \leq buffer\_memory(S_0) \leq buffer\_memory(S)$ and $\hat{\hat{I}}(S_0') < \hat{\hat{I}}((1S_0))$. By construction, $S_0' = (1T)$, for some iterand T. We define $S_1 \equiv T$. Thus, $buffer\_memory(S_1) \leq buffer\_memory(S)$ and $\hat{\hat{I}}((1S_1)) < \hat{\hat{I}}((1S_0))$.

Clearly, if $\hat{I}((1S_1)) \neq 0$, we can repeat the above process to obtain a valid, fully reduced single appearance schedule S_2 such that $buffer\_memory(S_2) \leq buffer\_memory(S_1)$ and $\hat{\hat{I}}((1S_2)) < \hat{\hat{I}}((1S_1))$. Continuing in this manner, we obtain a sequence of valid single appearance schedules $S_0, S_1, S_2, S_3, \ldots$ such that for each S_i in the sequence with $i > 0$, $buffer\_memory(S_i) \leq buffer\_memory(S)$, and $\hat{\hat{I}}((1S_i)) < \hat{\hat{I}}((1S_{i-1}))$. Since $\hat{\hat{I}}((1S_0))$ is finite, we cannot go on generating S_i's indefinitely — eventually, we will arrive at an S_n, $n \geq 0$, such that $\hat{I}((1S_n)) = 0$. From Fact 3.3, S_n is an R-schedule. *QED*.

Theorem 3.4 guarantees that from within the set of R-schedules for a given chain-structured SDF graph, we can always find a single appearance schedule that minimizes the buffer memory requirement over all single appearance schedules;

however, from (3-5), we know that in general, the R-schedules are too numerous for exhaustive evaluation to be feasible. The following subsection presents a dynamic programming algorithm that obtains an optimal R-schedule in polynomial time.

3.3.2 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, Godb73]. 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 paren-

thesization 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 3.6. 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 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_i is directed from A_i to A_{i+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 $subgraph(\{A_i, A_{i+1}, \ldots, A_j\}, G)$. Then, the minimum buffer memory requirement over all R-schedules for G is $b[1, n]$. If $1 \le i < j \le n$, then,

$$b[i, j] = min(\{(b[i, k] + b[k+1, j] + c_{i,j}[k]) \,|\, (i \le k < j)\}) \quad , \qquad (3\text{-}6)$$

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

$$c_{i,j}[k] = \frac{\mathbf{q}_G(A_k)produced(\alpha_k)}{gcd(\{\mathbf{q}_G(A_m)\,|\,(i \le m \le j)\})} \quad . \tag{3-7}$$

The *gcd* term in the denominator arises because from Fact 2.7, the repetitions vector $\mathbf{q}'$ of $subgraph(\{A_i, A_{i+1}, \ldots, A_j\}, G)$ satisfies

$$\mathbf{q}'(A_p) = \frac{\mathbf{q}_G(A_p)}{gcd(\{\mathbf{q}_G(A_m)\,|\,(i \le m \le j)\})} \quad \text{, for all } p \in \{i, i+1, \ldots, j\} \ .$$

A dynamic programming algorithm derived from the above formulation is specified in figure 3.7. In this algorithm, first the quantity $gcd(\{\mathbf{q}_G(A_m)\,|\,(i \le m \le j)\})$ 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 optimal R-schedule

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 α_i is directed from A_i to A_{i+1}.
output: an R-schedule for G that minimizes the buffer memory requirement.

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

for $i = 1, 2, \ldots, n$ $\text{Subcosts}[i, i] = 0$;
for $\text{chain\_size} = 2, 3, \ldots, n$
  for $\text{right} = \text{chain\_size}, \text{chain\_size} + 1, \ldots, n$
    $\text{left} = \text{right} - \text{chain\_size} + 1$;
    $\text{min\_cost} = \infty$;
    for $i = 0, 1, \ldots, \text{chain\_size} - 2$
      $\text{split\_cost} = (\mathbf{q}_G(A_{\text{left}+i}) / \text{GCD}[\text{left}, \text{right}]) \times produced(\alpha_{\text{left}+i})$;
      $\text{total\_cost} = \text{split\_cost} + \text{Subcosts}[\text{left}, \text{left} + i] + \text{Subcosts}[\text{left} + i + 1, \text{right}]$;
      if $(\text{total\_cost} < \text{min\_cost})$
        $\text{split} = i$; $\text{min\_cost} = \text{total\_cost}$
    $\text{Subcosts}[\text{left}, \text{right}] = \text{min\_cost}$; $\text{SplitPositions}[\text{left}, \text{right}] = \text{split}$;
output $\text{ConvertSplits}(1, n)$; /* Convert the SplitPositions array into an R-schedule */

procedure ConvertSplits(L, R)
implicit inputs: the SDF graph G and the GCD and 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 for $subgraph(\{A_L, A_{L+1}, \ldots, A_R\}, G)$ that minimizes the buffer memory requirement.

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

Figure 3.7.

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 $\Theta(n^3)$.

Theorem 3.5: 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: Let Ξ denote the (**for** $i = 0, 1, \ldots,$ chain_size $- 2$) loop, and denote total the number of iterations of Ξ by $I(\Xi)$. Observe that an iteration of Ξ is carried out for each possible split of each possible subchain in G that contains two or more actors. Now for $k = 2, 3, \ldots, n$, there are exactly $(n - k + 1)$ distinct k-actor subchains, and for each k-actor subchain, there are exactly $(k - 1)$ distinct split positions. Thus,

$$I(\Xi) = \sum_{k=2}^{n} (n - k + 1)(k - 1) \quad . \tag{3-8}$$

It is easily observed that $I(\Xi) \le \sum_{k=1}^{n} n^2 = n^3$, and thus, $I(\Xi)$ is $O(n^3)$.

To see that $I(\Xi)$ is $\Omega(n^3)$, define $z \equiv \left\lfloor \frac{(n-1)}{2} \right\rfloor$, and observe from (3-8) and from the identities $\sum_{i=1}^{m} i = \frac{m(m+1)}{2}$ and $\sum_{i=1}^{m} i^2 = \frac{m(m+1)(2m+1)}{6}$ for $m \in \{1, 2, 3, \ldots\}$, that

$$I(\Xi) = \sum_{k=1}^{(n-1)} k(n-k) \ge \sum_{k=1}^{z} k(n-k) \ge \sum_{k=1}^{z} k(z-k)$$

$$= z\sum_{k=1}^{z} k - \sum_{k=1}^{z} k^2 = \frac{1}{6}z(z+1)(z-1) \quad . \tag{3-9}$$

Now from the definition of z, $z \ge \frac{(n-2)}{2}$, so (3-9) implies that $I(\Xi) \ge \frac{1}{48}(n-1)(n-2)(n-3)$, and thus $I(\Xi)$ is $\Omega(n^3)$. *QED.*

3.3.3 Example: Sample Rate Conversion

The recently introduced 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 3.8(a). It is more efficient to perform the rate change in stages. Rate conversion ratios are chosen by examining the prime factors 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 $147 : 60$. One way to perform this conversion in three stages is $4:3$, $8:7$, and $5:7$. Figure 3.8(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 $\mathbf{q}(A, B, C, D, E) = (147, 49, 28, 32, 160)^T$; the optimal looped schedule given by our dynamic programming approach is $(49(3A)(1B))(4(7C)(8(1D)(5E)))$; and the associated buffer memory requirement is 260. In contrast, the alternative schedule

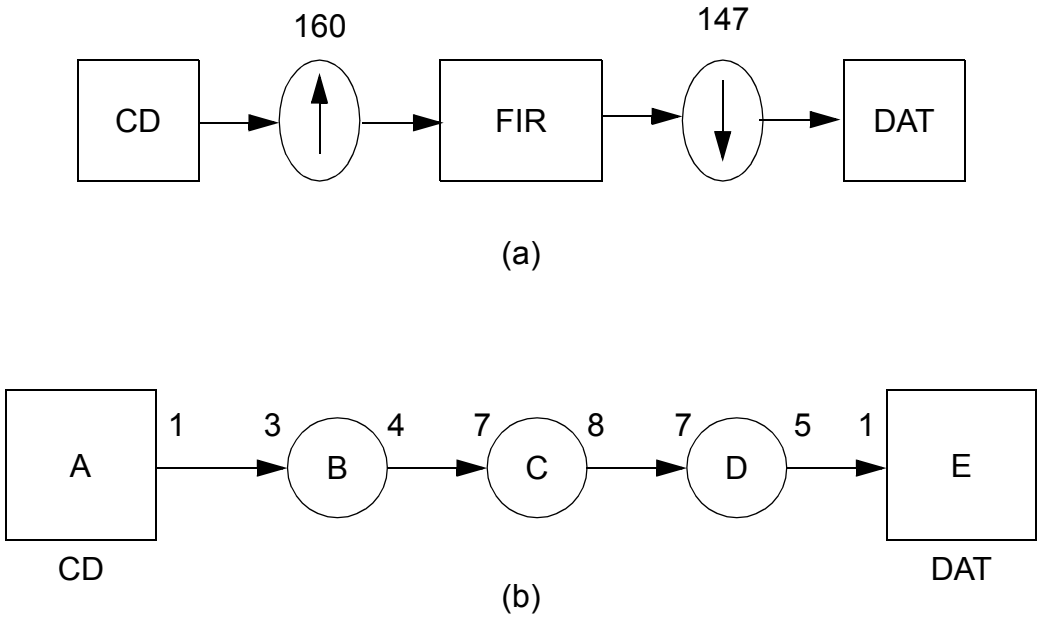


Figure 3.8. (a). CD to DAT sample rate change system.
(b). Multi-stage implementation of a CD to DAT sample rate system.

$(147A)(49B)(28C)(32D)(160E)$ has a buffer memory requirement of 729. This is an important savings with regard to current technology: a buffer memory requirement of 260 will fit in the on-chip memory of most existing programmable digital signal processors, while a buffer memory requirement of 729 is too high for all programmable digital signal processors, except for a small number of the most expensive ones.

3.3.4 Extensions

There are three simple extensions of the dynamic programming solution developed in Subsection 3.3.2. First, the technique applies to the more general class of well-ordered SDF graphs. 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

$$c_{i,j}[k] = \frac{\sum_{\alpha \in S_{i,j,k}} \mathbf{q}_G(A_k) produced(\alpha_k)}{gcd(\{\mathbf{q}_G(A_m) | (i \le m \le j)\})} \quad , \tag{3-10}$$

where

$$S_{i,j,k} \equiv \{\beta | (source(\beta) \in \{A_i, A_{i+1}, \ldots, A_k\}) \quad ; \\ \textbf{and } (sink(\beta) \in \{A_{k+1}, A_{k+2}, \ldots, A_j\})\}$$

that is, $S_{i,j,k}$ is the set of edges directed from one side of the split to the other side.

The dynamic programming technique of Subsection 3.3.2 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 Subsection 3.3.2 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 (3-10).

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

$$CS_{i,j} = \frac{max(\{total\_consumed(\alpha_k) + total\_consumed(\alpha_{k+1}) \,|\, (i \le k < j-1)\})}{gcd(\{\mathbf{q}(A_i), \mathbf{q}(A_{i+1}), \ldots, \mathbf{q}(A_j)\})}$$

for the subchain $A_i, A_{i+1}, \ldots, A_j$. We follow this computation with

$$b'[i,j] = min(\{b[i,j], CS_{i,j}\}) \quad , \tag{3-11}$$

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 sub-chains. Incorporating the techniques of this section with more general overlaying schemes is a topic for future work.

3.4 Related Work

3.4.1 Loop Scheduling in Gabriel

As part of the Gabriel project [Lee89], How [How90] was the one of the first to investigate the problem of scheduling SDF graphs for compact code. The first uniprocessor scheduler for Gabriel did not attempt to minimize code size, and was based on a simple heuristic for minimizing the buffer memory requirement [Lee89, Ho88a]. This heuristic involves deferring the firing of actors whose successors are fireable until all successors have used up the tokens on their input edges, and are no longer fireable. Furthermore, no actor is scheduled twice until all other actors have been tried. The technique is an intuitive way to keep excess tokens from accumulating in buffers, and thus to keep the buffer memory requirement low.

How's first approach to generating compact code was to post-process the minimum buffer memory scheduler with a pattern matching algorithm that finds successively repeated sequences of firings. The scheduler then groups such sequences into schedule loops. Since in this approach, looping is not considered at

all when constructing the ordering of invocations, the technique fails to synthesize compact schedules for even very simple examples. For example, for the simple acyclic SDF graph of figure 3.9, it is easily verified that the minimum buffer mem-

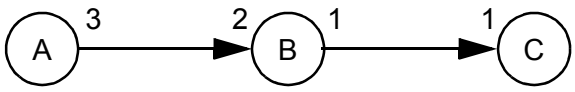


Figure 3.9. An example used to illustrate scheduling techniques used in the Gabriel design environment.

ory heuristic yields the schedule $ABCABCBC$. The most compact schedule that How's post-processor can extract from this is $ABCA(2BC)$, which contains two appearances per actor; since the graph of figure 3.9 is acyclic, valid single appearance schedules exist, and thus, the minimum buffer memory heuristic yields a sub-optimal result both with and without post-processing.

Gabriel's minimum buffer memory heuristic together with How's post-processing approach fails to provide looping opportunities because it does consider looping when it orders the invocations [How90]. Having made this observation, How proposed a technique that analyzes the SDF graph to directly construct repetitive invocation sequences. The technique involves isolating **connected subgraphs of uniform repetition count**[1], abbreviated **CSURC**. Given a connected, consistent SDF graph G, a subgraph G' is a CSURC of G if G' is connected, and there is a positive integer k such that $\mathbf{q}_G(A) = k, \forall A \in actors(G')$. How demonstrated experimentally that detecting and clustering CSURC's often greatly increases code compactness over the minimum buffer memory heuristic with post-

1. How used the term *frequency* in place of *repetition count.*

processing. This is mainly because multirate signal processing systems frequently consist of single sample rate subsystems, with changes in sample rate occurring only at scattered interface points.

Although How's CSURC-based scheduling greatly improves the ability to extract looping from SDF graphs, it has two major limitations. The first shortcoming is illustrated in figure 3.10(a)[1]. Here the clustering of the CSURC

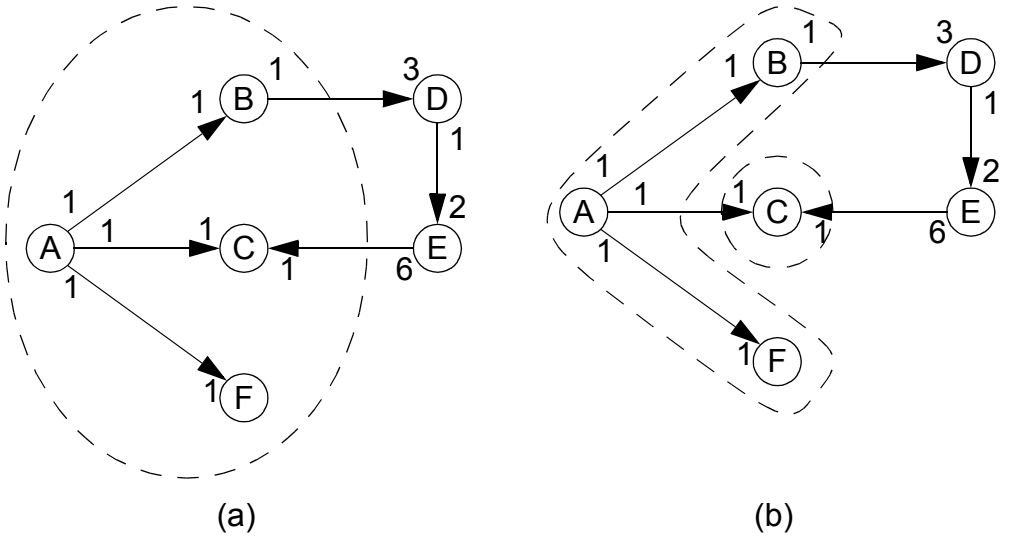


Figure 3.10. An example of how How's CSURC scheduling can lead to deadlock.

$subgraph(\{A, B, C, F\})$ results in a deadlocked graph. The deadlock arises because the root actor A has been subsumed by a hierarchical actor which is no longer a root actor. The execution of the graph must begin with A, but the cluster containing A needs external data to fire. A similar situation can occur when an edge with nonzero delay is subsumed by a CSURC.

Thus, $subgraph(\{A, B, C, F\})$ must be decomposed to retain as large a CSURC as possible without creating a deadlocked graph. The desired partition is

1. This example is taken from [How90].

shown in figure 3.10(b), and a corresponding looped schedule is $(2(3ABF)D)E(6C)$. Unfortunately, How was unable to deduce a general solution to the problem of efficiently decomposing a CSURC in a deadlocked clustered graph.

The second shortcoming of the CSURC approach arises from its inability to detect looping that occurs across changes in repetition count. In figure 3.11, we

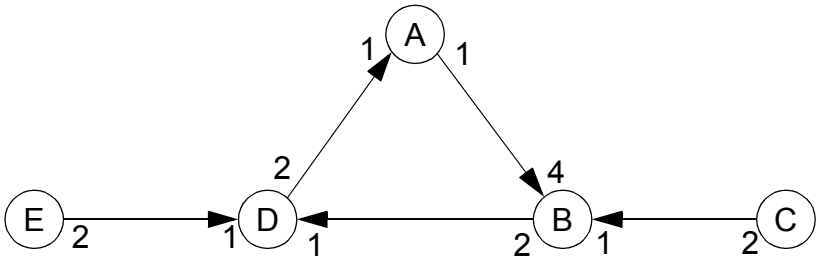


Figure 3.11. An SDF graph that offers opportunities for looping that span changes in repetition count.

show an SDF graph with opportunities for this kind of looping. Here $\mathbf{q}(A, B, C, D, E) = (8, 2, 1, 4, 2)^T$ and $S \equiv C(2E(2D(2A))B)$ is a schedule. Although this schedule reveals that a large amount of looping is inherent in the graph, clearly none of the looping results from CSURC's, since every edge induces a change in repetition count. In this case, the How's CSURC-driven schedule is the same as that produced by the minimum buffer memory heuristic with post-processing, which is $E(2D(2A))ECB(2D(2A))B$. Clearly, this schedule applies significantly less looping than S. It fails to recognize the opportunity to repeat a firing pattern involving A, D and E. As a result, C is allowed to fire midway through the schedule, and this breaks up the nested loop which could have spanned

almost the entire schedule.

In [Bhat93], a technique is described that generalizes How's CSURC scheme to exploit looping opportunities that occur across changes in repetition count. The approach involves constructing the cluster hierarchy in a pairwise fashion by clustering exactly two vertices at each step. The cluster selection is based on frequency of occurrence — the pair of adjacent actors is selected whose associated subgraph has the highest repetition count. This approach favors nested loops over "flat" loop hierarchies, and thus reduces the buffer memory requirement.

The technique of [Bhat93] also included a systematic method for dealing with deadlock. This method maintains the cluster hierarchy on the acyclic precedence graph rather than the SDF graph. Thus, it verifies whether or not a grouping introduces deadlock by checking whether or not it introduces a cycle in the APG. Furthermore, it is shown that this check can be performed quickly by applying a **reachability matrix**, which indicates for any two APG vertices (actor invocations) P_1 and P_2, whether there is a precedence path from P_1 to P_2.

Unfortunately, the storage cost of the reachability matrix proved prohibitive for multirate applications involving very large sample rate changes. Observe that this cost is quadratic in the number of distinct actor invocations in a minimal schedule period. For example, a rasterization actor that decomposes an image into component pixels may involve a change in repetition count on the order of $250{,}000$ to 1. If the rasterization output is connected to homogeneous actor (for example, a gamma level correction), this block alone will produce on the order of $(250{,}000)^2 = 6.25 \times 10^{10}$ entries in the reachability matrix! Thus very large changes in repetition count preclude straightforward application of the reachability matrix; this is unfortunate because looping is most important precisely for such

cases.

In contrast, for SDF graphs that contain no tightly interdependent components, the scheduling framework of Section 3.1 does not require use of the reachability matrix, the acyclic precedence graph, or any other data structure that can become unreasonably large. As mentioned in Section 3.2, our observations suggest that a large majority of practical SDF graphs fall into this category. For SDF graphs that contain tightly interdependent subgraphs, our scheduling framework naturally isolates the minimal subgraphs that require special care. Only when analyzing these tightly interdependent components, may the need arise for reachability matrix analysis, or some other explicit deadlock-detection scheme.

A second limitation of the technique of [Bhat93] is that, although it extracts looping more thoroughly that How's CSURC approach, it fails to process cycles in the graph optimally. This is illustrated in figure 3.12. Figure 3.12(a) depicts a multirate SDF graph, and here $\mathbf{q}(A, B, C) = (10, 4, 5)^T$. Two pairwise clusterings lead to graphs that have valid schedules — $subgraph(\{A, B\})$, having repetition count 2, and $subgraph(\{A, C\})$, having repetition count 5 (the clustering of $subgraph(\{B, C\})$ results in deadlock). Clustering the subgraph with the highest repetition count yields the hierarchical topology in figure 3.12(b), for which the most compact minimal valid schedule is $(2B)(2\Omega_{AC})B\Omega_{AC}B(2\Omega_{AC})$, which yields the schedule $(2B)(2(2A)C)B(2A)CB(2(2A)C)$ for figure 3.12(a). On the other hand, clustering the subgraph of lower repetition count, $subgraph(\{A, B\})$, as depicted in figure 3.12(c), yields the more compact schedule $(2\Omega_{AB})(5C) \Rightarrow (2(2B)(5A))(5C)$.

On the other hand, any loose interdependence algorithm guarantees that a minimum amount of code will be required for any actor that is not contained in a

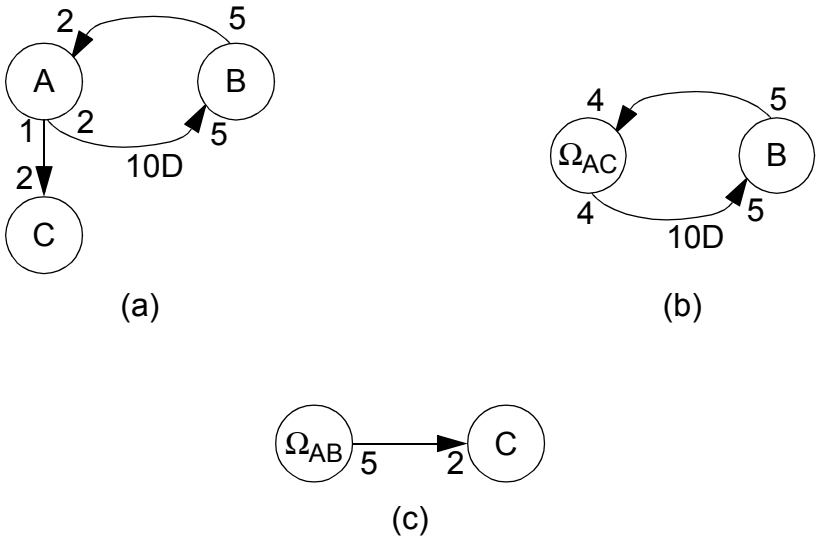


Figure 3.12. This example illustrates how clustering subgraphs based on repetition count alone can conceal looping opportunities that occur within cycles.

tightly interdependent component. As we discussed in Section 3.2, our preliminary observations suggest that tightly interdependent subgraphs are rare in practice, and thus, loose interdependence algorithms guarantee code size optimality for a large class of useful SDF graphs.

3.4.2 Buck's Loop Scheduler

The clustering algorithm developed in Section 3.2 is based largely on part of an alternative technique for constructing compact looped schedules that was developed by Buck [Buck93]. Buck's technique is designed to be more space and time efficient than the technique of [Bhat93], while extracting looping opportunities accross boundaries in repetition count almost as thoroughly. The main space and speed advantages are gained by using simple and efficient heuristics, rather

than a reachability matrix, to decide whether a consolidation of multiple invocations should be avoided.

In addition to applying clustering, Buck's technique employs an alternative mechanism for building hierarchy in which an individual actor A is replaced by an actor $T(A, n)$ that represents n successive invocations of A, for an arbitrary positive integer n. Thus each input edge α of A is replaced by an edge α' that differs only in the sink actor and the consumption parameter — $sink(\alpha') = T(A, n)$ and $consumed(\alpha') = n \times consumed(\alpha)$; and similarly, each output edge β is replaced by an edge β' that has identical parameters, with the exception that $source(\beta') = T(A, n)$ and $produced(\beta') = n \times produced(\beta)$. Buck refers to this process as *looping* actor A with a *loop factor* of n.

Buck's techniqe involves a clustering step, called the *merge pass*, in which adjacent actors that have the same repetition count are clustered; and a looping step, called the *loop pass*, in which selected actors are looped to eliminate mismatches in repetition count between adjacent actors. The merge pass and loop pass are alternated until neither pass produces any transformations, and then the algorithm terminates.

Given an SDF edge α, the merge pass clusters $source(\alpha)$ and $sink(\alpha)$ only if the following three conditions are met

1. $produced(\alpha) = consumed(\alpha)$; and

2. there is no path directed from $source(\alpha)$ to $sink(\alpha)$ that passes through an actor that is not a member of $\{source(\alpha), sink(\alpha)\}$; and

3. α is not contained in a strongly connected component subgraph, or

$$min(\{delay(\alpha') | (source(\alpha') = source(\alpha)) \textbf{ and } (sink(\alpha') = sink(\alpha))\})$$

$$= 0,$$

or

$$min(\{delay(\alpha') | (source(\alpha') = sink(\alpha)) \textbf{ and } (sink(\alpha') = source(\alpha))\})$$

$$= 0.$$

The merge pass repeatedly clusters pairs of adjacent actors that satisfy conditions 1 through 3 until no pairs remain that satisfy the conditions.

The loop pass is divided into two steps — the *integral loop pass* and the *nonintegral loop pass*. In the integral loop pass, a candidate looping opportunity is introduced by each edge α that satisfies $produced(\alpha) = k \times consumed(\alpha)$ or $consumed(\alpha) = k \times produced(\alpha)$, for some positive integer $k > 2$. If the candidate looping opportunity corresponding to α is chosen, then that member $z(\alpha) \in \{source(\alpha), sink(\alpha)\}$ that has higher repetition count (repetitions vector component) is looped with loop factor k. The candidate is selected if the following conditions hold

1. There is no edge α' directed to (from) $z(\alpha)$ such that $produced(\alpha') \neq consumed(\alpha')$, $delay(\alpha') > 0$, *and* α' is a member of a cycle.

2. No actor adjacent to $z(\alpha)$ can be looped to match the repetition count (repetitions vector component) of $z(\alpha)$.

3. After looping $z(\alpha)$, $\{T(z(\alpha), n), z'\}$ satisfies the merge pass clustering conditions, where z' is the single member of $(\{source(\alpha), sink(\alpha)\} - \{z(\alpha)\})$.

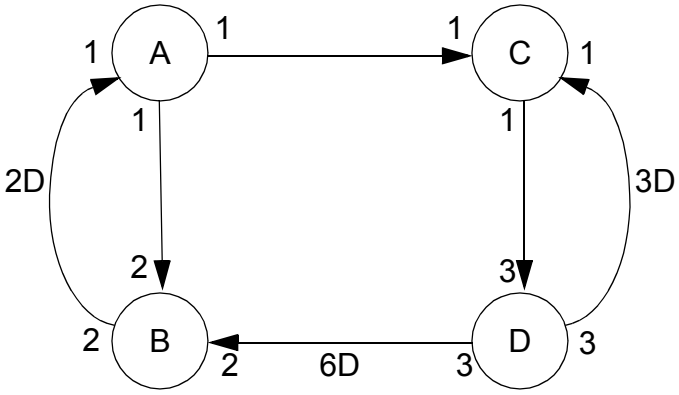
Here, conditions 1 and 3 are sufficient, but not necessary, to avoid deadlock; and condition 2 is provided to favor nested loops, which reduce the buffer

memory requirement over schedules that don't involve nesting [Bhat93].

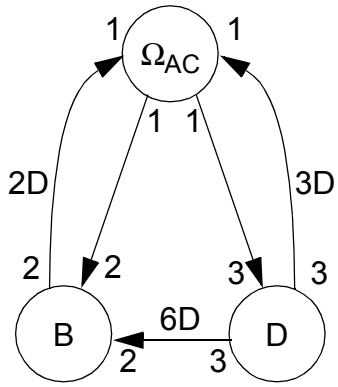
The nonintegral loop pass is designed to accommodate looping opportunities that arise from edges whose production and consumption parameters are not related by integer multiples. Here, the integral loop pass looping conditions are not sufficient to guarantee deadlock-free looping, and thus, the nonintegral loop pass is applied only to graphs that are tree structured or contain only two actors. With this restriction, deadlock avoidance is not an issue, but nonintegral looping opportunities that involve actors in the strongly connected components cannot be exploited.

Together, the merge pass, integral loop pass, and nonintegral loop pass provide a means for rapidly obtaining compact looped schedules. However, since they are based on heuristics, each pass can introduce suboptimalities (with regards to code size). For example, figure 3.13 illustrates how the merge pass can introduce tight interdependence from a graph that has a single appearance schedule. For the graph in figure 3.13(a), $\mathbf{q}(A, B, C, D) = (6, 3, 6, 2)^T$, and $(3(2A)B)(2(3C)D)$ is a valid single appearance schedule. Now observe that the edge $A \rightarrow C$ satisfies the merge pass clustering conditions, and that it is the only edge that satisfies the conditions. Thus, the merge pass clusters $subgraph(\{A, C\})$. It can easily be verified that the graph that results from this clustering, shown in 3.13(b), is tightly interdependent. Hence, a schedule constructed from a cluster hierarchy that includes the result of this merge pass operation cannot be a single appearance schedule.

Observe that in figure 3.13(a) there is only one possible subindependent partition — $\{A, B\}, \{C, D\}$. The merge pass cancels the existence of a single appearance schedule here because it consolidates actors from both sides of the partition, and thus, it destroys the subindependent partition.



(a)



(b)

Figure 3.13. An example that illustrates how Buck's merge pass can fail to preserve the existence of a single appearance schedule.

As an example of how the integral loop pass can introduce suboptimality, consider figure 3.14. For the SDF graph in figure 3.14(a), $\mathbf{q}(A, B, C, D) = (5, 10, 4, 4)^T$ and $(5A)(2(2D)(5B)(2C))$ is a valid looped schedule. No pair of adjacent actors in this graph satisfies the merge pass clustering conditions, so the first transformation of the graph is performed by the integral loop pass. It is easily verified that the edge $A \rightarrow B$ is the only edge that satisfies the integral loop pass conditions, and thus actor B is looped with loop factor 2. The resulting hierarchical SDF graph is shown in figure 3.14(b), and the repetitions vector for this new graph is given by $\mathbf{q}(A, T(B, 2), C, D) = (5, 5, 4, 4)^T$. Examination of this repetitions vector and figure 3.14(b) reveals that the transformation performed by the integral loop pass introduces a tightly interdependent subgraph — $subgraph(\{T(B, 2), C, D\})$. The hierarchical graph that corresponds to the subsequent merge pass operation is

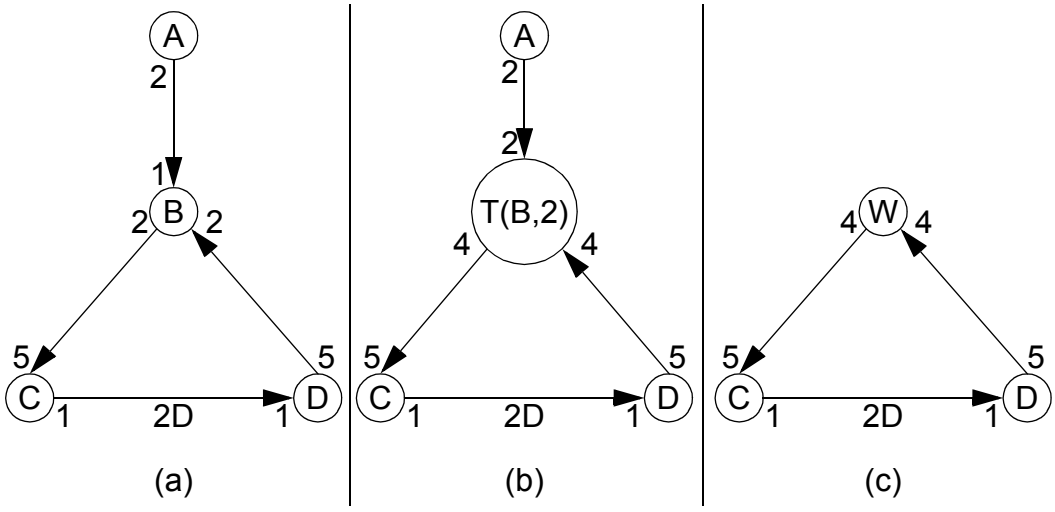


Figure 3.14. An example that illustrates suboptimal performance from Buck's integral loop pass.

shown in figure 3.14(c). As expected, the tight interdependence introduced by the loop pass persists, and we conclude that in this example, the integral loop pass has steered the solution away from a single appearance schedule.

Comparing Buck's merge pass / loop pass scheme with the techniques developed in this thesis reveals a trade-off in compile-time efficiency vs. optimality. Buck's scheduling technique is more time-efficient because it applies only local dataflow information; there is no need to recompute repetitions vectors and repeatedly determine connected and strongly connected components, for example. This same trade-off is observed with How's CSURC approach, but Buck's scheduler is more thorough than How's since it considers looping opportunities that span repetition-count boundaries and it systematically avoids deadlock.

Buck's merge pass directly inspired the clustering technique presented in Section 3.2 for increasing the use of registers in buffering. The merge pass was attractive for this purpose because it handled edges on which the production and consumption parameters are identically unity; it handled many actors that occur frequently in practice; and it was based on a clustering scheme that could easily be incorporated into the framework of loose interdependence algorithms. Our main modification to the merge pass clustering conditions was to replace the condition that there is no "external" path directed from the source actor to the sink actor (condition 2) with the stronger condition that the sink actor in the pairwise cluster candidate must have no predecessors other than the two actors in the candidate cluster. The previous example of figure 3.13 illustrates how a violation of this modified condition can result in suboptimal scheduling. The rigorous theory of looped schedules developed in this thesis allowed us to formally establish that our modification of Buck's merge pass algorithm always preserves code size optimality.

3.4.3 Vectorization

The techniques developed in Sections 3.1 and 3.2 in this thesis are related to techniques for transforming serial procedural programs into programs that are suitable for vector processors. Vector processors are computers that have special operations, called *vector instructions*, for operating on arrays of data. For example, in a vector processor, the following loop can be implemented by a single vector instruction:

```
DO 10 I = 1, 100
    X(I) = Y(I+10) + Z(I+20)
10 CONTINUE
```

A common syntax for the vector instruction corresponding to this loop is

```
X(1:100) = Y(11:110) + Z(21:120)
```

In a vector instruction, the computations of the components of the result vector are independent of one another, so deep pipelines can be employed without any hazards [Kogg81]. Also with a vector instruction, the number of instructions that must be fetched and decoded is reduced; interleaved memories can be exploited to reduce the average time required to read an operand from memory; and the pipeline hazards arising from the loop branch in the original (unvectorized) loop are eliminated[Henn90]. Often, as a consequence of upgrades in computing resources, programs written for conventional scalar processors must be ported to vector processors. Also, from the programmer's viewpoint, it is often more natural or convenient to write serial programs without worrying about efficiently utilizing vector instructions. These considerations have motivated the study of automatic techniques for vectorizing serial procedural programs.

Vectorization algorithms normally operate on a data structure called a *dependence graph*. The dependence graph of a procedural program segment is a directed graph in which each vertex corresponds to a statement of the program. If

v_1 and v_2 are vertices of a dependence graph and s_1 and s_2 are, respectively, the corresponding statements, then there is an edge directed from v_1 to v_2 if it has been determined that some invocation of s_2 is dependent on an invocation of s_1; that is, there exist invocations i_1 and i_2 of s_1 and s_2, respectively, such that executing i_2 before i_1 may be inconsistent with the semantics of the original program.

Unlike the precedence relationships specified by an SDF graph, the dependences in a dependence graph cannot always be determined exactly at compile-time. This is because the programming languages to which dependence graphs are applied are based on more general models of computation than SDF. For example, consider the following FORTRAN code segment in which the value of the variable X is not known at compile-time.

```
          DO 10 I = 1, X
s1:           A(I) = 1
s2:           B(I) = A(100 - I)
          10 CONTINUE
```

Here, s_2 depends on s_1 if and only if $X \geq 50$. Unless it is known that the value of X will definitely be less than 50, there is a dependence graph edge directed from the vertex corresponding to s_1 to the vertex corresponding to s_2.

Another significant difference between SDF graphs and dependence graphs is that SDF graph edges specify iteration implicitly — through mismatches in the production and consumption parameters — whereas with dependence graphs, the repetition of statements results from control-flow structure that is specified explicitly in the corresponding program. With SDF graphs, no control-flow structure exists a-priori, and we must construct one carefully with regards to the available

memory in the target processor before proceeding with other scheduling optimizations. Once the control-flow has been specified for an SDF graph, and code blocks for each actor have been inlined, dependence graphs can be constructed and dependence graph analysis can be applied to further optimize the target program. However, the construction of the initial control-flow structure is a crucial step, and we expect that failure in this step is generally difficult to overcome through post-optimization. For example, recall that How's study [How90], discussed in Subsection 3.4.1, confirmed that pattern matching on a schedule designed for minimum buffer memory requirement does not acceptably minimize the code size. When compiling an SDF graph, the scheduling framework of Section 3.1 can be applied first. If the resulting target program fits within the available processor memory, then post-optimization techniques, such as those that apply dependence graphs, loop unrolling [Dong79], or reorganizing the loop structure to improve memory access locality [Wolf91], can be applied until the remaining memory is exhausted.

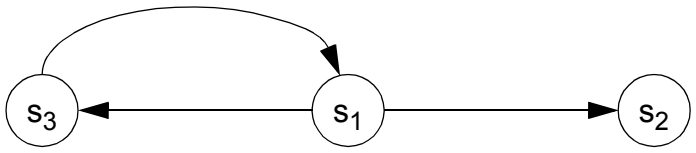
The vectorization problem is similar in structure to the problem of constructing compact looped schedules for SDF graphs since just as strongly connected components in an SDF graph can limit looping opportunities, cycles in a dependence graph limit vectorization. Vectorization is most commonly applied to the innermost loop of a group of nested loops. If the dependence graph for the inner loop is acyclic, then each statement can be vectorized provided that a matching vector instruction exists. If cycles are present, then they are carefully analyzed to see if they can be ignored or if transformations can be applied to eliminate them [Wolf89].

A common tool for vectorization is the *loop distribution* transformation, which was introduced by Muraoka in [Mura71]. In loop distribution, the body of a loop is partitioned into segments, and a separate loop is created for each segment.

As an example of loop distribution, and how it can be applied to vectorization, consider the FORTRAN loop below.

```
          DO 10 I = 1, 10
s1 :          A(I) = B(I) + C(I - 1)
s2 :          D(I) = 2 * A(I)
s3 :          C(I) = A(I) + 5
          10 CONTINUE
```

The dependence graph for this loop is:



We see that s_1 and s_3 form a dependence graph cycle, and that s_2 is not part of any cycle. We can replace the loop with one loop that spans the s_1-s_3 cycle and a second loop for s_2, which can be vectorized. The transformed program that results from this combination of loop distribution and vectorization is shown below.

```
DO 10 I = 1, 10
    A(I) = B(I) + C(I - 1)
    C(I) = A(I) + 5
10 CONTINUE
D(1:10) = 2 * A(1:10)
```

We see that this method of transformation bears similarities with the loose interdependence scheduling framework.

If the target processor has multidimensional vector instructions available, then it may be desirable to vectorize across multiple nested loops[1]. Nested loop vectorization is the form of vectorization that is most closely related to the tech-

niques developed in Section 3.1 of this thesis. Two main approaches to nested loop vectorization have emerged — the *outside-in* vectorization of Allen and Kennedy [Alle87], and the *inside-out* vectorization of Muraoka [Mura71]. Respectively, the relationship between these two techniques is somewhat analogous to the differences between our loose interdependence scheduling framework and the method of Ritz et. al [Ritz93] described in the following subsection.

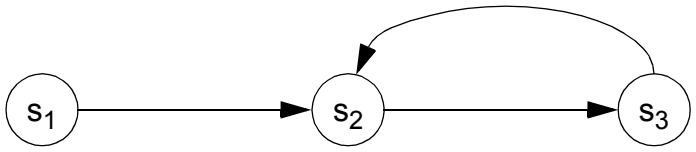
Suppose that $L_1, L_2, \ldots, L_n$ is a sequence of perfectly nested FORTRAN loops; that is, there are no statements between the loops. Suppose that L_1 is the outermost loop, L_2 is the next outermost loop, and so on. In outside-in vectorization, the L_i's are traversed starting with the outermost loop and working inward. First, the dependence graph for $L_1, L_2, \ldots, L_n$ is examined, and loop distribution is applied to isolate strongly connected components and vectorizeable statements. Then, for each strongly connected component, the L_1 loop is fixed and the dependence graph for $L_2, L_3, \ldots, L_n$ is examined. Again, loop distribution is applied, and the method continues recursively on each strongly connected component of the dependence graph for the $L_2, L_3, \ldots, L_n$ combination.

For example, consider the nested loops below.

```
        DO 10 I = 1, 100
            DO 20 J = 1, 100
s1 :            A(I, J) = X(I, J) + Y(I, J)
s2 :            B(I, J) = A(I, J) + C(I - 1, J)
s3 :            C(I, J) = B(I, J) * 6
        20 CONTINUE
        10 CONTINUE
```

1. In [Wolf89], Wolfe states that modern vector processors do not support multidimensional vector instructions, and thus, nested loop vectorization is seldom applied anymore.

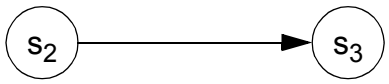
The associated dependence graph is:



Since s_1 is not part of a dependence cycle, it is isolated and vectorized, and this results in the transformed program below.

```
    A(1:100,1:100) = X(1:100,1:100) + Y(1:100,1:100)
    DO 10 I= 1, 100
        DO 20 J = 1, 100
s2:         B(I, J) = A(I, J) + C(I - 1, J)
s3:         C(I, J) = B(I, J) * 6
    20 CONTINUE
    10 CONTINUE
```

Next, the dependence graph for the inner loop is examined:



Since no dependence graph cycles exist, the inner loop can be vectorized, and the final result of applying outside-in vectorization is:

```
A(1:100,1:100) = X(1:100,1:100) + Y(1:100,1:100)
DO 10 I= 1, 100
    B(I,1:100) = A(I,1:100) + C(I-1,1:100)
    C(I,1:100) = B(I, 1:100) * 6
10 CONTINUE
```

This approach bears resemblance to the scheduling framework of loose interdependence algorithms. When scheduling SDF graphs, the outermost loop corresponds to a singe period of the periodic schedule. The strongly connected components of the SDF graph are isolated by the clustering process of step 2 in

figure 3.2. Then, for each strongly connected component, we focus on the next inner loop nesting level of the target program by examining the interdependencies within a minimal schedule period for the given strongly connected component, and attempting to find a subindependent partition. Just as some dependence graph edges disappear as we descend the nesting levels of a group of nested loops, SDF graph edges can become "ignorable" as a loose interdependence algorithm recursively decomposes strongly connected components of an SDF graph. Given a consistent, connected SDF graph G, an edge α does not impose precedence constraints within a minimal schedule period for G if and only if $delay(\alpha) \geq \mathbf{q}_G(sink(\alpha)) \times consumed(\alpha)$. From Fact 2.7, whenever G' is a connected subgraph of G and $A \in actors(G')$, we have $\mathbf{q}_{G'}(A) \leq \mathbf{q}_G(A)$. Thus, as a loose interdependence algorithm decomposes a strongly connected component into finer and finer components, the amount of delay required for a given edge to be ignorable (within a minimal schedule period) decreases, in general.

In contrast to the top-down approach of outside-in vectorization, Muraoka's inside-out vectorization works by examining the innermost loops first and working outward. If both techniques are fully applied, inside-out vectorization and outside-in vectorization yield the same result. However, the outside-in method is computationally more efficient since a statement that can be vectorized for a series of nested loops is examined once rather than repeatedly for each loop.

3.4.4 Minimum Activation Schedules in COSSAP

The techniques in this thesis focus on compiling SDF graphs to minimize the code size and to increase the efficiency of buffering. At the Aachen University of Technology, as part of the COSSAP software synthesis environment for DSP,

Ritz et. al have investigated the minimization of code size in conjunction with a different secondary optimization criterion: minimization of the context-switch overhead, or the average rate at which **actor activations** occur [Ritz93]. An actor activation occurs whenever two distinct actors are invoked in succession; for example, the schedule $(2(2B)(5A))(5C)$ for figure 3.12(a) results in five activations per schedule period. Activation overhead includes saving the contents of registers that are used by the next actor to invoke, if necessary, and loading state variables and buffer pointers into registers. In the code generation system described in [Ritz93], the context-switch overhead also includes a function call, which in turn requires saving the current value of the program counter (the return address of the function call), branching to the location of the function, retrieving the return address when the function is completed, and branching to that return address.

In [Ritz93], the average rate of activations for a periodic schedule S is estimated as the number of activations that occur in one iteration of S divided by the blocking factor of S, and this quantity is denoted by $N'_{act}(S)$. For example, for figure 3.12(a), $N'_{act}((2(2B)(5A))(5C)) = 5$, and $N'_{act}((4(2B)(5A))(10C)) = 9/2 = 4.5$. If for each actor, each invocation takes the same amount of time, and if we ignore the time spent on computation that is not directly associated with actor invocations (for example, schedule loops), then $N'_{act}(S)$ is directly proportional to the number of actor activations per unit time. In practice, these assumptions are seldom valid; however, $N'_{act}(S)$ gives a useful estimate and means for comparing schedules. For consistent acyclic SDF graphs, clearly N'_{act} can be made arbitrarily large by increasing the blocking fac-

tor sufficiently; thus, as with the problem of constructing compact schedules, the extent to which the activation rate can be minimized is limited by the strongly connected components.

The technique developed in [Ritz93] attempts to find the valid single appearance schedule that minimizes N'_{act} over all valid single appearance schedules. The technique applies only to SDF graphs that have single appearance schedules. Minimizing the number of activations does not imply minimizing the number of appearances, and thus, the primary objective of the techniques in [Ritz93] agrees with our primary objective — code size minimization. As a simple example, consider the SDF graph in figure 3.15. It can be verified that for this graph, the

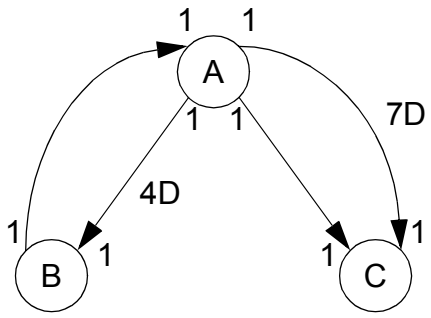


Figure 3.15. This example illustrates that minimizing actor activations does not imply minimizing actor appearances.

lowest value of N'_{act} that is obtainable by a valid single appearance schedule is 0.75, and one valid single appearance schedule that achieves this minimum rate is $(4B)(4A)(4C)$. However, valid schedules exist that are not single appearance schedules, and that have values of N'_{act} below 0.75; for example, the valid schedule $(4B)(4A)(3B)(3A)(7C)$ contains two appearances of A and B, and sat-

isfies $N'_{act} = 5/7 = 0.71$.

In [Ritz93], the **relative vectorization degree** of a fundamental cycle C in a consistent, connected SDF graph G is defined by

$$N_G(C) \equiv max(\{min(\{D_G(\alpha') \mid \alpha' \in parallel(\alpha)\}) \mid \alpha \in edges(C)\}) \quad , \tag{3-12}$$

where $D_G(\alpha) \equiv \left\lfloor \frac{delay(\alpha)}{total\_consumed(\alpha, G)} \right\rfloor$ is the delay on edge α normalized by the total number of tokens consumed by $sink(\alpha)$ in a minimal schedule period of G, and

$$parallel(\alpha) \equiv \{\alpha' \in edges(G) \mid source(\alpha') = source(\alpha) \textbf{ and } sink(\alpha') = sink(\alpha)\}$$

is the set of edges with the same source and sink as α. For example, if G denotes the SDF graph in figure 3.12(a) and χ denotes the cycle in G whose associated graph contains the actors A and B, then $D_G(\chi) = \left\lfloor \frac{10}{20} \right\rfloor = 0$; and if G denotes the graph in figure 3.15 and χ denotes the cycle whose associated graph contains A and C, then $D_G(\chi) = \left\lfloor \frac{7}{1} \right\rfloor = 7$.

Ritz et. al postulate that given a strongly connected SDF graph, a valid single appearance schedule that minimizes N'_{act} can be constructed from a **complete hierarchization**, which is a cluster hierarchy such that only connected subgraphs are clustered, all cycles at a given level of the hierarchy have the same relative vectorization degree, and cycles in higher levels of the hierarchy have strictly higher relative vectorization degrees that cycles in lower levels. Figure 3.16 depicts a

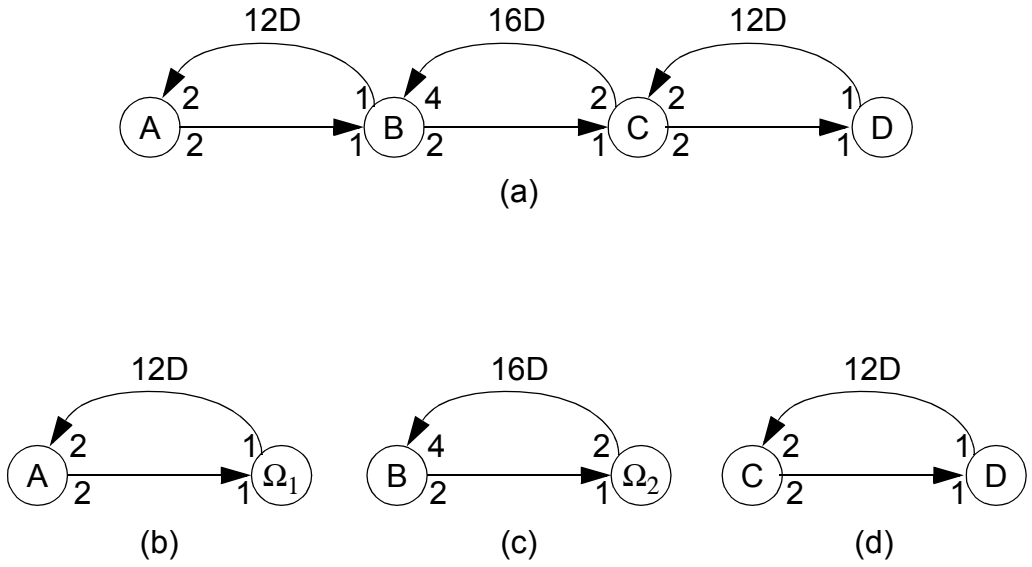


Figure 3.16. A complete hierarchization of a strongly connected SDF graph.

complete hierarchization of an SDF graph. Figure 3.16(a) shows the original SDF graph; here, $\mathbf{q}(A, B, C, D) = (1, 2, 4, 8)^T$. Figure 3.16(b), shows the top level of the cluster hierarchy. The hierarchical actor Ω_1 represents $subgraph(\{B, C, D\})$, and this subgraph is decomposed as shown in figure 3.16(c), which gives the next level of the cluster hierarchy. Finally, figure 3.16(d), shows that $subgraph(\{C, D\})$ corresponds to Ω_2 and is the bottom level of the cluster hierarchy.

Now observe that the relative vectorization degree of the fundamental cycle in figure 3.16(c) with respect to the original SDF graph is $\left\lfloor \frac{16}{8} \right\rfloor = 2$, while

the relative vectorization degree of the fundamental cycle in figure 3.16(b) is $\left\lfloor \frac{12}{2} \right\rfloor = 6$; and the relative vectorization degree of the fundamental cycle in figure 3.16(c) is $\left\lfloor \frac{12}{8} \right\rfloor = 1$. Thus, we see that the relative vectorization degree decreases as we descend the hierarchy, and thus the hierarchization depicted in figure 3.16 is complete. The hierarchization step defined by each of the SDF graphs in Figures 3.16(b)-(d) is called a **component** of the overall hierarchization.

The technique described in [Ritz93] constructs a complete hierarchization by first evaluating the relative vectorization degree of each fundamental cycle, determining the maximum vectorization degree, and then clustering the graphs associated with the fundamental cycles that do not achieve the maximum vectorization degree. This process is then repeated recursively on each of the clusters until no new clusters are produced. In general, this bottom-up construction process has unmanageable complexity; for example, in the worst case, the number of fundamental cycles in a directed graph is $\sum_{i=1}^{n-1} \binom{n}{n-i+1}(n-1)!$ [John75]. However, this normally doesn't create problems in practice since the strongly connected components of useful signal processing systems are often small, particularly in large grain descriptions.

Once a complete hierarchization is constructed, the technique of [Ritz93] constructs a schedule "template" — a sequence of loops whose iteration counts are to be determined later. For a given component Π of the hierarchization, if v_Π is the vectorization degree associated with Π, then all fundamental cycles in Π contain at least one edge α for which $D_G(\alpha) = v_\Pi$. Thus, if we remove from Π all

edges in the set $\{\alpha | D_G(\alpha) = v_\Pi\}$, the resulting graph is acyclic, and if $F_{\Pi,1}, F_{\Pi,2}, \ldots, F_{\Pi,n_\Pi}$ is a topological sort of this acyclic graph, then valid schedules exist for Π that are of the form $T_\Pi \equiv (i_\Pi(i_{\Pi,1}F_{\Pi,1})(i_{\Pi,2}F_{\Pi,2})\ldots(i_{\Pi,n_\Pi}F_{\Pi,n_\Pi}))$. This is the subschedule template for Π.

Here, each $F_{\Pi,j}$ is a vertex in the hierarchical SDF graph G_Π associated with Π. Thus, each $F_{\Pi,j}$ is either a **base block** — an actor in the original SDF graph G — or a hierarchical actor, which represents the execution of a periodic schedule for the corresponding subgraph of G. Now let A_Π denote the set of actors in G that are contained in G_Π and in all hierarchical subgraphs nested within G_Π; and let $k_\Pi \equiv gcd(\{i_{\Pi,j} | 1 \le j \le n_\Pi\})$. Thus we have

$$i_{\Pi,j} = k_\Pi \mathbf{q}_{G_\Pi}(F_{\Pi,j}), j = 1, 2, \ldots, n_\Pi \tag{3-13}$$

In [Ritz93], it is stated that number of activations that T_Π contributes to N'_{act} is given by $\frac{|B_\Pi| q_G(A_\Pi)}{k_\Pi}$, where B_Π is the set of base blocks in G_Π. Thus, if H denotes the set of hierarchical components in the given complete hierarchization, then

$$N'_{act} = \sum_{\Pi \in H} \frac{|B_\Pi| q_G(A_\Pi)}{k_\Pi}. \tag{3-14}$$

In the proposed technique, an exhaustive search over all i_Π and k_Π is carried out to minimize (3-14). The search is restricted by constraints derived from the requirement that the resulting schedule for G be valid. As with the construction of complete hierarchizations, it is argued that the simplicity of strongly connected components in most practical applications permits this expensive evaluation scheme.

As with the techniques presented in Sections 3.1 am 3.2 of this thesis, the minimum activation scheduler of [Ritz93] provides a solution for constructing schedules that minimize code size. However, with regards to scheduling for minimum code size, the solution in this thesis is more general for three reasons. First, our scheduling framework guarantees code size optimality for all actors that lie outside the tightly interdependent components, and thus, it handles graphs that do not have single appearance schedules. In contrast, the techniques of [Ritz93] apply only to SDF graphs that have single appearance schedules.

Second, the minimum code size scheduler of [Ritz93] is designed for the specific secondary goal of minimizing actor activations. In contrast, our scheduling framework can be adapted to different secondary optimization goals. For example, the clustering techniques of [Bhat93] can be incorporated into the acyclic scheduling algorithm to minimize the buffer memory requirement; the technique of Section 3.3 can be applied to any chain-structured graphs that arise in the cluster hierarchy; and the technique related to Theorem 3.3 can be employed to increase the use of registers.

Finally, we have demonstrated that loose interdependence algorithms exist that have polynomial time-complexity. In contrast, the solution of [Ritz93] does not have polynomially-bounded complexity, and it will rapidly become infeasible if the input graph is sufficiently complicated. Fortunately, this threshold will rarely

be reached by the systems for which the technique was designed — large grain specifications of signal processing algorithms.

Despite the differences in generality, for the specific purpose of jointly minimizing code size and actor activation rate for SDF graphs that have single appearance schedules, the method of [Ritz93] is superior to that proposed in this thesis. Furthermore, since the techniques of [Ritz93] require an optimization pass that traverses all levels of the cluster hierarchy, it is unlikely that these techniques can be directly incorporated into our scheduling framework, which restricts the component algorithms to operate only on one specific level of the hierarchy at a time.

3.4.5 Thresholds

Constructing looped schedules for SDF graphs that minimize actor activations is related to the concept of **thresholds**, which is discussed by Allen and Kennedy [Alle87] in the context of compiling FORTRAN programs into code for vector computers. As a simple example, consider the FORTRAN code fragment in figure 3.17(a). Due to the recurrences in the body of this loop, loop distribution cannot be applied and none of the statements can be vectorized. However, if we "split" the loop up as shown in figure 3.17(b), loops amenable to distribution and vectorization emerge. Figure 3.17(c) shows the result of applying distribution and vectorization to the inner loops of figure 3.17(b).

The transformation from the loop in figure 3.17(a) to the loop in figure 3.17(b) is an application of thresholds. A threshold is loosely defined as the minimum number of iterations that elapse between the definition of a variable and its use in a dependence. Thus, if we can construct an inner loop whose iteration count is equal to one less than the threshold, then this inner loop may be amenable to dis-

```
    DO 100 I = 1, 100
        Y(I) = F1(X(I-5))
        Z(I) = F2(Y(I))
        X(I) = F3(W(I-10), Z(I))
        W(I) = F4(X(I))
100 CONTINUE
```

(a)

```
    DO 100 I = 1, 10
        DO 90 J = 1, 2
            DO 80 K = 1, 5
                II = 10*(I-1) + 5*(J-1) + K
                Y(II) = F1(X(II-5))
                Z(II) = F2(Y(II))
                X(II) = F3(W(II-10), Z(II))
80 CONTINUE
90 CONTINUE
        DO 70 J = 1, 10
            II = 10*(I-1) + J
            W(II) = F4(X(II))
70 CONTINUE
100 CONTINUE
```

(b)

```
    DO 100 I = 1, 10
       DO 90 J = 1, 2
           II = 10*(I-1) + 5*(J-1)
           Y(II+1:II+5) = F1(X(II-4:II))
           Z(II+1:II+5) = F2(Y(II+1:II+5))
           X(II+1:II+5) = F3(W(II-9:II-5),Z(II+1:II+5))
90 CONTINUE
       W(10*I - 9:10*I) = F4(X(10*I - 9:10*I))
100 CONTINUE
```

(c)

Figure 3.17. An illustration of thresholds, and of a relationship between thresholds and minimum activation schedules for SDF graphs.

tribution and/or vectorization. This transformation is particularly useful for vector machines in which vector instructions outperform equivalent scalar instruction sequences for short vector lengths; that is, if the start-up overhead for performing a vector instruction is small compared to the execution time of a scalar instruction.

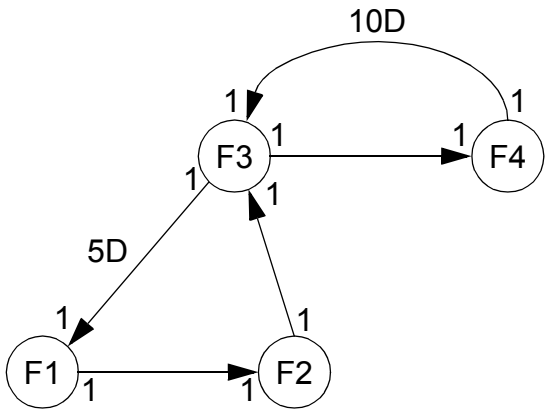


Figure 3.18. An SDF graph that corresponds to the dependence relationships in figure 3.17(a).

In some cases, the problem of applying thresholds efficiently can be solved by constructing a minimum activation schedule for a homogeneous SDF graph. For example, the dependence relationships in figure 3.17(a) can be modeled by the homogeneous SDF graph depicted in figure 3.18. Here, the actors correspond to the subroutines F1, F2, F3 and F4 in figure 3.17, and each edge corresponds to one of the arrays W, X, Y or Z. It can be verified that $(2(5F1)(5F2)(5F3))(10F4)$ is a single appearance schedule that minimizes the activation rate for figure 3.18, and the correspondence between this schedule and the vectorized FORTRAN code of figure 3.17(c) is easily seen.

The problem of applying thresholds is in some ways more general, and in some ways less general than the problem of scheduling SDF graphs to minimize

activations. It is more general because complicated patterns of data transfers — for example data dependent, multi-dimensional, or nonlinear patterns — can be specified by arbitrary FORTRAN statements whereas in SDF graphs, each edge always corresponds to a linear stream of data with the producing and consuming computations offset by a constant amount (the edge delay) that is known at compile time. On the other hand, the threshold application problem is less general because in its underlying model of computation, each fundamental operation consumes and produces a single data value. Thus, unlike the SDF case, there is no issue of repetition and looping arising implicitly from mismatches in production and consumption parameters along data dependence edges.

4

INCREASING THE EFFICIENCY OF BUFFERING

4.1 Introduction

Ho [Ho88a] developed the first compiler for pure SDF semantics. The compiler, part of the Gabriel design environment [Lee89], was targeted to the Motorola DSP56000 programmable digital signal processor and the code that it produced was markably more efficient than that of existing C compilers. However, due to its inefficient implementation of buffering, the compiler could not match the quality of good handwritten code, and the disparity rapidly worsened as the granularity of the graph decreased.

The mandatory placement of all buffers in memory, rather than in registers, is a major cause of the high buffering overhead in Gabriel. Although this is a natural way to compile SDF graphs, it can create an enormous amount of overhead when actors of small granularity are present. This is illustrated in Figure 4.1. Here, a graphical representation of an atomic addition actor is placed alongside typical assembly code that would be generated if straightforward buffering tactics are used. The target language is assembly language for the Motorola DSP56000, *input1* and *input2* represent memory addresses where the operands to the addition

actor are stored, and *output* represents the location in which the output token will be buffered.

In Figure 4.1, observe that four instructions are required to implement the addition actor. Simply augmenting the compiler with a register allocator and a mechanism for considering buffer locations as candidates for register-residence can reduce the cost of the addition to three, two, or one instruction. The Comdisco Procoder graphical DSP compiler [Powe92] demonstrates that integrating buffering with register allocation can produce code comparable to the best manually-written code.

The Comdisco Procoder's performance is impressive, however the Procoder framework has one major limitation: it is primarily designed for homogeneous SDF, and thus, it becomes less efficient when multiple sample rates are present. Furthermore, the techniques apply only when the buffers can be mapped *statically* to memory. In general, this need not be the case, and we will elaborate on this topic in Section 4.2.

In this chapter, we develop compile-time analysis techniques to optimize the buffering of SDF graphs that involve multiple sample rates. Multirate buffers

Figure 4.1. An illustration of inefficient buffering for an SDF graph.

are often best implemented as contiguous segments of memory to be accessed by indirect addressing, and thus they cannot be mapped to machine registers. Efficiently implementing such buffers requires reducing the amount of indexing overhead. We show that for SDF, there is a large amount of information available at compile-time that can be used to optimize the indexing of multirate buffers. Also, multirate SDF graphs may lead to very large memory requirements if large sample rate changes are involved, and this problem is compounded by the presence of schedule loops. Thus, it may be highly desirable to overlay noninterfering buffers in the same physical memory space as much as possible. This chapter presents ways to analyze the dataflow information to detect opportunities for overlaying buffers that can be incorporated into best-fit and related memory allocation schemes.

We begin by reviewing the important code generation issues that are pertinent to multirate SDF graphs. In Section 4.2, we present a classification of buffers based on dataflow properties and we discuss these different categories with respect to storage requirements. The following three sections present code optimization techniques. Section 4.3 discusses minimizing spills of address registers to memory. Section 4.4 examines the problem of overlaying buffers for compact memory allocation. Section 4.5 considers optimization opportunities that apply to circular buffers. Finally, Section 4.6 presents a detailed summary of the proposed methods.

4.1.1 Code Generation for Looped Schedules

An important code generation issue for looped schedules is the accessing of a buffer from within a schedule loop. The difficulty lies in the requirement for different invocations of the same actor to be executed with the same block of instructions. As a simple example, consider Figure 4.2, which shows a multirate

SDF graph, a looped schedule for the graph, and an outline of assembly code that could efficiently implement this schedule. In the code outline, the statement "do #n LABEL" specifies n successive executions of the block of code between the *do* statement and the instruction at location *LABEL*. Thus, the successive firings of actor B are carried out with a loop. This requires that both invocations of B must access their inputs with the same instruction, and that the output data for A be stored in a manner that can be accessed iteratively. This in turn suggest writing the data produced by A to successive memory locations, and having B read this data using the register autoincrement or autodecrement addressing modes that are typical in programmable digital signal processors. Here, the output tokens of A are stored in successive locations buf and buf + 1, and B reads these values into local register x0 through the autoincremented buffer pointer r2.

The techniques in this chapter do not depend on a specific language for

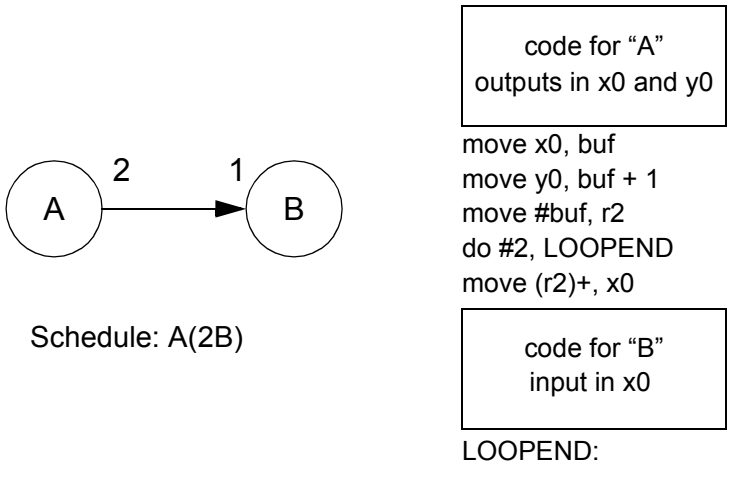


Figure 4.2. An example of compiled code for a looped schedule.

defining the actors. However the techniques are best-suited when actor inputs and outputs are referenced symbolically, and the assignment of machine registers and memory locations is performed by the compiler, as in the Comdisco Procoder [Powe92]. In this type of actor definition language, a simple addition actor might have the following as its defining code block:

```
add in1, in2, out
```

It is left to the compiler to replace in1, in2, and out with register references and to make sure that data is routed appropriately between the registers. For example, if the adder is executed through a loop, and this loop does not contain the actor whose output is consumed by input port in1, it is generally desirable to load the register corresponding to in1 through an address register. This is the case with the input to actor *B* in Figure 4.2. Alternatively, the schedule may permit data to be exchanged directly through registers, in which case the generated code might look like:

```
add r0, r1, r2
add r2, r3, r4
```

(this corresponds to a cascade of adders).

Another important code generation issue is register allocation, which is critical both for data and address registers. Scheduling heuristics for improving register allocation in homogeneous SDF block diagrams are discussed in [Powe92]. These techniques can be applied to homogeneous subgraphs in multi-rate graphs in conjunction with clustering techniques, such as those presented in Section 3.2. A recently-developed approach to register allocation studied by Hendren et al. [Hend92] appears promising for multirate code generation. In this technique, a hierarchy of circular-arc graphs is extracted from nested loop code, and

heuristics for coloring this class of graphs are applied. The techniques developed in this chapter do not depend on a specific method of register allocation.

We conclude this subsection with two definitions.

Definition 4.1: Given an SDF graph G, a looped schedule S for G, and an actor A in G, a **common code space set**, abbreviated **CCSS**, for A is the set of invocations of A that are represented by some appearance of A in S.

A CCSS is thus a set of invocations carried out by a given sequence of instructions in program memory (code space). For example, consider the looped schedule $(4A)C(2B(2C)BC)(2BC)$ for the SDF graph in Figure 4.3(a). The CCSS's for this looped schedule are $\{A_1, A_2, A_3, A_4\}$, $\{C_1\}$, $\{B_1, B_3\}$, $\{C_2, C_3, C_5, C_6\}$, $\{B_2, B_4\}$, $\{C_4, C_7\}$, $\{B_5, B_6\}$, and $\{C_8, C_9\}$.

It will be useful to examine the *flow* of common code space sets. This can be depicted with a directed graph, called the **CCSS flow graph**, that is largely analogous to the *basic block* graph [Aho88] used in conventional compiler techniques. Each CCSS corresponds to a vertex in the CCSS flow graph, and an edge is inserted from a CCSS δ_1 to a CCSS δ_2 if and only if there are invocations $A_i \in \delta_1$ and $B_j \in \delta_2$ such that B_j is invoked immediately after A_i. To illustrate CCSS flow graph construction, Figure 4.3(b) shows the CCSS flow graph associated with the schedule $(4A)C(2B(2C)BC)(2BC)$ for the SDF graph in Figure 4.3(a).

4.1.2 Modulo Addressing

Most programmable DSPs offer a *modulo* addressing mode, which can be

used in conjunction with careful buffer sizing to alleviate the memory cost associ-

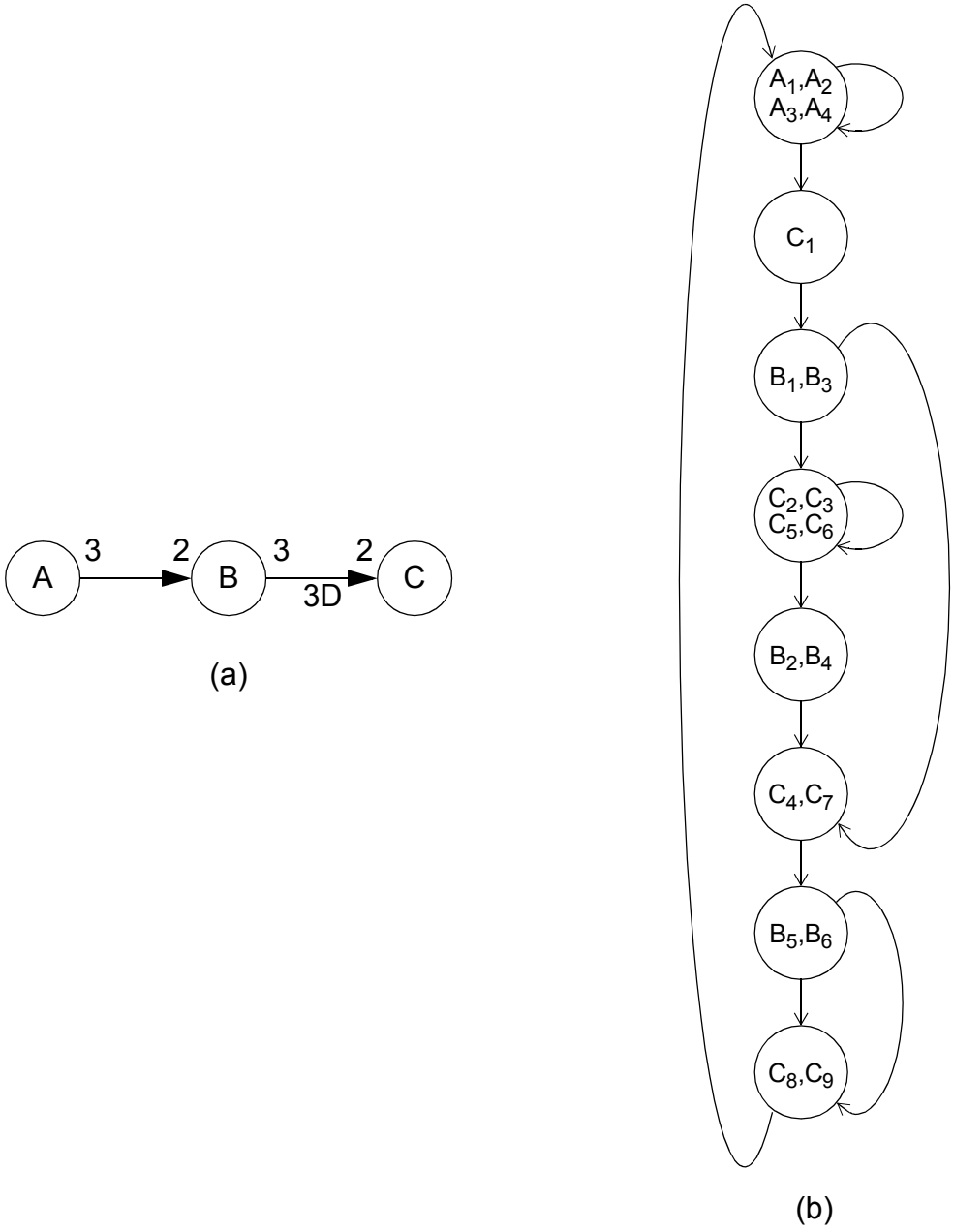


Figure 4.3. An illustration of common code space sets and the CCSS flow graph.

ated with requiring buffer accesses to be sequential. This addressing mode allows for efficient implementation of circular buffers, for which indices need to be updated modulo the length of the buffer so that they can wrap around to the other end.

For example, in the Motorola DSP56000 programmable DSP, a modifier register MX is associated with each address register RX. Loading MX with an integer $n > 0$ specifies a circular buffer of length $n + 1$. The starting address of the buffer is determined by the value v that is stored in RX. If we let b denote the value obtained by clearing the $\lceil \log_2(n+1) \rceil$ least significant bits of v, then assuming that $b \le v \le (b+n)$, an autoincrement access (RX)+ updates RX to $b + ((v - b + 1) \bmod (n+1))$.

Figure 4.4 illustrates the use of modulo addressing to decrease memory

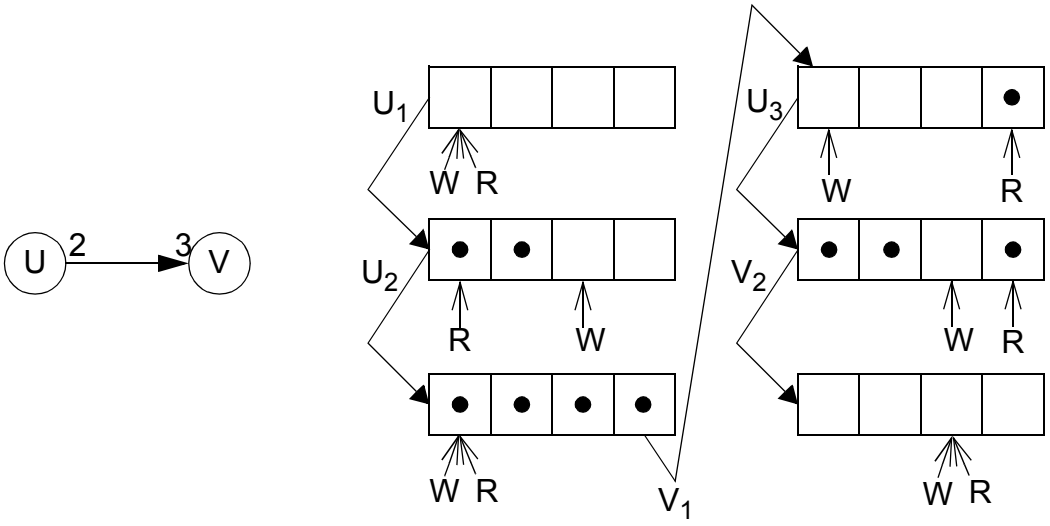


Figure 4.4. An illustration of modulo addressing.

requirements when sequential buffer access is needed. The schedule $U(2UV)$ would clearly require a buffer size of 6 for iterative access if only linear addressing is available. However, as the sequence of buffer diagrams in Figure 4.4 shows, only four memory locations are required when postincrement modulo addressing is used. W and R respectively denote the write pointer for actor U and the read pointer for V, and a black dot inside a buffer slot indicates a **live token** — a token that has been produced but not yet consumed. Note that the accesses of the second invocation of U and the second invocation of V wrap around the end of the buffer.

Observe also that the pointers R and W can be reset at the beginning of each schedule period to point to the beginning of the buffer, and thus the access patterns depicted in Figure 4.4 could be repeated every schedule period. This would cause the locations in each buffer access to be **static** — fixed for every schedule period — and hence they would be known values at compile time.

This illustration renders false the previous notion that for static buffering, the total number of tokens exchanged on an edge per schedule period must always be a multiple of the buffer size. As we will show in the following section, the requirement holds only when there is a nonzero delay associated with the edge in question.

4.2 Buffer Parameters

To guide memory allocation and code generation, we must determine four qualities of each buffer — the **logical size** of the buffer, whether the buffer occupies a region of contiguous memory locations, whether the accesses to the buffer are static, and whether the buffer is circular or linear. By the logical size of a buffer, we mean the number of memory locations required for the buffer if it is

implemented as a single contiguous block of memory. For example, the buffer for the graph in Figure 4.4 will have a logical size of four or six depending, respectively, on whether or not we are willing to pay the cost of resetting the buffer pointers before the beginning of every schedule period. In Section 4.4, we will show that it may be desirable to implement a buffer in multiple nonadjacent segments of physical memory.

Note that in our model of buffering, as in Figure 4.4, each token is read (consumed) from the same memory location that it is produced into, and thus there is no rearrangement of live tokens in the physical memory space.

4.2.1 Static vs. Dynamic

For an SDF edge α, static buffering means that for both $source(\alpha)$ and $sink(\alpha)$, the ith token accessed in any schedule period resides in the same memory location as the ith token accessed in any other schedule period [Lee87]. A buffer that is not static is called a **dynamic buffer**. From our discussion of Figure 4.4, it is clear that if $delay(\alpha) = 0$, static buffering can occur with a logical buffer size equal to the maximum number of live tokens that coexist in the buffer. However, if α has nonzero delay, then we must impose the additional constraint that $total\_consumed(\alpha)$ is some positive integral multiple of the buffer length.

The need for this constraint is illustrated in Figure 4.5. Here, the minimum buffer size according to the rule for zero delay is four, since up to four tokens concurrently exist in the buffer for the given schedule. Figure 4.5 shows the succession of buffer states if a buffer of this length is used. Since there is a delay on the edge, there will always be a token in the buffer at the beginning of each schedule period — this is the first token consumed by invocation V_1. For static buffering,

we need this "delay token" — which is consumed in the schedule period *after* it is produced — to reside in the same memory location every schedule period. Comparison of the initial and final buffer states in Figure 4.5 reveals that this is not the case since the write pointer W did not wrap around to point to its original location. Clearly, W could have returned to its original position if and only if the total number of advances made by W (six, in this case) was an integer multiple of the buffer length. But the total number of advances made by W is simply $total\_consumed(U \rightarrow V)$. We summarize with the following theorem.

Theorem 4.1: For a given schedule, the logical buffer size n must satisfy the following two conditions:

(1). n cannot be less than the maximum number of live tokens that coexist on the corresponding edge α.

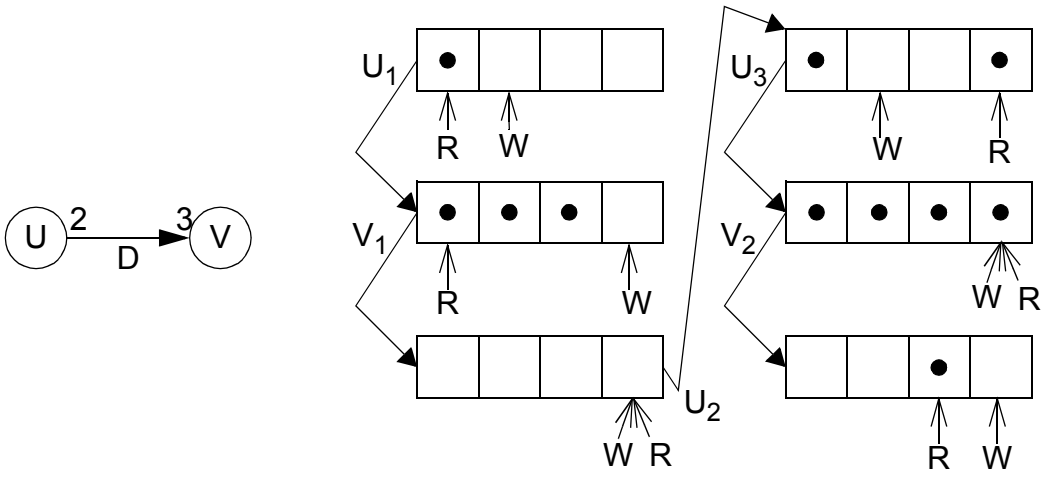


Figure 4.5. The effect of delay on the minimum buffer size required for static buffering.

(2). If $delay(\alpha) = 0$, then static buffering is possible with any logical buffer size that meets criterion (1). Otherwise, static buffering is possible if and only if for some positive integer k, $total\_consumed(\alpha) = kn$.

Thus, static buffering for an edge with delay may require additional storage space — 50 % more in the case of the example in Figure 4.5. The difference may be negligible for most buffers, but it must be kept in mind when sample rates are very high. Further trade-offs between static and dynamic buffering are discussed in Section 4.3.

4.2.2 Contiguous vs. Scattered

Once we have decided whether a buffer is to be static or dynamic, we may decide upon whether it will be a **contiguous buffer**, occupying a section of successive physical memory locations, or whether the buffer may be **scattered** through memory. Scattered buffering allows more flexibility in memory allocation, which can lead to lower memory requirements. However, as we discussed in Section 4.1, contiguity constraints between the location of successive buffer accesses may be imposed by loops in the schedule. Similarly, loops that are contained in actor code blocks lead to contiguity constraints.

Dynamic buffering also induces contiguity constraints. In dynamic buffering, no invocation accesses the logical buffer at the same offset every schedule period. To see this, suppose that some invocation A_j accesses a buffer Υ for some edge α at the same offset every schedule period. Since the buffer pointer for A_j advances $total\_consumed(\alpha)$ positions from one schedule period to the next, it follows that $total\_consumed(\alpha)$ must be a positive integer multiple of the logical buffer size of Υ, and thus the buffer must be static. Thus, a dynamic buffer cannot

be implemented with only absolute addressing, and if an actor A accesses a dynamic buffer, the current position in the buffer must be maintained as a state variable of A. We will elaborate on the contiguity requirements for dynamic buffering in Section 4.4.

An important aspect of the physical layout of a buffer is the effect on total storage requirements. The locations of a scattered buffer are not restricted to be mapped to continuous memory addresses, and graph coloring [Golu80] can be used to assign physical memory locations to the set of scattered buffers. If all scattered buffers correspond to delayless edges, then the interference graph becomes an interval graph, and interval graphs can be colored with the minimum number of colors in time that is linear in the number of vertices and edges [Carl91]. The presence of delay on one or more of the relevant edges complicates graph-coloring substantially. A delay results in a token that is read in a schedule period after the period in which it is written, and thus the lifetime of the token crosses one or more iterations of the program's outermost loop. The resulting interference graphs belong to the class of circular-arc graphs [Hend92]. Finding an optimal coloring for this class of graphs is intractable, but effective heuristics have been demonstrated [Hend92].

When subsets of variables must reside in contiguous locations, we expect that the memory requirements will increase since this imposes additional constraints on the storage allocation problem. Until further insight is gained about this effect, we cannot accurately estimate how much more memory will be required if a particular scattered buffer is changed to a contiguous buffer. However, since optimally compact storage layout requires scattered buffers, it is likely that when data-memory requirements are severe, edges should be implemented as scattered buffers whenever possible. We will discuss storage optimization further in Section 4.4.

4.2.3 Linear vs. Modulo

For each contiguous buffer, we must determine whether modulo address updates will be required to make the buffer pointer wrap around the end of the buffer. Such modulo address updates normally require overhead; the amount of overhead varies from processor to processor. For instance, recall the discussion in Section 4.1.2 regarding the Motorola DSP56000's hardware support for modulo address generation. Here a "modifier register" must be loaded with the buffer size before modulo updates can be performed on the corresponding address register, so there is a potential overhead of one instruction every time the buffer pointer is swapped into the register file. When there is no hardware support for modulo addressing, as with general purpose microprocessors such as the MIPS R3000 [Kane87], the modulo update must be performed in software every time the buffer is accessed. This typically requires an overhead of several instructions for each buffer access.

In Section 4.5, we will present general techniques for eliminating modulo accesses. Presently, we conclude that circular buffering may potentially introduce execution-time overhead. For edges with delay, this risk is unavoidable — circular buffers are mandatory. However, for some delay-free edges, it may be preferable to forego the data-memory savings offered by modulo buffering so that the overhead can be avoided. For an SDF edge α, a buffer size of $total\_consumed(\alpha)$ clearly guarantees that no modulo accesses will be required — provided that we reset the buffer pointer at the start of every schedule period. Smaller buffer sizes (divisors of $total\_consumed(\alpha)$ which meet or exceed the maximum number of coexisting tokens) are also possible, but one must first verify that no access within a loop wraps around the buffer. This expensive check is rarely worth the effort. A simple rule of thumb can be used to decide whether to switch to linear buffering for a

delayless edge: we prioritize each delayless edge α by the "urgency measure" μ defined by

$$\mu(\alpha) = \left\lceil \frac{total\_consumed(\alpha)}{\text{minimum buffer size of } \alpha} \right\rceil \times \left\lceil \frac{1}{total\_consumed(\alpha) - (\text{minimum buffer size of } \alpha)} \right\rceil \quad (4\text{-}1)$$

The first bracketed term is the number of modulo accesses that occur on each end of α every schedule period, and the denominator in the second term is the storage cost to convert this edge to a static buffer of size $total\_consumed(\alpha)$. Thus $\mu(\alpha)$ denotes the number of modulo accesses eliminated per unit of additional storage. We simply convert the edges with the highest μ values until we have exhausted the remaining data memory. Many variations on this scheme are possible, and architectural restrictions on the layout of storage, such as multiple independent memories [Lee88b] may require modification.

4.3 Increasing the Efficiency of Static Buffers

The storage economy of dynamic buffering comes at the expense of potential execution-time overhead. When a pointer to a dynamic buffer is swapped out of its physical register, it is mandatory that its value be spilled to memory so that the next time the pointer is used, it can resume from the correct position in the buffer. With static buffering, we know the offset at which every invocation accesses the buffer. Thus we can resume buffer addressing with an immediate value and there is no need to spill the pointer to memory. As a result, every time a buffer pointer of the source or sink actor is swapped out, dynamic buffering

requires an extra store to memory.

For instance, consider the example in Figure 4.6. Here, it is easily verified

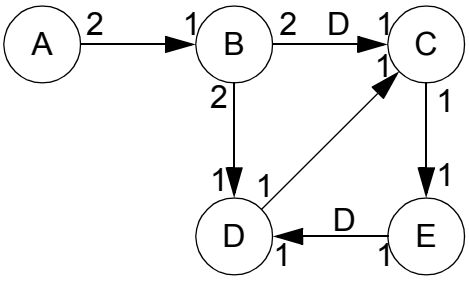


Figure 4.6. An example of how loops can limit the advantages of static buffering.

that $\mathbf{q}(A, B, C, D, E) = (1, 2, 4, 4, 4)^T$. Since $total\_consumed(B \rightarrow C) = 4$, a buffer of size four suffices for static buffering on the edge $B \rightarrow C$. Now, the code block for actor C must access $B \rightarrow C$ through some physical address register R , and R must contain the correct buffer position C_{rp} every time the code block is entered. If it is not possible to dedicate R to C_{rp} for the entire inner loop $(2DCE)$, then R must be loaded with the current value of C_{rp} just prior to entering the code block for C . Since the code block executes invocations C_1 , C_2 , C_3 , and C_4 — the members of the associated CCSS — and each of these invocations accesses the buffer at a different offset, we cannot load R with an immediate value. The value to load into R must be obtained from a memory location and the

current value of C_{rp} must be written into this location whenever R is swapped out. It can easily be verified that at most three tokens coexist on $B \rightarrow C$ at any given time, and thus a dynamic buffer of size three could implement the edge. Since the organization of loops precludes exploiting the static information of a length four buffer, dynamic buffering is definitely preferable in this situation.

It is not always the case that different members of a CCSS access a static buffer at different offsets. As an illustration of this, consider again the SDF graph in Figure 4.3(a), and consider the looped schedule $(4A)C(2B(2C)BC)(2BC)$. We can tabulate the offsets for every buffer access in the program to examine the access patterns for each CCSS. Such a tabulation is shown in Table 4.1, assuming that static buffers of length 12 and 6 are used for the edges $A \rightarrow B$ and $B \rightarrow C$, respectively. The *access port* column specifies the different actor-edge incidences

access port	invocation	offset
$(A \rightarrow B) \gg B$	1	0
	2	2
	3	4
	4	6
	5	8
	6	10
$B \gg (B \rightarrow C)$	1	3
	2	0
	3	3
	4	0
	5	3
	6	0

access port	invocation	offset
$(B \rightarrow C) \gg C$	1	0
	2	2
	3	4
	4	0
	5	2
	6	4
	7	0
	8	2
	9	4
$A \gg (A \rightarrow B)$	1	0
	2	3
	3	6
	4	9

Table 4.1. A tabulation of the buffer access patterns associated with the schedule $(4A)C(2B(2C)BC)(2BC)$ for the SDF graph in Figure 4.3(a).

in the SDF graph. For example, $A \raisebox{0.5ex}{\scriptsize »} (A \rightarrow B)$ refers to the connection of edge $A \rightarrow B$ as the output edge of actor A, and $(B \rightarrow C) \raisebox{0.5ex}{\scriptsize »} C$ refers to the connection of edge $B \rightarrow C$ as the input edge of actor C. The *invocation* column lists the firings of the actor with the associated access port, and the offset at which the ith invocation of this actor references the access port is given in the ith offset entry for the access port. Examination of Table 4.1 reveals that the members of CCSS $\{C_4, C_7\}$ read from edge $B \rightarrow C$ at the same offset. Similarly, the write accesses of the common code space sets $\{B_1, B_3\}$ and $\{B_2, B_4\}$ occur respectively at the same offsets. If all members of a CCSS Υ access an edge α at the same offset, we say that Υ **accesses** α **statically**.

Thus, when a pointer into a static buffer is spilled, and the pointer is accessed elsewhere from within a loop, it is not always necessary to spill the pointer to memory. The procedure for determining whether a spill is necessary at a given swap point can be conceptualized easily in terms of the CCSS flow graph, which we introduced in Section 4.1.1. Suppose that a buffer pointer associated with actor A and edge α must be swapped out of its register at some point in the program. First, we must determine the vertex x in the CCSS flow graph that corresponds to this swap point. From x, we traverse all paths until they either reach the end of the program, they traverse the same vertex twice (they traverse a cycle), or they reach an occurrence of a CCSS for A. We are interested only in the *first* time a path encounters a CCSS for A. Let P be the set of all paths p directed from x that reach a CCSS for A before traversing any vertex twice, and let $A(p)$ denote

the first CCSS for A that p encounters. Then the buffer pointer must be spilled to memory if and only if the set P contains a member that does not access α statically.

Traversing paths at every spill may be extremely inefficient. Instead, we can perform a one-time analysis of the loop organization to construct a table containing the desired reachability information. The concept is similar to the conventional global dataflow analysis problem of determining which variable definitions reach which parts of the program [Aho88]. However, our problem is slightly more complex. In global dataflow analysis, we need to know which variable definitions are live at a given point in the program. For eliminating buffer pointer spills, we need to know which points in a program can reach a given CCSS *without passing through another CCSS for the same actor*. This information can be summarized in a boolean table that has each entry indexed by an ordered pair of common code space sets (C_1, C_2). The entry for (C_1, C_2) will be true if and only if there is a control path directed from C_1 to C_2 that does not pass through another CCSS for the actor that corresponds to C_2. We refer to this table as the **first-reaches table**, since it indicates the points (the common code space sets) at which control first reaches a given actor from a given CCSS. Table 4.2 shows the first-reaches table for the looped schedule $(4A)C(2B(2C)BC)(2BC)$. The CCSS flow graph corresponding to this schedule is depicted in Figure 4.3(b).

The first-reaches table can be systematically constructed by a technique, specified in [Bhat92], that is based largely on the methods described in [Aho88] for computing reaching definitions. An important difference is that a separate pass through the loop hierarchy is required to construct the columns associated with each actor, whereas reaching definitions can all be dealt with in a single pass. In

practice, however, we are concerned only with the columns of the first-reaches matrix that correspond to actors that access multiword contiguous buffers, so often a large number of passes can be skipped.

To fully assess the benefits of choosing static buffering over dynamic buffering for a particular edge, we must consult the first-reaches table at every spill point. Performing this check for every multiword buffer is very expensive. Instead, we should generally perform this check only for the sections of the program that are executed most frequently.

	A_1 A_2 A_3 A_4	C_1	B_1 B_3	C_2 C_3 C_5 C_6	B_2 B_4	C_4 C_7	B_5 B_6	C_8 C_9
A_1,A_2,A_3,A_4	T	T	T	F	F	F	F	F
C_1	T	F	T	T	F	F	F	F
B_1,B_3	T	F	F	T	T	F	F	F
C_2,C_3,C_5,C_6	T	F	F	T	T	T	F	F
B_2,B_4	T	F	T	F	F	T	T	F
C_4,C_7	T	F	T	T	F	F	T	T
B_5,B_6	T	F	T	F	F	F	T	T
C_8,C_9	T	T	T	F	F	F	T	T

Table 4.2. The first-reaches table associated with the looped schedule $(4A)C(2B(2C)BC)(2BC)$ (the corresponding CCSS flow graph is shown in Figure 4.3(b)). The entry corresponding to a row CCSS X and a column CCSS Y is *true* ("T") if and only if there is a control path directed from X to Y that does not pass through another CCSS for the actor that corresponds to Y.

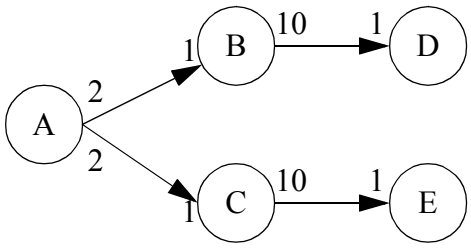
4.4 Overlaying Buffers

When large sample rate changes are involved, assigning each buffer to a single contiguous block of physical memory may require more data memory space than what is available. In this section, we show how to fragment buffers in physical memory, which can expose more opportunities for overlaying [Fabr82]. This technique can be used to improve first-fit, best-fit, and related storage optimization schemes, which are frequently applied to memory allocation for variable sized data items. In [Fabr82], Fabri has studied more elaborate storage optimization schemes that incorporate a generalized interference graph. Such schemes are also compatible with the methods developed in this section.

4.4.1 Fragmenting Buffer Lifetimes

Figure 4.7 illustrates how lifetime analysis and fragmentation information can be used to reduce storage requirements. Here, a multirate SDF graph is depicted along with a looped schedule for the graph and the resulting buffer lifetime profiles. In the first profile, each edge is treated as an indivisible unit with respect to storage allocation. We see that this straightforward designation of buffer lifetimes reveals no opportunities to share storage and thus the edges $A \rightarrow B$, $A \rightarrow C$, $B \rightarrow D$, and $C \rightarrow E$ require 2, 2, 10, and 10 units of storage, respectively, for a total of 24 units.

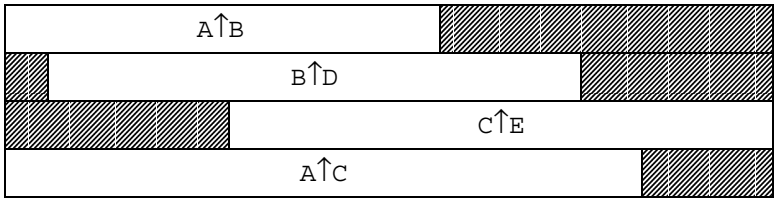
Notice, however, that the invocations that access $B \rightarrow D$ can be divided into two sets $\{B_1, D_1, D_2, \ldots, D_{10}\}$ and $\{B_2, D_{11}, D_{12}, \ldots, D_{20}\}$ such that all tokens are produced in the same set in which they are consumed — there is no interaction among the two sets. Thus, they can be considered as separate units for



Schedule: AB(10D)C(10E)B(10D)C(10E)

(a)

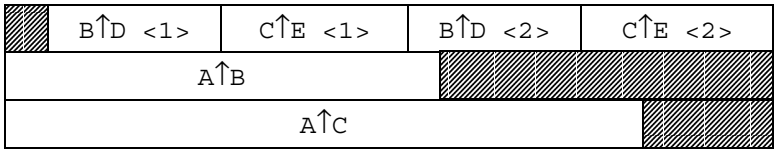
$A_1\ B_1\ D_1 \ldots D_{10}\ C_1\ E_1 \ldots E_{10}\ B_2\ D_{11} \ldots D_{20}\ C_2\ E_{11} \ldots E_{20}$



Aggregate Buffer Lifetimes

(b)

$A_1\ B_1\ D_1 \ldots D_{10}\ C_1\ E_1 \ldots E_{10}\ B_2\ D_{11} \ldots D_{20}\ C_2\ E_{11} \ldots E_{20}$



Buffer Period Lifetimes

(c)

Figure 4.7. An illustration of opportunities to overlay buffers based on the periodicity of accesses.

storage allocation, with lifetimes ranging from B_1 through D_{10}, and B_2 through D_{20}, respectively. We call these two invocation subsets the **buffer periods** of $B \to D$, and we denote them by successive indices as $B \to D\langle 1\rangle$ and $B \to D\langle 2\rangle$. The concept of a buffer period will be defined precisely in the next subsection. The live range for $C \to E$ can be decomposed similarly and the resulting lifetime profile is depicted in Figure 4.7(c) (we suppress the "$\langle 1\rangle$" index for edges that have only one buffer period). This new profile reveals that we can map both $B \to D$ and $C \to E$ to the same 10-unit block of storage, because even though the aggregate lifetimes of these edges conflict, the buffer periods do not. Thus, the memory requirement for buffering can be reduced almost in half to 14 units of storage.

This fragmentation technique can be exploited by first-fit, best-fit, and related storage optimization schemes. In such schemes, we maintain a list of variables along with their sizes and lifetimes; if a variable x becomes live earlier than another variable y, then x occurs earlier in the list than y. Also, we maintain a free-list of unallocated contiguous segments of memory. At each step, we remove the head of the variable list from the list, and we assign it to a free memory block for the duration of the variable's lifetime. In first-fit allocation, we choose the first free block of sufficient size, while in best fit, we choose the free block of sufficient size whose size differs from the size of the variable by the least amount. In general, best-fit leads to more compact allocation, while first-fit is computationally more efficient.

For example, if we use the aggregate buffer lifetimes in Figure 4.7(b), then neither first-fit, best-fit, nor any other storage allocation scheme will achieve any overlaying between the four variables to be allocated, and 24 units of storage are

required. On the other hand, the fragmented buffer information in Figure 4.7(c) separates the items to be allocated into six variables. It can easily be verified that both first-fit and best-fit allocation require only 14 units of storage to achieve a valid storage layout for Figure 4.7(c).

4.4.2 Computing Buffer Periods

There are four mechanisms that can impose contiguity constraints on successive buffer accesses of an edge α — writes to α occurring from a loop inside $source(\alpha)$; reads from α occurring from a loop inside $sink(\alpha)$; placement of $source(\alpha)$ or $sink(\alpha)$ within a schedule loop; and dynamic buffering. The constraints imposed by these mechanisms can be specified as subsets of tokens that must be buffered in the same block of storage. For example, suppose that for the SDF graph in Figure 4.8(a), actor A is programmed so that it writes its output tokens from within a single loop inside the actor code block. The resulting contiguity constraints are illustrated in Figure 4.7(b) — the three tokens produced by each invocation must be stored in three adjacent memory locations. We specify these two constraints by the subsets $\{A[1], A[2], A[3]\}$ and $\{A[4], A[5], A[6]\}$, where $A[i]$ represents the ith token accessed by A in a minimal schedule period[1], for $1 \le i \le total\_consumed(A \rightarrow B)$. The constraints that result if the reads of actor B occur from within a loop inside the actor are depicted in Figure 4.8(c), and we represent these constraints as $\{B[1], B[2]\}$, $\{B[3], B[4]\}$, and $\{B[5], B[6]\}$. However, since we must ultimately superimpose all constraints, we would like to express them in terms of the same actor.

1. This notation assumes that the edge in question (in this case $A \rightarrow B$) is understood. Also, for simplicity, we assume that the blocking factor is one; however, the analysis in this section generalizes easily to any finite blocking factor.

Our convention is to express all contiguity constraints for an edge in terms of the source actor. Thus, noting the unit delay on $A \rightarrow B$, we translate Figure 4.8(c) to $\{A[6], A[1]\}$, $\{A[2], A[3]\}$, and $\{A[4], A[5]\}$.

As a more complete example, consider the SDF subgraph in Figure 4.9, which we use to represent a common cascade of multirate DSP actors. Here, actor E represents an 8-fold *upsampler*, which consumes one token per invocation and outputs a token with the same data value along with 7 zero-valued tokens; and

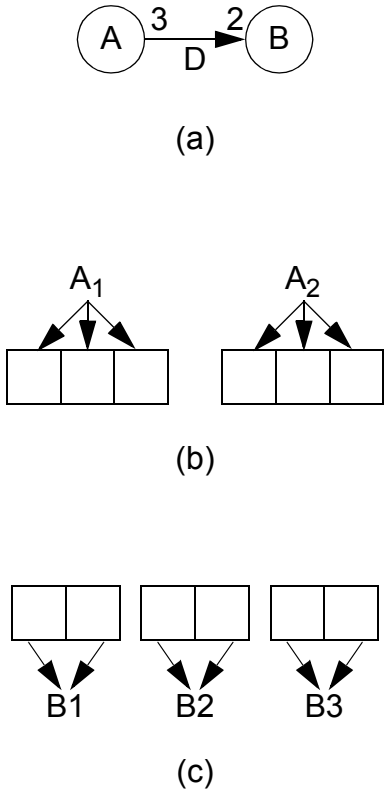


Figure 4.8. An illustration of buffering constraints when edges are accessed through loops inside actor definitions.

actor F represents a 5-fold *decimator*, which consumes 5 tokens and outputs one token with the same data value as the first token consumed. For clarity, we have specified E to be a simple form of upsampler; however, similar contiguity constraints can apply to more elaborate upsamplers, such as an upsampler that performs linear interpolation. Now if clustering $subgraph(\{E, F\})$ in the enclosing graph does not produce deadlock, then it is easily verified that the looped schedule $(2EF)(3E(2F))$ can be used to invoke this subsystem. Figure 4.10 shows a possible implementation of this loop schedule if the target language is C.

Now, from examination of the code blocks for E in Figure 4.10, we see that in each invocation of E, the last seven accesses of the output edge (all but the first) are generated from within a loop inside the corresponding code block for E. Thus, we constrain the last seven data values output by E to be written to contiguous memory locations. This leads to the contiguity constraints $\{E[1]\}$, $\{E[2-8]\}$, $\{E[9]\}$, $\{E[10-16]\}$, $\{E[17]\}$, $\{E[18-24]\}$, $\{E[25]\}$, $\{E[26-32]\}$, $\{E[33]\}$, $\{E[34-40]\}$, where we have used $E[i-j]$ as shorthand notation for $E[i], E[i+1], \ldots, E[j]$. If there were no other contiguity constraints for the output edge of E, the tokens $\{E[1]\}$,

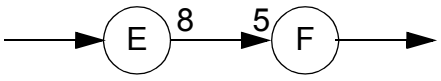


Figure 4.9. An SDF subgraph that represents a cascade of an upsampler and a decimator.

```
{ /* begin subschedule for subgraph({E,F}) */
    /* initialize read and write pointers for
    edges that are internal to the subgraph */
    E_writeptr = 0;
    F_readptr = 0;

    for (i=0; i<2; i++) {
        /* begin code block. for CCSS {E1,E2} */
        temp1 = E_inpbuf[E_readptr++];
        E_outbuf[E_writeptr++] = temp1;
        for (i2=0; i2<7; i2++) {
            E_outbuf[E_writeptr++] = 0;
        }
        /* end code block for CCSS {E1,E2} */

        /* begin code block for CCSS {F1,F2} */
        temp2 = E_outbuf[F_readptr++];
        F_outbuf[F_writeptr++] = temp2;
        F_readptr += 4; /* skip over next 4 tokens */
        /* end code block for CCSS {F1,F2} */
    } /* end schedule loop (2 E F) */

    for (i=0; i<3; i++) {
        /* begin code block. for CCSS {E3-E5} */
        temp1 = E_inpbuf[E_readptr++];
        E_outbuf[E_writeptr++] = temp1;
        for (i2=0; i2<7; i2++) {
            E_outbuf[E_writeptr++] = 0;
        }
        /* end code block for CCSS {E3-E5} */

        for (i2=0; i2<2; i2++) {
            /* begin code block for CCSS {F3-F8} */
            temp2 = E_outbuf[F_readptr++];
            F_outbuf[F_writeptr++] = temp2;
            F_readptr += 4; /* skip over next 4 tokens */
            /* end code block for CCSS {F3-F8} */
        } /* end schedule loop (2 F) */
    } /* end schedule loop (3 E (2 F)) */
} /* end subschedule for subgraph({E,F}) */
```

Figure 4.10. An example of C code that can be used to implement the looped schedule $(2EF)(3E(2F))$ for the subsystem of Figure 4.9.

$\{E[9]\}$, $\{E[17]\}$, $\{E[25]\}$ and $\{E[33]\}$ could be mapped to five distinct memory locations, and the sets of tokens $\{E[2-8]\}$, $\{E[10-16]\}$, $\{E[18-24]\}$, $\{E[26-32]\}$, and $\{E[34-40]\}$ could be mapped to five independent seven-unit blocks of contiguous storage. However, due to additional contiguity constraints that arise due to schedule loops, which we discuss below, this flexibility cannot be exploited for the implementation in Figure 4.10.

As with computing the contiguity constraints that arise from intra-actor loops, determining the constraints due to schedule loops is straightforward. Given an edge α, and an actor $N \in (\{source(\alpha)\} \cup \{sink(\alpha)\})$, each outermost schedule loop L in the periodic schedule defines a constraint set that consists of all access by N of α that occur within L. We can derive these from the contiguous ranges of invocations of A and B that L encapsulates. We map all accesses within a loop to the same physical block of memory because we cannot easily perform isolated resets of read/write pointers inside loops. Expensive schemes — such as testing the loop index to determine which physical buffer to use or maintaining an array of buffer locations — are required to fragment buffering within a loop. We do not consider such schemes presently because we expect that their benefits are rare, and thus we consolidate accesses within loops to the same physical buffers.

For the example of Figures 4.9 and 4.10, the given looped schedule is $(2EF)(3E(2F))$. This schedule has two outermost schedule loops, $(2EF)$ and $(3E(2F))$, and thus two constraint sets emerge. The first schedule loop encapsulates the first two invocations of E, which together produce tokens $\{E[1-16]\}$, and the first two invocations of F, which consume tokens $\{E[1-10]\}$. Taking the union of these two sets gives us the constraint set imposed by the outermost loop $(2EF)$ — $\{E[1-16]\}$. The other outermost loop, $(3E(2F))$, encapsu-

lates the third through fifth invocations of E, which produce $\{E[17-40]\}$, and the third through eighth invocations of F, which consume $\{E[11-40]\}$. Taking the union yields $\{E[11-40]\}$ as the constraint set imposed by the outermost loop $(3E(2F))$. Thus, the two outermost loops of $(2EF)(3E(2F))$ respectively impose the constraint sets $\{E[1-16]\}$ and $\{E[11-40]\}$. Since these two constraint sets overlap (over the tokens $E[11-16]$), they are equivalent to a single constraint set that is obtained by taking their union — $\{E[1-40]\}$.

Thus, the schedule loops in Figure 4.10 impose a single constraint set on the edge $E \rightarrow F$, and this is the set $\{E[1-40]\}$. It follows that for the given schedule, $E \rightarrow F$ must be mapped to a single block of contiguous memory — fragmentation cannot be performed. In Figure 4.10, the single block of contiguous memory for $E \rightarrow F$ is implemented by the array *E_outbuf*.

So far we have only mentioned that dynamic buffering can also lead to constraint sets, but we have not described this effect. The effects of dynamic buffering, which are more subtle than the conditions imposed by loops, will be discussed fully in Subsection 4.4.3.

For an SDF edge α, the constraint sets due to intra-actor looping, inter-actor looping (schedule loops), and dynamic buffering together define the logical sections of a buffer that are restricted to contiguous segments of physical memory. We also include the trivial singleton constraints $\{A[1]\}, \{A[2]\}, \ldots, \{A[total\_consumed(\alpha)]\}$, where $A = source(\alpha)$, which we need to account for tokens that don't appear in any of the other constraint sets. We refer to the entire collection of constraint sets, including the single-

ton constraints, as the *collection of constraint sets imposed on* α. Then, determining the buffer periods, which can be viewed as the maximal independent constraint sets, amounts to partitioning the entire collection into maximal nonintersecting subsets.

Definition 4.2: Given an SDF graph G, an edge α in G, and a looped schedule S for G, let $C = C_1, C_2, \ldots, C_k$ denote the collection of constraint sets imposed on α. Suppose that $b = \{b_1, b_2, \ldots, b_n\}$ is a subset of C such that

(1). No member of b is independent of all other members of b — if $n > 1$, then for each b_i, there is at least one $b_j \neq b_i$ such that $b_i \cap b_j \neq \varnothing$; *and*

(2). b is independent of the remainder of C — that is, $\left(\bigcup_{1}^{\cdot\cdot} b_z \right) \cap \left(\bigcup_{E \in (C - b)} E \right) = \varnothing$.

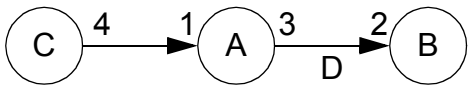
Then $\left(\bigcup_{1}^{\cdot\cdot} b_z \right)$ is called a **buffer period** for α.

One can easily verify that for a given schedule, each edge α has a unique partition into buffer periods. Furthermore, tokens in the same buffer period must be mapped to the same contiguous physical buffer, whereas distinct buffer periods can be mapped to different segments of memory. Finally, the amount of memory required for a buffer period is simply the maximum number of coexisting live tokens in that buffer period.

Figure 4.11 depicts an example that we will use to illustrate the consolidation of different constraint sets into buffer periods. The schedule of Figure 4.11(a) does not contain any loops. If the buffer accesses within A or B do not occur

within intra-actor loops, then only the singleton constraint sets apply to $A \rightarrow B$, and the buffer periods are $\{A[1]\}, \{A[2]\}, \ldots, \{A[12]\}$.

Now suppose that all accesses of $A \rightarrow B$ by A are performed from within a loop inside A. The corresponding constraint set is shown in the second row of Figure 4.11(b), and we obtain the resulting buffer periods by superimposing the first two rows of Figure 4.11(b) —



Schedule: $CAABABBABBB$

(a)

Some Possible Constraint Sets

Singletons	$\{A[1]\}, \{A[2]\}, \ldots, \{A[12]\}$
Actor A writes to $A \rightarrow B$ from a loop	$\{A[1-3]\}$, $\{A[4-6]\}$, $\{A[7-9]\}$, $\{A[10-12]\}$
Encapsulate A_1, A_2 in a schedule loop	$\{A[1-6]\}$
Encapsulate B_5, B_6 in a schedule loop	$\{A[8-11]\}$
Actor B reads $A \rightarrow B$ from a loop	$\{A[12], A[1]\}$, $\{A[2], A[3]\}$, $\{A[4], A[5]\}$, $\{A[6], A[7]\}$, $\{A[8], A[9]\}$, $\{A[10], A[11]\}$

(b)

Figure 4.11. This example illustrates how superimposing different constraint sets can lead to different buffer periods. The figure depicts an SDF graph, a schedule for the graph and five possible mechanisms for imposing contiguity constraints on the edge $A \rightarrow B$.

$\{A[1-3]\}, \{A[4-6]\}, \{A[7-9]\}, \{A[10-12]\}$. If we add the additional condition that the first two invocations of A are grouped into a schedule loop (we change the schedule to $C(2A)BABBABBB$), then we must consider another constraint set $\{A[1-6]\}$. The new buffer periods are the combination of the 17 constraint sets in the first three rows of Figure 4.11(b) — $\{A[1-6]\}, \{A[7-9]\}, \{A[10-12]\}$. Now if we encapsulate B_5 and B_6 within a schedule loop (the new schedule is $C(2A)BABBAB(2B)$), the resulting constraint set is $\{B[9-12]\}$, which is equivalent to $\{A[8-11]\}$, due to the unit delay. This new constraint forces us to merge buffer periods $\{A[7-9]\}$ and $\{A[10-12]\}$, and the resulting buffer periods are $\{A[1-6]\}$ and $\{A[7-12]\}$. Finally, if we impose the condition that B reads $A \rightarrow B$ through an intra-actor loop, then we have the six additional constraint sets shown in the fifth row of Figure 4.11(b). The first of these constraint sets intersects both of the remaining buffer periods and we are left with a single buffer period $\{A[1-12]\}$.

4.4.3 Contiguity Constraints for Dynamic Buffers

Dynamic buffering imposes contiguity constraints between buffer accesses whenever a read occurs when the number of tokens on an edge α exceeds $total\_consumed(\alpha)$. In such situations, the token to be read co-exists with the corresponding token of the next schedule period — so we cannot dedicate a single memory location to that token. For a given edge, an efficient way to deal with such cases is to force all of these accesses to occur in the same contiguous block ς of memory. Since the location of each of these accesses varies between schedule peri-

ods, the accesses are performed through read/write pointers. Any read that occurs when the token population is within $total\_consumed(\alpha)$, however, corresponds to a token whose location is independent of ς. To explain this effect precisely, we introduce the following definition.

Definition 4.3: Let G be an SDF graph and suppose that α is an edge in G. Then a **transaction** on α is an ordered pair (i, j), such that $1 \le i, j \le total\_consumed(\alpha)$ and[1]

$$j = ((i - 1 + delay(\alpha)) \bmod total\_consumed(\alpha)) + 1 \quad .$$

Thus, (i, j) is a transaction on α if the jth token consumed by $sink(\alpha)$ in any given schedule period is the ith token produced by $source(\alpha)$ in that schedule period or some earlier schedule period. For a given periodic looped schedule S for G, we say that the transaction (i, j) is a **static transaction** if the number of tokens existing on α just prior to the jth read access by $sink(\alpha)$ of α is less than or equal to $total\_consumed(\alpha)$. We can express this condition as

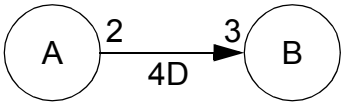
$$delay(\alpha) + produced(\alpha)N_A - consumed(\alpha)(N_B - 1) - (j - 1) \bmod consumed(\alpha) \le total\_consumed(\alpha) \quad ,$$

where $N_B = 1 + \lfloor (j - 1) / (consumed(\alpha)) \rfloor$ is the invocation of $sink(\alpha)$ during which the jth read access of α occurs, and N_A is number of invocations of $source(\alpha)$ that precede the N_Bth invocation of $sink(\alpha)$ in S. We say that a transaction is a **dynamic transaction** if it is not a static transaction.

The transactions on an edge can be determined easily from the acyclic pre-

1. The $+1$ and -1 are required in this expression because we (by convention) number tokens starting at 1 rather than 0.

cedence graph, and the static and dynamic transactions can be identified by simulating the activity on the edge over one schedule period. Figure 4.12 illustrates the decomposition of a buffer based on static and dynamic transactions. Here, the repetitions vector is given by $\mathbf{q}(A, B) = (3, 2)^T$, and thus $total\_consumed(A \rightarrow B) = 6$, Now, it is easily verified that for the given schedule, the maximum number of tokens that coexist on $A \rightarrow B$ is 8 — so clearly dynamic buffering applies. However, from the lower table in Figure 4.12(a), we see that the third, fifth, and sixth read accesses of B occur when there are $total\_consumed(\alpha)$ or fewer tokens queued on $A \rightarrow B$. This corresponds to the set of static transactions, which is summarized in the table labeled *transactions* in Figure 4.12(a). Thus tokens associated with transactions $(1, 5)$, $(2, 6)$ and $(5, 3)$ can be buffered in independent memory locations, while $(3, 1)$, $(4, 2)$ and $(6, 4)$ must be maintained in contiguous memory. The resulting constraint sets are $\{A[1]\}$, $\{A[2]\}$, $\{A[5]\}$, and $\{A[3], A[4], A[6]\}$. Figure 4.12(b) illustrates the use of these constraint sets to form independent buffering units. Here, $A[1]$, $A[2]$ and $A[5]$ are mapped to independent (not necessarily contiguous) memory locations L1, L2, and L3 respectively, and the remaining constraint set is mapped to a contiguous five-word block of storage, labeled the "dynamic buffer component". Five words are required because this is the maximum number of coexisting tokens from $\{A[3], A[4], A[6]\}$. Figure 4.12(b) shows how the profile of live tokens in this buffering arrangement changes through the first schedule period. Each live token is represented by an ordered pair i, j, which denotes the jth token to be consumed by actor B in schedule period i, and a shaded region designates the absence of a token. Observe that for each live token



Schedule: AABAB

(a)

transactions

(1, 5) <static>	(4, 2) <dynamic>
(2, 6) <static>	(5, 3) <static>
(3, 1) <dynamic>	(6, 4) <dynamic>

read access	number of tokens on A→B just prior to the access
B[1]	8
B[2]	7
B[3]	6
B[4]	7
B[5]	6
B[6]	5

(b)

	L1	L2	L3	Dynamic Buffer Component				
initially	1,3			1,1	1,2	1,4		
after A_1	1,3	1,5	1,6	1,1	1,2	1,4		
after A_2	1,3	1,5	1,6	1,1	1,2	1,4	2,1	2,2
after B_1		1,5	1,6			1,4	2,1	2,2
after A_3	2,3	1,5	1,6	2,4		1,4	2,1	2,2
after B_2	2,3			2,4			2,1	2,2

Figure 4.12. An illustration of static and dynamic transactions for a dynamic buffer. In (b), i, j represents the live token that is to be the jth token consumed by actor B in schedule period i.

s in the dynamic buffer component, there is some point in the schedule period when s coexists with the corresponding token of the next or previous schedule period. This is precisely why these tokens must be buffered as a contiguous unit. Observe also that in the dynamic buffer component, the read and write pointers for B and A, respectively, each shift three positions to the right (in a modulo-5 sense) every schedule period. These pointers are not involved in accesses of L1, L2 and L3 — these locations can be accessed using absolute addressing.

For the example in Figure 4.12, mapping all accesses of $A \rightarrow B$ to a single contiguous segment ς of memory requires an 8-word block of memory, while decomposing this buffer based on static and dynamic transactions allows a partition into four mutually independent blocks of 1, 1, 1 and 5 words. Although the net requirement of physical memory is the same, there is less potential for fragmentation, or equivalently, more opportunity for buffer reuse [Fabr82] when this example is a subsystem in a larger graph. Furthermore, the lifetime of ς extends throughout the entire schedule period, whereas L2 and L3 are live only in the interval between invocations A_1 and B_2. These two locations may thus be reused for other parts of the graph.

It is not obvious, however, that decomposing a buffer based on static and dynamic transactions will never increase the net memory requirements. If we refer to the tokens associated with static transactions and dynamic transactions as *static tokens* and *dynamic tokens* respectively, then the transaction-based decomposition requires a set of memory blocks whose sizes total $N_s + N_d$ words, where N_s is the number of static tokens (in a single schedule period) and N_d is the maximum number of coexisting dynamic tokens. If this sum exceeds the maximum number of

coexisting tokens on the edge, then without further analysis — for which currently there are no general techniques — we cannot guarantee that decomposing the buffer will not be detrimental. Fortunately, however, $(N_s + N_d)$ is always equal to the undecomposed dynamic buffer size, as the following theorem suggests.

Theorem 4.2: Suppose that G is a consistent, connected SDF graph, S is a minimal, valid schedule for G, and α is an edge in G. Suppose also that the maximum number of coexisting tokens $M(\alpha)$ on α during an execution of S exceeds $total\_consumed(\alpha)$. Then $N_s + N_d = M(\alpha)$, where N_s is the number of static tokens and N_d is the maximum number of coexisting dynamic tokens.

Proof: Suppose that at some time τ in the schedule period there are R live tokens on α, and first suppose that $R \geq total\_consumed(\alpha)$. Since the tokens buffered on an edge are successive, the last $total\_consumed(\alpha)$ tokens produced by $source(\alpha)$ are live at time τ. Thus, there is a token corresponding to each static transaction on the edge. It follows that there are $R - N_s$ dynamic tokens on α at time τ.

Now suppose that $R < total\_consumed(\alpha)$. We consider two cases here:

Case 1 — $(R < total\_consumed(\alpha))$ **and** $(N_s < total\_consumed(\alpha) - R)$. Then,

$$(\text{The number of dynamic tokens at time } \tau) \leq R < total\_consumed(\alpha) - N_s < M(\alpha) - N_s .$$

Case 2 — $(R < total\_consumed(\alpha))$ **and** $(N_s \geq total\_consumed(\alpha) - R)$. Then,

(The number of dynamic tokens at time τ) $\leq$
$R - (N_s - (total\_consumed(\alpha) - R))$

$= total\_consumed(\alpha) - N_s < M(\alpha) - N_s$.

From the above discussion — for both $(R \geq total\_consumed(\alpha))$ and $(R < total\_consumed(\alpha))$ — the number of dynamic tokens when $R = M(\alpha)$ is $(M(\alpha) - N_s)$, and this amount of dynamic tokens cannot be exceeded with any other value of R. Therefore, $N_d = M(\alpha) - N_s$, which is equivalent to the desired result. *QED*.

We conclude this section by pointing out that it is possible to decompose the dynamic buffer component further — each dynamic transaction can be mapped to an independent block of memory. For example, the dynamic buffer component in Figure 4.12 can be separated into three two-word fragments corresponding to transactions $(3, 1)$, $(4, 2)$ and $(6, 4)$. This could be achieved simply by using different read and write pointers for each of the associated accesses — we would need three separate write pointers for $A[3]$, $A[4]$ and $A[6]$, and three separate read pointers for $B[1]$, $B[2]$ and $B[4]$. The overhead associated with this scheme is significant, but difficult to gauge precisely. First, it places more pressure on the address-register allocator and may increase the amount of spilling. This, in turn requires an extra memory location to save each spilled item. Finally, the sum of the independent dynamic transaction segments (in this case $2 + 2 + 2 = 6$) may exceed the maximum number of coexisting dynamic tokens (in this case 5). Thus, for small to moderate dynamic buffer sizes it is unlikely that decomposing the dynamic buffer component further will be of value. However, when large delays are involved, it may provide substantial new opportunities for overlaying.

For example, for the SDF graph and schedule in Figure 4.13, there are no static transactions for the edge $B \rightarrow C$, and a 100-word block of memory is required for this edge if we do not decompose the dynamic buffer component. However, if we view each of the four dynamic transactions $(1, 1)$, $(2, 2)$, $(3, 3)$ and $(4, 4)$ as a separate unit, we can implement $B \rightarrow C$ with four independent 25-word blocks of memory. This additional freedom may lead to much better overall memory use if this example is used as a subsystem in a more complex graph.

4.5 Eliminating Modulo Address Computations

In Subsections 4.1.2 and 4.2.1, we discussed the use of circular buffers to decrease memory requirements and to implement edges that have delay, and in Subsection 4.2.3, we discussed the overhead associated with accessing circular buffers, which ranges from zero to a few instructions for processors that have hardware support for circular buffering, such as the Motorola DSP56000, to several instructions for processors that do not have hardware support, as with general purpose microprocessors such as the MIPS R3000. In this section, we develop a sys-

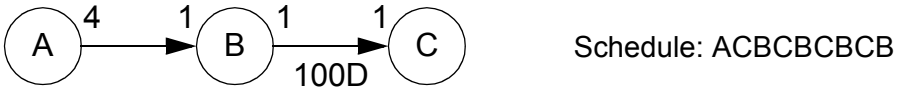


Figure 4.13. An example that illustrates the benefits of decomposing the dynamic buffer component into a separate segment for each dynamic transaction.

tematic approach to eliminating modulo accesses.

4.5.1 Determining Which Accesses Wrap Around

First, we show how to efficiently determine which accesses of a circular buffer wrap around the end of the buffer. For a static circular buffer, this is straightforward — if α denotes the edge in question and J denotes the blocking factor, we simply determine the values of $n \in \{0, 1, \ldots, J \times total\_consumed(\alpha) - 1\}$ for which

$$\rho_0 + n = (\text{some positive integer}) \times \text{BUFSIZE} ,$$

where BUFSIZE denotes the length (number of words) of the circular buffer, and ρ_0 denotes the buffer position of the initial access — that is, $\rho_0 = delay(\alpha)$ if we are concerned with the accesses of $source(\alpha)$ and $\rho_0 = 0$ if we are concerned with $sink(\alpha)$.

For dynamic buffers, different accesses will wrap around the end of the buffer in different schedule periods. However, there may still exist invocations whose accesses do not wrap around in any schedule period. To determine these invocations we need to use two simple facts of modulo arithmetic.

Fact 4.1: Suppose that a, b and c are positive integers, and suppose that a divides both b and c. Then for some nonnegative integer k, $(b \bmod c) = ka$.

Proof: By definition,

$$(b \bmod c) = b - \left(\left\lfloor \frac{b}{c} \right\rfloor \times c \right) . \qquad (4\text{-}2)$$

Both the subtrahend and minuend of the left hand side of (4-2) are divisible by a,

so a must divide $(b \bmod c)$. *QED*.

Fact 4.2: Suppose that p and q are coprime positive integers, let I_q denote the set $\{0, 1, \ldots, (q-1)\}$, and suppose that $r \in I_q$. Then for all $k_1 \in I_q$ there exists $k_2 \in I_q$ such that $(r + pk_2) \bmod q = k_1$.

Proof: (By contraposition). Suppose that for some $k_1 \in I_q$, there is no $k_2 \in I_q$ such that $(r + pk_2) \bmod q = k_1$. Then $((r + px) \bmod q)$ takes on at most $(q-1)$ distinct values as x varies across I_q. Thus, there exist distinct $k_{2a}, k_{2b} \in I_q$ such that

$$(r + k_{2a}p) \bmod q = (r + k_{2b}p) \bmod q = k \text{ , for some } k \in I_q, \tag{4-3}$$

which implies that there exist distinct nonnegative integers r_a and r_b such that

$$(r + k_{2a}p) = (r_a q + k) \text{ , and } (r + k_{2b}p) = (r_b q + k) \text{ ,} \tag{4-4}$$

and thus,

$$(k_{2a} - k_{2b})p = (r_a - r_b)q \text{ .} \tag{4-5}$$

Now since $k_{2a}, k_{2b} \in \{0, 1, \ldots, (q-1)\}$, it follows from (4-5) that p and q are not coprime. Thus, the original assumption that p and q are coprime cannot hold. *QED*.

Applying Fact 4.1 with

$$a = gcd(\{J \times total\_consumed(\alpha), \text{BUFSIZE}\}) \text{ ,}$$

$b = k_1 \times J \times total\_consumed(\alpha)$, and

$c = \text{BUFSIZE}$,

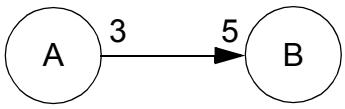
we see that for each positive integer k_1 , there is a nonnegative integer k_2 such that

$$\begin{aligned}(k_1 \times J \times total\_consumed(\alpha)) \bmod \text{BUFSIZE} \\ = k_2 gcd(\{J \times total\_consumed(\alpha), \text{BUFSIZE}\})\end{aligned} \tag{4-6}$$

This means that we can consider each dynamic buffer as consisting of successive "windows" of size $gcd(\{J \times total\_consumed(\alpha), \text{BUFSIZE}\})$. In some schedule period, if $source(\alpha)$ or $sink(\alpha)$ performs its i th access at offset j of window w_x , then since the i th access shifts $(J \times total\_consumed(\alpha))$ positions from schedule period to schedule period, we know that the i th access in any schedule period will occur at offset j of some window. For example, for the dynamic buffer in Figure 4.14, it is easy to verify that for all odd schedule periods, the window offset for the first access of A is 0 .

Now let w_s denote $gcd(\{J \times total\_consumed(\alpha), \text{BUFSIZE}\})$, the size of each window. Also, let $n_w = \text{BUFSIZE} / w_s$, the number of windows. Suppose that in the first schedule period, access i occurs at offset j of window w (assume now that windows and offsets are numbered starting at 0). Then the window number of the i th access in some later schedule period k can be expressed as $((w + ((k \times J \times total\_consumed(\alpha)) / w_s)) \bmod n_w)$. This is simply the initial window number plus the number of windows traversed modulo the number of windows. To this expression, we can apply Fact 4.2 with

$$p = \frac{J \times total\_consumed(\alpha)}{w_s} = \frac{J \times total\_consumed(\alpha)}{gcd(\{J \times total\_consumed(\alpha), \text{BUFSIZE}\})} ,$$



Schedule: AABAABAB

$total\_consumed(A \rightarrow B) = 15$

BUFSIZE = 10

$J = 1$

$gcd(\{J \times total\_consumed(A \rightarrow B), \text{BUFSIZE}\}) = 5$ ("window" size)

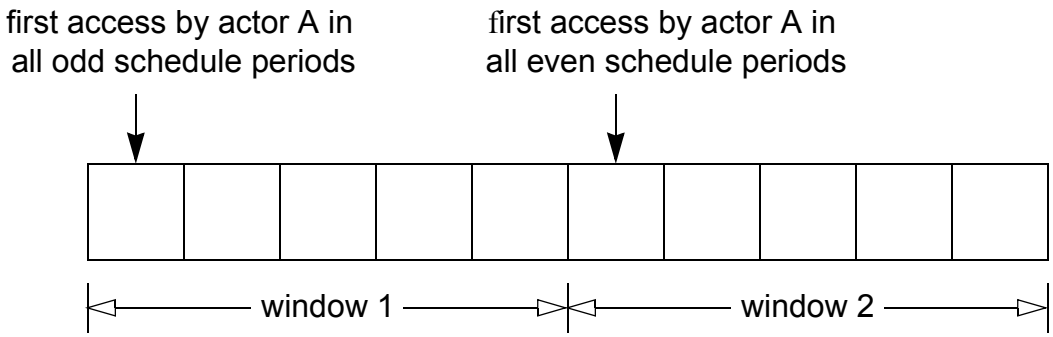


Figure 4.14. An illustration of repetitive access patterns in $gcd(\{J \times total\_consumed(\alpha), \text{BUFSIZE}\})$ -word windows within a buffer.

$q = n_w = \dfrac{\text{BUFSIZE}}{gcd(\{J \times total\_consumed(\alpha), \text{BUFSIZE}\})}$, and $r = w$. Interpreting this result, we see that for each window w', there will be schedule periods (values of k) in which the jth access occurs in w'. Thus, the jth access of some schedule period will wrap around the end of the buffer if and only if the jth access of the first schedule period occurs at the end of a window.

We have proved the following theorem.

Theorem 4.3: Suppose that α is an edge in a connected, consistent SDF graph; suppose $A \in (\{source(\alpha)\} \cup \{sink(\alpha)\})$; and define $\rho_0 = delay(\alpha)$ if $A = source(\alpha)$, and $\rho_0 = 0$ if $A = sink(\alpha)$. Then for $j \in \{1, 2, \ldots, J \times total\_consumed(\alpha)\}$, the jth access of α by A wraps around the end of the buffer if and only if

$$(\rho_0 + (j - 1)) \bmod w_s = w_s - 1 \ ,$$

where $w_s = gcd(\{J \times total\_consumed(\alpha), \text{BUFSIZE}\})$.

The check of Theorem 4.3 can be further simplified by observing the periodicity of the modulo term — we need only determine the first access that wraps around, which we denote by j_w, explicitly:

$$j_w = w_s - (\rho_0 \bmod w_s) \ . \tag{4-7}$$

Then, we immediately obtain the complete set S_w of accesses that wrap around by

$$S_w = S_w(\alpha, \text{BUFSIZE}, J) = \tag{4-8}$$

$$\left\{ j_w + n \times w_s \middle| \left(n \in \left\{ 0, 1, \ldots, \middle| \frac{(J \times total\_consumed(\alpha)) - 1}{w} \middle| \right\} \right) \right\}$$

For the example of Figure 4.14, we have $j_w = 5$, and $S_w = \{5, 10, 15\}$. Code to implement these accesses must perform modulo address computations. These modulo computations will correspond to accesses that wrap around only one-third of the time. However, unless, we increase the blocking factor, we must ensure that these accesses are always performed with modulo updates. In general, modulo computations will wrap around one out of every $n_w = \frac{\text{BUFSIZE}}{gcd(\{J \times total\_consumed(\alpha), \text{BUFSIZE}\})}$ times.

We can reduce the average rate at which modulo computations must be performed by a factor of n_w if we increase the blocking factor to n_w. Assuming that all invocations of the same actor require the same amount of time to execute[1], the rate at which modulo computations must be performed is proportional to $R_M \equiv \frac{|S_w|}{J}$, where $|S_w|$ denotes the number of members in the set S_w. The denominator term J is required because the amount of execution time required for a schedule period (an iteration of the target program's outermost loop) is proportional to the blocking factor. For example, in Figure 4.14, $J = 1$, $S_w = \{5, 10, 15\}$, $|S_w| = 3$, and $R_M = 3$. If we increase the blocking factor to 2 and retain the same buffer size, $S_w = \{10, 20, 30\}$, $|S_w| = 3$, and $R_M = 1.5$ — thus the frequency of required modulo address computations decreases by a

1. In general, this assumption does not hold; in such cases our analysis is not exact, but it gives a useful estimate.

factor of 2.

Observe that the number of modulo computations required also depends on the choice of the buffer size. Clearly, one out of $gcd(\{J \times total\_consumed(\alpha), \text{BUFSIZE}\})$ accesses requires a modulo computation. Thus the modulo overhead varies (neglecting looping considerations, which will be discussed in Subsection 4.5.2) inversely with $gcd(\{J \times total\_consumed(\alpha), \text{BUFSIZE}\})$. For example in Figure 4.14, a 7-word buffer can support the given schedule. However, this requires $15 / gcd(\{15, 7\}) = 15$ modulo computations per minimal schedule period: every access must perform a modulo update! Increasing the buffer size to 10 results in 5 times fewer modulo computations. Thus, for frequently executed sections of code, it may be beneficial to explore tolerable increases in buffer size for the possible reduction of modulo updates.

4.5.2 Handling Loops

In the absence of schedule loops and loops within the actor code blocks, the number of modulo computations required in the target code is exactly $|S_w|$. However, a loop may cause the same physical instructions to perform both wrap-around accesses and linear accesses. In such cases, we must either unroll the loop to isolate the accesses that wrap around, or we must perform a modulo address computation for every access that is executed from within the loop. Here we assume that the loop structure is fixed: we focus on analyzing the loop structure to eliminate modulo accesses while leaving the loops intact.

To eliminate unnecessary modulo address computation for the read or write accesses performed by some actor A from/to an edge α, we first identify the set of

distinct physical instruction sequences, called **buffer access instruction sequences**, that will be used to access α by A. This concept is similar to common code space sets, which associate blocks of program memory with actor invocations. However, the buffer access instruction sequences depend on intra-actor loops as well as schedule loops.

For a given buffer access instruction sequence, the corresponding machine instruction(s) must perform a modulo address computation if and only if the associated set of buffer accesses I_a intersects the set of wrap-around accesses — that is, if and only if $(I_a \cap S_w) \neq \varnothing$. In practice, however, we do not need to explicitly compute and maintain S_w nor the access sets associated with each buffer instruction sequence. We simply simulate the buffer activity, traversing the buffer access instruction sequences in succession, for one schedule period and apply Theorem 4.3 for each access. If Φ denotes the current buffer access instruction sequence in our simulation, and the current access is the jth access of edge α by actor A, then we mark Φ as requiring a modulo computation if

$$(\rho_0 + (j-1)) \bmod gcd(\{J \times total\_consumed(\alpha), \text{BUFSIZE}\}) \quad .$$

$$= (gcd(\{J \times total\_consumed(\alpha), \text{BUFSIZE}\}) - 1) \quad .$$

All buffer access instruction sequences that are not marked by this simulation can be translated into simple linear address updates.

4.6 Summary

We have presented a classification of buffers based on whether they are

static or dynamic, linear or modulo, and contiguous or scattered; we have evaluated the impact of these choices on storage requirements; and we have suggested guidelines for choosing between them. More thorough and systematic techniques to determine an optimal combination of buffering parameters is an important and challenging area for further study.

In Section 4.3, we introduced dataflow analysis techniques to minimize the spilling of address registers under static buffering. How useful and effective these techniques are depend both on the number of available registers and on how expensive a spill to memory is. For example, in the Motorola DSP56000, eight registers are available for addressing, while spills can often be performed with no run-time overhead (by doing them in parallel with other operations [Powe92]). In contrast, in the MIPS R3000, any of the available 32 registers can be used for addressing, and at least one instruction cycle is required for a spill. Being able to accurately and efficiently estimate the effects of spilling would be useful in deciding between static and dynamic buffering.

In Section 4.4, we developed lifetime analysis techniques that aid in reducing storage requirements for buffers. An important area for further investigation is the incorporation of addressing trade-offs between contiguous and scattered buffering. For example, if a logical buffer of length n is assigned to n mutually non-contiguous memory locations, then in general n absolute addresses must be employed. For programmable DSPs such as the DSP56000, arbitrary absolute addresses require an additional word of program memory and an additional instruction cycle, while register-indirect accesses to a contiguous buffer involve no program memory overhead and can often be performed in parallel with other useful operations [Powe92]. In contrast, many general purpose microprocessors allow large absolute displacements to be accessed through single-word instructions, but

they do not allow register-indirect accesses to issue in parallel with other instructions. Furthermore, many do not support hardware autoincrement — a separate instruction must be issued to update the buffer pointer. Thus, more aggressive scattering of buffers may favor such general purpose processors, while there is a strong trade-off between buffer storage, address storage, and execution time in the DSP56000 and most other digital signal processors.

Also a scattered buffer can consist of multiple contiguous blocks of memory, each of which is accessed through a separate buffer pointer. Managing these multiple buffer pointers introduces another machine-dependent trade-off. Further examining the machine-dependent aspects of contiguous vs. scattered buffering is an important direction for future work.

Finally, we presented techniques to reduce modulo addressing overhead for both static and dynamic buffers. These techniques apply whenever modulo buffers are used, but how much improvement is gained depends on how expensive a modulo address update is in the target processor.

5

FURTHER WORK

This thesis has presented a formal theory for constructing and manipulating loops from SDF representations of digital signal processing algorithms, and based on this theory, techniques have been presented for compiling SDF programs into efficient code for programmable processors. The techniques have focused on the minimization of code size, the minimization of the buffer memory requirement, and the efficiency of buffering. We have defined a class of code-size-minimizing schedules called *single appearance schedules*. The central contribution of this thesis is a uniprocessor scheduling framework that constructs single appearance schedules whenever they exist, and when single appearance schedules do not exist, guarantees optimal code size for all actors that are not contained in a certain type of subgraph called a *tightly interdependent subgraph*.

This scheduling framework has been implemented in Ptolemy, a design environment for simulation, prototyping, and software synthesis of heterogeneous systems [Buck92]. A large part of the implementation in Ptolemy was performed by Joseph Buck, a graduate student colleague at the time and now with Synopsys Inc., and Soonhoi Ha, a post-doctoral fellow of U.C. Berkeley at the time and now a lecturer at Seoul National University. The implementation has been tested on

several practical examples, such as the digital audio tape to compact disc sample rate conversion system of Figure 1.1, developed by Thomas M. Parks, a fellow graduate student at U. C. Berkeley; and a QMF filter bank that was developed by Alan Peevers, who is now at Emu/Creative Systems. Our scheduling framework has constructed optimally compact schedules for all of these examples. An example of particular interest is a rake receiver for spread spectrum communications, developed by Sam Sheng, a fellow graduate student at U. C. Berkeley. For this example, in the C code generation domain of Ptolemy, our scheduling framework generated a code file whose size was under 35 kilobytes, while Buck's loop scheduler [Buck93], discussed in Subsection 3.4.2, generated a 1.3 megabyte code file. Although, the fast heuristics on which Buck's scheduler is based often succeed in constructing very compact schedules, in this particular instance, the more thorough techniques developed in this thesis outperformed Buck's scheduler by more than a factor of 37.

In this remainder of this section, we discuss a number of problems that remain open in the area of compiling SDF graphs.

5.1 Tightly Interdependent Graphs

Loose interdependence algorithms guarantee optimal code size for each actor that does not lie in a tightly interdependent subgraph, and they guarantee that the number of appearances of each actor within a tightly interdependent subgraph is determined entirely by the tight scheduling algorithm, however, this thesis does not propose any techniques that give guarantees on how compactly a tightly interdependent component will be scheduled. Thus, to provide a more complete solution to the problem of generating compact code, it would be useful to study

techniques for scheduling tightly interdependent graphs compactly. Once developed, such techniques can be incorporated in the tight scheduling algorithm without affecting the performance of the other three component algorithms.

One direction of study in the problem of compactly scheduling tightly interdependent graphs is the application of *retiming*. Retiming was proposed by Leiserson et al. [Leis83] as a technique for minimizing the clock period of synchronous digital circuits. Extension of the retiming concept to general SDF graphs was discussed by Lee in [Lee86]; and in [Zivo93], Zivojnovic et al. formally analyze properties of retimed SDF graphs and they formulate an integer linear programming solution to the problem of minimizing the total delay count of an SDF graph through retiming.

In an SDF graph, retiming can be viewed as rearranging the delays in accordance with certain constraints. As a simple example of retiming, and how it can improve the scheduling of tightly interdependent subgraphs, consider the example of Figure 5.1. Figure 5.1(a) shows a tightly interdependent SDF graph, and Figure 5.1(b) shows how moving the delay on the edge $A \rightarrow B$ to the edge $B \rightarrow A$ results in a graph that has a single appearance schedule. This application of

Figure 5.1. An example of how retiming can lead to more compact schedules of SDF graphs.

retiming can be implemented by firing B once as a *preamble* to the periodic schedule. However, the code to construct this preamble will negate the advantage of having a more compact periodic schedule. Alternatively, since the transformation in computation may lead simply to a transient that diminishes with time, it may be valid to ignore the preamble and directly implement a periodic schedule for the retimed graph. In such cases, applications of retiming such as the example of Figure 5.1 can improve code size compactness for tightly interdependent graphs. Thus, it would be useful if we could efficiently determine when a tightly interdependent SDF graph can be retimed into an SDF graph that has a single appearance schedule, and if we could determine appropriate retimings for such cases.

5.2 Buffering

In Section 4.6, we discussed some directions for further study to develop systematic methods to choose optimally between static vs. dynamic, linear vs. modulo, and contiguous vs. scattered buffers. Also, more powerful techniques are desirable for minimizing the buffer memory requirement of a schedule. We have presented a technique to construct the single appearance schedule that minimizes the buffer memory requirement for a chain-structured SDF graph. Techniques to address this problem for general acyclic graphs would be useful for incorporation into the acyclic scheduling algorithm. Similarly, a technique for constructing sub-independent partitions that leads to minimum buffer memory requirement is desirable.

Systematic assessment of scheduling trade-offs between the code size and the buffer memory requirement is another area for further study. For example, if a single appearance schedule has been constructed, and the resulting code does not

fully occupy the available program memory, then we would like to know how the remaining program memory can be utilized to expand the schedule in such a way that the buffer memory requirement is minimized. Alternatively, one can attempt to develop a scheduling algorithm that is not restricted to single appearance schedules and attempts to jointly minimize the code size and buffer memory requirement. A further step in this scheduling problem is incorporating considerations that relate to buffer overlaying.

5.3 Parallel Computation

A number of scheduling techniques have been developed for compiling SDF graphs into efficient code for multiprocessor systems, for example [Sih91, Prin91, Liao93]. However these techniques do not consider code size constraints. Thus, it would be useful to extend the loose interdependence scheduling framework to address parallelism as well as memory requirements. An interesting problem that arises in this domain is the construction of optimal single appearance *parallel* schedules.

One restricted class of single appearance parallel schedules that would be useful to consider is that in which each schedule loop is either a serial loop, whose iterations are to be executed in succession as in the uniprocessor scheduling case, or a **doall** loop — a loop in which all iterations can execute simultaneously without any synchronization between them [Zima90]. Given a schedule loop in this model, it is executed serially if for some invocation I of the loop, data produced by some iteration of I is consumed by another iteration of I. If all iterations of a given invocation of a schedule loop are independent, then all iterations are executed in parallel. The execution "mode" of each loop can be represented by

appending a letter s to the iteration count if it is to be executed serially and a letter p if it is to be executed as a doall loop. Thus, for example the schedule $(2p(2sA)(4pB))$ corresponds to the processor-time execution profile in Figure 5.2.

The basic problem to address in constructing this type of single appearance parallel schedule is determining the schedule that maximizes the throughput. Unlike the problem of minimizing code size, the solution to an instance of this problem depends in general on the execution time of each actor invocation, and thus a solution cannot be obtained from a topological analysis alone. This complication applies even to homogeneous SDF graphs. As a simple example, consider the homogeneous graph in Figure 5.3, and consider the single appearance parallel schedules $S_1 \equiv (6s(3s(5pA))(5s(3pB)))(90pC)$,

$S_2 \equiv (5s(4s(5pA))(10s(2pB)))(100pC)$, and

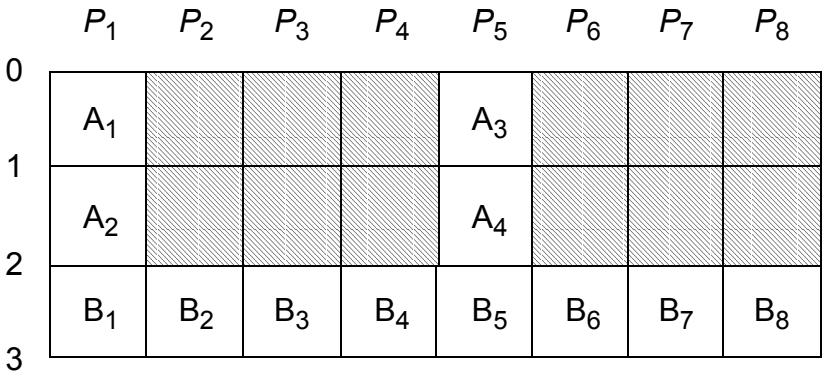


Figure 5.2. The processor-time execution profile for the single appearance parallel schedule $(2p(2sA)(4pB))$. The vertical axis corresponds to time and the horizontal co-ordinate identifies one of eight available processors $P_1 - P_8$. It is assumed that each actor invocation takes one time unit. A shaded region indicates that no operation is performed.

$S_3 \equiv (8s(3s(4pA))(4s(3pB)))(96pC)$.

If we denote the execution times of actors A, B and C by t_A, t_B, t_C, respectively, then we can measure the throughput of S_1 as $throughput(S_1) = 90/(6(3t_A + 5t_B) + t_C)$ minimal schedule periods per unit time. Similar expressions can easily be derived for the throughput of S_2 and S_3. The table below lists the throughput of each schedule for three different sets of execution times.

t_A	t_B	t_C	$throughput(S_1)$	$throughput(S_2)$	$throughput(S_3)$
500	150	1	0.00667	0.00571	0.00571
50	1	950	0.0479	0.0500	0.0440
1	950	950	0.00305	0.00206	0.00306

In this table, we see that each set of execution times corresponds to a different throughput-minimizing schedule from among the three schedules considered. Thus, we see that even for homogeneous SDF graphs, the construction of optimal

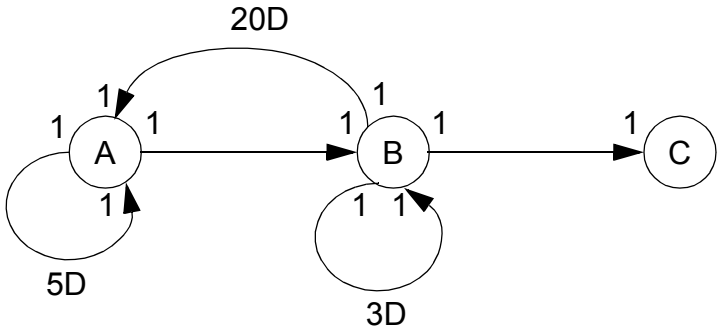


Figure 5.3. An example used to illustrate the problem of constructing single appearance parallel schedules.

single appearance parallel schedules (with regards to throughput) cannot be based solely on topological considerations.

When constructing single appearance parallel schedules or more general looped schedules for parallel computation, it would be useful to consider the time required for interprocessor communication. Scheduling techniques for SDF graphs that take interprocessor communication into account have been developed by Liao et al. [Liao93] and Sih [Sih91]; however these techniques do not attempt to construct loops in the target code.

REFERENCES

[AbuS81]

W. A. Abu-Sufah, D. J. Kuck, and D. H. Lawrie, "On the Performance Enhancement of Paging Systems Through Program Analysis and Transformations," *IEEE Transactions on Computers*, **Vol. C-30, No. 5**, May, 1981.

[Acke82]

W. B. Ackerman, "Data Flow Languages," *IEEE Computer Magazine*, **Vol. 15, No. 2**, February, 1982.

[Ade94]

M. Ade, R. Lauwereins, and J. A. Peperstraete, "Buffer Memory Requirements in DSP Applications," presented at *IEEE Workshop on Rapid System Prototyping*, Grenoble, June, 1994.

[Aho88]

A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers Principles, Techniques, and Tools*, Addison-Wesley, 1988.

[Alle87]

R. Allen and K. Kennedy, "Automatic Transformation of Fortran Programs to Vector Form," *ACM Transactions on Programming Languages and Systems,* **Vol**. **9**, **No. 4**, October, 1987

[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.

[Arvi90]

Arvind and R. S. Nikhil, "Executing a Program on the MIT Tagged-Token

Dataflow Architecture," *IEEE Transactions on Computers*, **Vol. C-39**, **No. 3**, March, 1990.

[Arvi91]

Arvind, L. Bic, and T. Ungerer, "Evolution of Dataflow Computers," *Advanced Topics in Dataflow Computing*, Prentice-Hall, 1991.

[Ashc75]

E. A. Ashcroft, "Proving assertions about Parallel Programs," *Journal of Computer and Systems Science*, **Vol. 10**, **No. 1**, 1975.

[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.

[Bhat92]

S. S. Bhattacharyya and E. A. Lee, *Memory Management for Synchronous Dataflow Programs*, Memorandum No. UCB/ERL M92/128, Electronics Research Laboratory, University of California at Berkeley, November, 1992.

[Bhat93]

S. S. Bhattacharyya and Edward A. Lee, "Scheduling Synchronous Data-flow Graphs for Efficient Looping," *Journal of VLSI Signal Processing*, **No. 6**, 1993.

[Bier93]

J. C. Bier, P. D. Lapsley, and E. A. Lee, *Design Tools and Methodologies*

for DSP Systems — Volume 1: DSP Design Challenges, Methodologies, and Tools, Berkeley Design Technologies, Inc., Fremont, California, 1993.

[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.

[Buck92]

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*, January, 1994.

[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.

[Carl91]

M.C. Carlisle and E. L. Lloyd, "On the k-coloring of Intervals," *Advances in Computing and Information — ICCI 1991*, Ottowa, Canada, Lecture Note 497 — Springer Verlag, May, 1991.

[Chao93]

L-F. Chao and E. H-M. Sha, *Static Scheduling for Synthesis of DSP Algorithms on Various Models*, technical report, Department of Computer Science, Princeton University, 1993.

[Chas84]

M. Chase, "A Pipelined Dataflow Architecture for Signal Processing: the NEC μPD7281," *VLSI Signal Processing*, IEEE Press, 1988.

[Chow88]

F. C. Chow, “Minimizing Register Usage Penalty at Procedure Calls,” *SIGPLAN Notices,* **Vol. 23**, **No. 7**, 1988.

[Corm90]

T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, McGraw-Hill, 1990.

[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.

[Davi89]

J. W. Davidson and D. B. Whalley, *Methods for Saving and Restoring Register Values Across Function Calls*, Technical Report 89-11, Department of Computer Science, University of Virginia, 1989.

[Davi92]

J. W. Davidson and A. M. Holler, “Subprogram Inlining: A Study of its Effects on Program, Execution Time,” *IEEE Transactions on Software Engineering*, **Vol. 18, No. 2,** February, 1992.

[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.

[Denn92]

J. B. Dennis, *Stream Data Types for Signal Processing*, technical report, September, 1992.

[Dong79]

J.J. Dongarra and A.R. Hinds, "Unrolling Loops in FORTRAN," *Software-Practice and Experience*, **Vol. 9**, March, 1979.

[Egol93]

T. Egolf, S. Famorzadeh, and V. Madisetti, *On Library-Based Compiler Optimization for Programmable DSPs*, technical report, School of Electrical Engineering, Georgia Institute of Technology, December, 1993.

[Fabr82]

J. Fabri, *Automatic Storage Optimization*, UMI Research Press, 1982.

[Fish84]

J. A. Fisher, "The VLIW Machine: A Multiprocessor for Compiling Scientific Code," *IEEE Computer Magazine*, **Vol. 17**, **No. 7**, July, 1984.

[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.

[Garb90]

J. Garbers, H. J. Promel, and A. Steger, "Finding Clusters in VLSI Circuits," *Proceedings of the IEEE International Conference on Computer-Aided Design*, Santa Clara, November, 1990.

[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.

[Geni90]

D. Genin, P. Hilfinger, J. Rabaey, C. Scheers, and H. De Man, "DSP Specification Using the Silage Language," *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Albuquerque, April, 1990.

[Gera92]

A. Gerasoulis and T. Yang, "A Comparison of Clustering Heuristics for Scheduling Directed Acyclic Graphs on Multiprocessors," *Journal of Parallel and Distributed Computing*, **Vol. 16**, 1992.

[Godb73]

S. S. Godbole, "On Efficient Computation of Matrix Chain Products," *IEEE Transactions on Computers*, **Vol. C-32**, **No. 9**, September, 1973.

[Golu80]

M. C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, 1980.

[Gurd85]

J. R. Gurd, C. C. Kirkham, and I. Watson, "The Manchester Prototype Dataflow Computer," *Communications of the ACM*, **Vol. 28**, **No. 1**, January, 1985.

[Hart88]

J. Hartung, S. L. Gay, and S. G. Haigh, "A Practical C Language Compiler/Optimizer for Real Time Implementation on a Family of Floating Point DSPs," *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, New York, April, 1988.

[Hend92]

L. Hendren, G. Gao, and C. Mukerji, “A Register Allocation Framework Based on Hierarchical Cyclic Interval Graphs,” *Lecture Notes in Computer Science*, February, 1992.

[Henn90]

J. L. Hennessy and D. A. Patterson, *Computer Architecture A Quantitative Approach*, Morgan Kaufmann, 1990.

[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.

[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.

[John75]

Johnson, “Finding all the Elementary Circuits of a Directed Graph,” *SIAM Journal of Computing*, **Vol. 4**, **No. 1**, March, 1975.

[Kafk90]

S. M. Kafka, “An Assembly Source Level Global Compactor For Digital Signal Processors,” *Proceedings of the International Conference on Acous-*

tics, Speech, and Signal Processing, Albuquerque, April, 1990.

[Kane87]

G. Kane, *MIPS RISC Architecture*, Prentice-Hall, 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.

[Karp66]

R. M. Karp and R. E. Miller, "Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing," *SIAM Journal of Applied Math*, **Vol. 14**, **No. 6**, November, 1966.

[Kell61]

J. Kelly, Lochbaum, and V. Vyssotsky, "A Block Diagram Compiler," *Bell System Technical Journal,* **Vol. 40**, **No. 3**, May, 1961.

[Kogg81]

P. M. Kogge, *The Architecture of Pipelined Computers*, McGraw Hill, 1981.

[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," presented at *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, 1982.

[Lee88a]

E. A. Lee, “Recurrences, Iteration and Conditionals in Statically Scheduled Block Diagram Languages,” *VLSI Signal Processing III*, IEEE Press, 1988.

[Lee88b]

E. A. Lee, “Programmable DSP Architectures: part I,” *IEEE ASSP Magazine*, **Vol. 5**, **No. 4**, October, 1988.

[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 Par-*

allel 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.

[Lee94]

E. A. Lee, *Dataflow Process Networks*, draft, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, April, 1994.

[Leis83]

C. E. Leiserson, F. M. Rose, and J. B. Saxe, "Optimizing Synchronous Circuitry by Retiming," *Third Caltech Conference on VLSI*, March, 1983.

[Liao93]

G. Liao, G. R. Gao, E. Altman, and V. K. Agarwal, *A Comparative Study of DSP Multiprocessor List Scheduling Heuristics*, technical report, School of Computer Science, McGill University.

[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.

[Mess84]

D. G. Messerschmitt, "Structured Interconnection of Signal Processing Programs," *Proceedings of Globecom*, Atlanta, 1984.

[Muel92]

F. Mueller and D. B. Whalley, "Avoiding Unconditional Jumps by Code

Replication," *International Conference on Programming Language Design and Implementation*, San Francisco, June, 1992.

[Mura71]

Y. Muraoka, *Parallelism Exposure and Exploitation in Programs*, Ph.D. thesis, Report 71-424, Department of Computer Science, University of Illinois at Urbana-Champaign, February, 1971.

[Murt93]

P. K. Murthy, *Multiprocessor DSP Code Synthesis in Ptolemy*, Master's project report, Memorandum No. UCB/ERL M93/6, 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 Programs", *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Adelaide, Australia, April, 1994.

[Murt94b]

P. K. Murthy and E. A. Lee, *On the Optimal Blocking Factor for Blocked Periodic Schedules*, Memorandum No. UCB/ERL M94/46, Electronics Research Laboratory, University of California at Berkeley, June, 1994.

[Najj92]

W. A. Najjar, R. Roh, and A. P. Wim Bohm, "Initial Performance of a Bottom-Up Clustering Algorithm for Dataflow Graphs," *Proceedings of the IFIP Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, Orlando, January, 1993.

[Ohal91]

D. R. O'Hallaron, *The Assign Parallel Program Generator*, Memorandum

CMU-CS-91-141, School of Computer Science, Carnegie Mellon University, May, 1991.

[Parh91]

K. K. Parhi and D. G. Messerschmitt, "Static Rate-Optimal Scheduling of Iterative Data-Flow Programs via Optimum Unfolding," *IEEE Transactions on Computers*, **Vol. 40**, **No. 2**, February, 1991.

[Pino94]

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*, to appear in 1994.

[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.

[Prin91]

H. Printz, *Automatic Mapping of Large Signal Processing Systems to a Parallel Machine*, Ph.D. thesis, Memorandum CMU-CS-91-101, School of Computer Science, Carnegie Mellon University, May, 1991.

[Prin92]

H. Printz, "Compilation of Narrowband Spectral Detection Systems for Linear MIMD Machines," *Proceedings of the International Conference on Application Specific Array Processors*, Berkeley, 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.

[Schm91]

U. Schmidt and K. Caesar, “Datawave: a Single-Chip Multiprocessor for Video Applications,” *IEEE Micro Magazine*, **Vol. 11**, **No. 3**, June, 1991.

[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.

[Tarj72]

R. Tarjan, “Depth-First Search and Linear Graph Algorithms,” *SIAM Journal of Computing*, June, 1972.

[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.

[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.

[Wolf89]

M. Wolfe, *Optimizing Supercompilers for Supercomputers*, MIT Press, 1989.

[Wolf91]

M. E. Wolf and M. S. Lam, "A Data Locality Optimizing Algorithm," *Proceedings of the ACM Conference on Programming Language Design and Implementation*, San Francisco, June, 1991.

[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.

[Zivo93]

V. Zivojnovic, S. Ritz, and H. Meyr, "Multirate Retiming: A Powerful Tool for Hardware/Software Codesign," technical report, Institute for Integrated Systems in Signal Processing, Aachen University of Technology, 1993.

INDEX OF TERMINOLOGY AND NOTATION