ABSTRACT

Title of dissertation: INTEGRATED SOFTWARE SYNTHESIS FOR SIGNAL PROCESSING APPLICATIONS

Ming-Yung Ko, Doctor of Philosophy, 2006

Dissertation directed by: Professor Shuvra S. Bhattacharyya
Department of Electrical and Computer Engineering, and
Institute for Advanced Computer Studies

Signal processing applications usually encounter multi-dimensional real-time performance requirements and restrictions on resources, which makes software implementation complex. Although major advances have been made in embedded processor technology for this application domain — in particular, in technology for programmable digital signal processors — traditional compiler techniques applied to such platforms do not generate machine code of desired quality. As a result, low-level, human-driven fine-tuning of software implementations is needed, and we are therefore in need of more effective strategies for software implementation for signal processing applications.

In this thesis, a number of important memory and performance optimization problems are addressed for translating high-level representations of signal processing applications into embedded software implementations. This investigation centers around signal-processing-oriented dataflow models of computation. This form of dataflow provides a coarse grained modeling approach that is well-suited to the signal processing domain and is increasingly supported by commercial and research-oriented tools for design and implementation of signal processing systems.

Well-developed dataflow models of signal processing systems expose high-level application structure that can be used by designers and design tools to guide optimization of hardware and software implementations. This thesis advances the suite of techniques available for optimization of software implementations that are derived from the application structure exposed from dataflow representations. In addition, the specialized architecture of programmable digital signal processors is considered jointly with dataflow-based analysis to streamline the optimization process for this important family of embedded processors. The specialized features of programmable digital signal processors that are addressed in this thesis include parallel memory banks to facilitate data parallelism, and signal-processing-oriented addressing modes and address register management capabilities.

The problems addressed in this thesis involve several inter-related features, and therefore an integrated approach is required to solve them effectively. This thesis proposes such an integrated approach, and develops the approach through formal problem formulations, in-depth theoretical analysis, and extensive experimentation.

INTEGRATED SOFTWARE SYNTHESIS FOR SIGNAL PROCESSING APPLICATIONS

by

Ming-Yung Ko

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2006

Advisory Committee

Professor Shuvra S. Bhattacharyya, Chair/Advisor
Professor Rajeev Barua
Professor Rama Chellappa
Professor Gang Qu
Professor Chau-Wen Tseng, Dean’s Representative

© Copyright by

Ming-Yung Ko

2006

DEDICATION

to Daphne (Shu-Hua), Ruth, and Josephine

ACKNOWLEDGMENTS

First above all, I would like to acknowledge my Heavenly Father (and the Son, Jesus Christ). It is He who brought me to study in this country and know His truth and way. It is with the Lord’s provision and encouragement that I can stand up out of the challenging time during my graduate study life. I am a blessed servant of His in that He graciously gave me countless “luck” in the achievement of my research work. *Unless the LORD builds the house, its builders labor in vain. Unless the LORD watches over the city, the watchmen stand guard in vain (Psalm 127:1).*

I wish to thank my advisor, Professor Shuvra Bhattacharyya, for his guidance and encouragement in this endeavor. His enthusiasm for research and insights into many technical problems have motivated me greatly during the course of this work. He is very patient and experienced on editing technical documents so that a number of my ordinary writings turn out to be masterpiece through his polishing. His endless encouragement always supports students to strive for the best.

Financial support for this thesis was provided in part by the Semiconductor Research Corporation (contract number: 2001-HJ-905) and Agilent Technologies. I would like to acknowledge their back for advanced computer research.

It has been a pleasure to interact with many students in the DSPCAD group and the Embedded Systems Research Lab, including Vida, Dong-Ik, Nitin, Neal, Shahrooz, Sumit, Fuat, Sadagopan, Lin, Chung-Ching, Chia-Jui, Sankalita, Mainak, Ankush, Sebastian, Celine, and Ivan. Our collaborations and discussions have strengthened this thesis and enriched my experience here. Especially Vida, we have gone through together a number of course work, internship at Information Sciences Institute (ISI), and the final stage of thesis completion.

Finally and the most important, I would like to thank my family for their love, support, and patience during these years while I focused on studies. Thanks be to my wife, Shu-Hua, for taking care of house chores, raising and educating children, being accompanying me going through numerous peaks and valleys in my graduate study life, and, most important of all, giving spiritual support and inspiration. Thanks also be to my two lovely girls, Ruth and Josephine, for bringing several delightful memories.

TABLE OF CONTENTS

LIST OF FIGURES

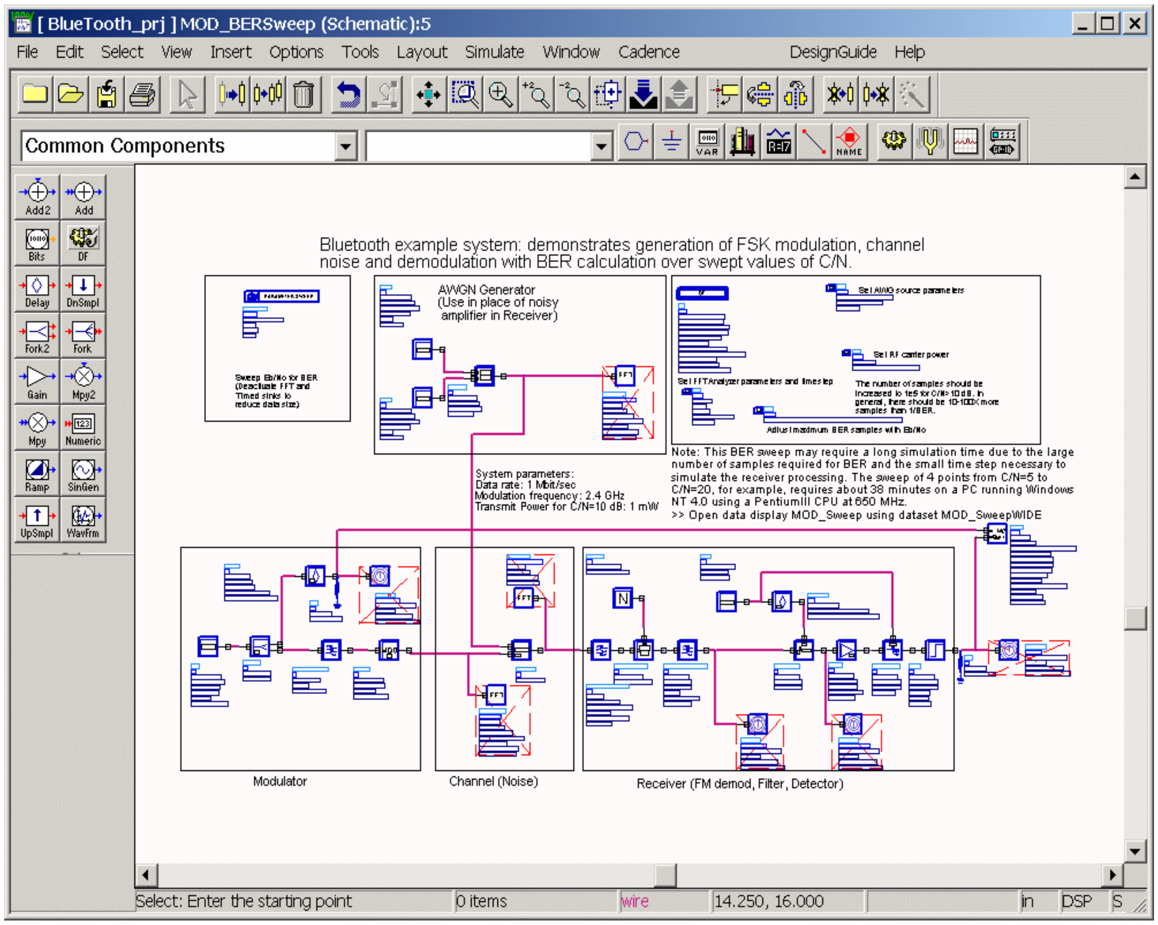
Chapter 1. Introduction

Embedded computer systems are computer systems that are specialized for particular applications or application domains. Such systems can be found almost everywhere in our daily life, such as in telephones, stereos, automobiles, PDAs (personal digital assistants), elevators, digital cameras, VCRs, microwave ovens, and televisions. One trend in the electronics industry in recent decades is the rapidly growing volume and variety of portable consumer products, that involve embedded computer systems. This trend drives the need of applications with high complexity and scale; short time-to-market development windows; strict real-time performance requirements; economic utilization of limited resources and overall financial cost effectiveness; and efficient power consumption management. Due to the rapidly increasing importance of embedded systems, and the complexity of their underlying implementation constraints, it is important to investigate systematic approaches to address their constraints and optimization objective.

Among the various application domains of embedded systems, computations in the **digital signal processing (DSP)** context present several common features that are realized in many modern embedded systems. DSP software is often implemented on **programmable digital signal processors (PDSPs)**, which are microprocessors that are specialized for DSP applications. In addition to basic computing capabilities, PDSPs incorporate specialized address generation units, datapath structures, addressing modes, and memory architectures. Due to the real-time constraints and optimization objectives, as well as the specialized hardware structure, traditional compiler optimization techniques are inadequate in generating quality code on PDSPs to produce expected performance. Therefore, most PDSP software development still adopts the scheme of piecing together code from manually tuned assembly program libraries. The design process is therefore tedious and error prone, and efficient design methodologies for software synthesis on PDSPs are crucial.

On the other hand, block-diagram-based graphical design environments has been increasingly adopted in DSP application development. Such graphical programming environments, that allow applications to be specified as hierarchy of block diagrams, offer several advantages. Perhaps the most obvious of these advantages is their intuitive appeal. In addition to offering intuitive appeal, the specification of systems in terms of connections between pre-defined, encapsulated functional blocks naturally promotes desirable software engineering practices such as modularity and code reuse. There are also a number of more technical advantages of graphical DSP design tools. These advantages hinge on the use of appropriate models of computation to provide the precise underlying block diagram semantics. Such models of computation provides formal methodologies in design verification, bounded memory determination, deadlock detection, and most important, automatic synthesis of implementations. One commercial example of such graphical design tools is given in Figure 1.1 for Advanced Design System (ADS) by Agilent.

This thesis work is focused on software synthesis for DSP applications. **Software synthesis** is a compiler technology which refers to automated derivation of a software implementation (application program) in some programming language given a library of subprogram modules, a subset of selected modules from this library, and a specification of how these selected modules interact to implement the target application. Software synthesis is particularly suitable for large scaled, modular application implementation in block-diagram-based specification. Moreover, to promote execution performance, the compilation process needs to be aware of and exploit the hardware strengths of the target PDSP or processing units. As a result, the optimization problems involved are complex to solve due to the knowledge required for the underlying computation models, PDSP computing architecture, as well as various stringent constraints. A sketch of block-diagram-based software synthesis flow is illustrated in Figure 1.2.



Bluetooth example system: demonstrates generation of FSK modulation, channel noise and demodulation with BER calculation over swept values of C/N.

Sweep Eb/No for BER (Deactivate FFT and Timed sinks to reduce data size)

The number of samples should be increased to 1e5 for C/N>10 dB. In general, there should be 10-100X more samples than 1/BER.

System parameters:
Data rate: 1 Mbit/sec
Modulation frequency: 2.4 GHz
Transmit Power for C/N=10 dB: 1 mW

Note: This BER sweep may require a long simulation time due to the large number of samples required for BER and the small time step necessary to simulate the receiver processing. The sweep of 4 points from C/N=5 to C/N=20, for example, requires about 38 minutes on a PC running Windows NT 4.0 using a PentiumIII CPU at 650 MHz.
>> Open data display MOD_Sweep using dataset MOD_SweepWIDE

Select: Enter the starting point

Figure 1.1 A snapshot of Advanced Design System by Agilent.

1.1. Contributions of this thesis

Specifically, the thesis work is concerned about optimization problems in the DSP software synthesis flow. On one hand, memory resource is typically tight in the PDSPs due to the demands for light weighted and compact sized portable electronic devices. Therefore, minimization of code and data memory space consumption can be found in a number of research work. On the other hand, memory structure in PDSPs is usually tailored to promote DSP execution efficiency. Adequate exploitation of such memory organization helps promoting memory access efficiency and overall system performance. There are also quite a few other specialized empowered functionalities offered in modern PDSPs. Quite often, to achieve their strengths, memory space overhead is necessary. In this thesis, a set of problems of memory space requirement reduction, appropriate utilization of specialized memory structure, and execution performance improvement under memory space constraints is addressed. The targeted problems and our contributions are summarized in the following subsections.

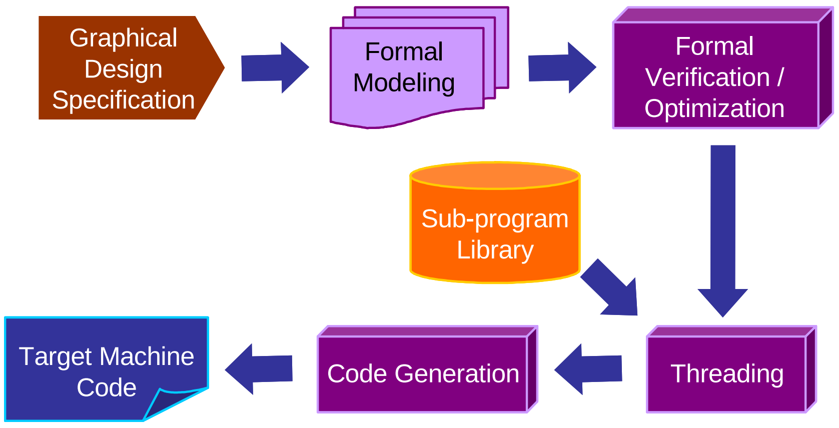


Figure 1.2 Block-diagram-based software synthesis flow.

1.1.1. Nested procedure synthesis

Several papers have studied inline synthesis (i.e., insertion of entire code blocks from a subprogram library) to jointly minimize data and code size, systematic procedure synthesis (i.e., placement of small-sized pointers to code blocks), however, has not been found, which is supposed to go beyond the restrictions for compact inline synthesis. In this thesis, nested procedure synthesis for a special case of applications is first proposed to generate proven minimal data space consumption. The synthesis algorithm is then extended to handle arbitrary cases through systematic decompositions on the block-diagram-based specification. Formal theories are rigorously established to ensure polynomial-bounded number of procedures synthesized. Analyses are then performed to identify application characteristics where the strengths of the proposed algorithms can be appropriately unleashed.

1.1.2. Memory-constrained block processing optimization

DSP applications involve processing long streams of input data and it is important to take into account this form of processing when implementing software. Task-level block processing is a useful transformation that improves execution performance by allowing subsequences of data (instead of scalar) items to be processed per task invocation. In this way, several benefits can be obtained, including reduced context switch overhead, increased memory locality, improved utilization of processor pipelines, and use of more efficient DSP oriented addressing modes. On the other hand, block processing generally results in increased memory space requirements since it effectively increases the numbers of data associated with tasks. Previous studies did not offer software designers a clear map of memory-performance trade-off, which is important in design space exploration.

In this thesis work, algorithms are developed to carefully take into account memory constraints to achieve efficient block processing configurations within given memory space limitations. Theories are derived to relate block processing and memory space overhead through rearranging iteration counts of looped task invocations, where challenges are faced in dealing with task execution sequence validation and interactions of prime factorization results of loop iteration counts. Experimental results indicate that our algorithms evaluate optimal memory-constrained block processing solutions most of the time, no matter for real applications or randomly generated benchmarks.

1.1.3. Parameterizable hierarchical looped execution

Previous studies on inline synthesis are primarily built on hierarchical looped execution with fixed loop iterations. Not only flexible execution patterns can not be captured in the synthesized implementation, but also aggressive data space compaction is prohibited. A new format of looping constructs is devised in this thesis to allow configureable functions in describing loop iterations. Moreover, the format takes into consideration the underlying hardware strengths, e.g., constant increments and fast looping instructions. Setting of run-time parameters is also acceptable such that a single looping structure in this format is capable of representing multiple looping instances with similar structures. The run-time parameterizability is especially suitable for FPGA synthesis and relevant experiments are conducted to illustrate significant FPGA area savings and frequency enhancement.

1.1.4. Optimization for dual-memory bank architecture

A number of modern PDSPs are quipped with dual homogeneous memory banks in their architecture to enable data memory parallelism. While the previous work primarily focuses on joint memory bank assignment and register allocation for scalar variables, memory space occupied by arrays, which account for a large portion of memory cost in DSP applications, have not been addressed.

Unlike the problem formulation for scalar variables, where complex data parallelism restrictions are presented and probabilistic search is often resorted, bipartite structure is observed in a particular DSP-oriented computation model to permit optimal parallel memory bank accesses. However, memory bank capacity requirement alone still makes the optimization problem intractable and an NP-hard reduction is given. Hence, a heuristic is proposed to aggressively narrow the bank capacity gap and computes optimal solutions most of the time for real DSP applications.

1.2. Outline of the thesis

The outline of the thesis is as follows. Chapter 2 presents the background of block-diagram-based DSP software synthesis and introduces an underlying model of computation. Chapter 3 discusses a systematic synthesis approach through procedure implementation. Chapter 4 examines task-level block processing optimization under memory space constraints. Chapter 5 proposes a parameterizable hierarchical loop format and associated derivation algorithms, which demonstrate effectiveness in both FPGA and software synthesis. Chapter 6 explores the joint optimization of memory space reduction and parallelism for computers with dual memory banks. Conclusion and summary of the research topics studied in this thesis are drawn in Chapter 7.

Chapter 2. Software synthesis background

2.1. Software synthesis

In our context, software synthesis is a coarse-grained compiler technology that involves the automated derivation of a software implementation in some programming language given a library of subprogram modules and a specification of how these selected modules interact to implement the target application. This compiler technology is employed in graphical design environments, which may be based on block diagram representations, textual representation, or a mixture of the two.

To demonstrate how a program is synthesized, we use Figure 2.1 ported from Ptolemy project as an example. Each rectangular block in the figure represents a subpro-

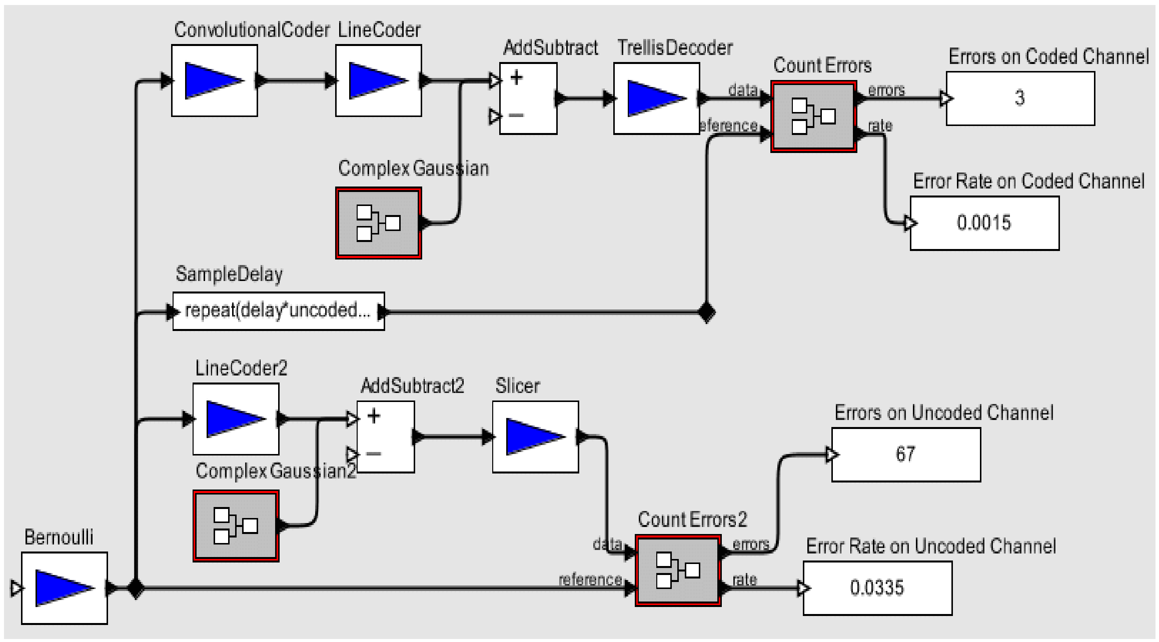
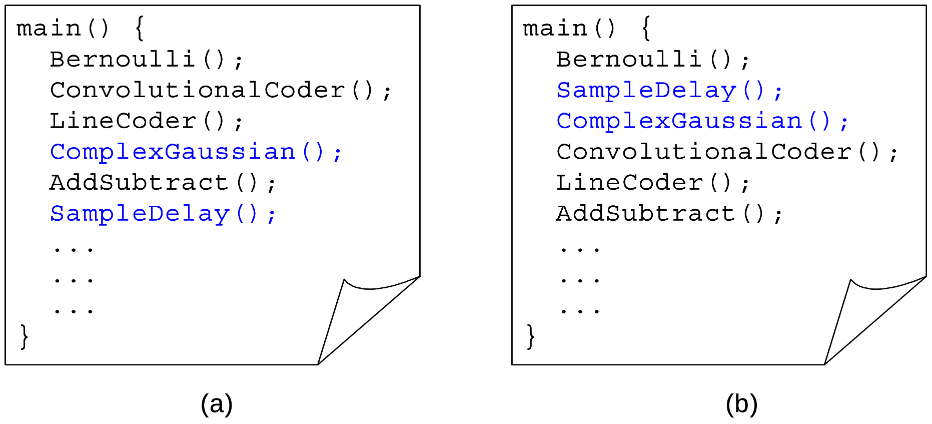


Figure 2.1 An example to demonstrate software synthesis.

gram and has corresponding pre-implemented code. Arrows connecting blocks indicate data dependences between subprograms. Applications specified in this way do not specify explicit orders in executing subprograms and it is up to the underlying compiler to do the calculation. Two valid program implementations (suppose in C programming language) are presented in Figure 2.2 as demonstration. Note that the placement of *SampleDelay()* and *ComplexGaussian()* in the synthesized C programs could be different and are still considered valid instantiations. In addition to the two cases given in Figure 2.2, there are several other possibilities, if those programs are generated in a way such that the data dependences are not violated. One the other hand, subprogram execution order decision, also termed **scheduling** problem, has significant impacts on various optimization problems, e.g., code/data space consumption reduction, and they will be discussed later in several places of this thesis.

Besides the scheduling problem, there are several other software synthesis optimization problems. To name a few, a subprogram can be implemented as inlined code or procedure instantiation, where each strategy has their pros and cons. The problem of



```
main() {
  Bernoulli();
  ConvolutionalCoder();
  LineCoder();
  ComplexGaussian();
  AddSubtract();
  SampleDelay();
  ...
  ...
  ...
}
```

(a)

```
main() {
  Bernoulli();
  SampleDelay();
  ComplexGaussian();
  ConvolutionalCoder();
  LineCoder();
  AddSubtract();
  ...
  ...
  ...
}
```

(b)

Figure 2.2 Two examples of programs synthesized for Figure 2.1.

minimizing subprogram initiation overhead, if any, is critical to the synthesized implementation. Incorporation of parameterizable configurations to generate multiple, conditional software implementation instances is also an interesting research topic.

2.2. Background of dataflow

Dataflow is the most used computation model underlying many popular graphical DSP system design tools. **Dataflow** specifications are directed graphs in which nodes (**actors**) represent computational tasks and edges represent data dependences. Actors are activated when sufficient inputs are available, and FIFO queues (or **buffers**) are usually allocated to hold data as it is transferred across the edges. **Delays**, which model instantiations of the z^{-1} operator, are also associated with edges, and are typically implemented as initial data values in the associated buffers. Examples of commercial DSP design tools that incorporate dataflow semantics include System Canvas from Angeles Design Systems [38], SPW from Cadence Design Systems, ADS from Agilent, Cocentric System Studio from Synopsys [11], LabVIEW from National Instruments, GEDAE from Lockheed, and the Autocoding Toolset from Management, Communications, and Control, Inc. [47]. Research-oriented tools and languages related to dataflow-based DSP design include DIF from University of Maryland [21], Ptolemy from U. C. Berkeley [17], PeaCE from Seoul National University [53], GRAPE from K. U. Leuven [29], Compaan from Leiden University [50], and StreamIt from MIT [55].

For DSP system implementation, it is often important to analyze the memory requirements associated with the FIFOs for the dataflow edges. In this context, the **buffer**

cost (or **buffer size**) of a buffer means the maximum amount of data (in terms of bytes) that resides in the buffer at any instant.

For DSP design tools, **synchronous dataflow (SDF)** [30] is the most commonly used form of dataflow. The major advantage of SDF is the potential for static analysis and optimization. SDF imposes the restriction that for each edge in the dataflow graph, the numbers of data values produced by each invocation of the source actor and the number of data values consumed by each invocation of the sink actor are constant values. Given an SDF edge e, the numbers of data values produced $prd(e)$ and consumed $cns(e)$ are fixed at compile time for each invocation of the source actor $src(e)$ and sink actor $snk(e)$, respectively.

A **schedule** is a sequence of actor invocations (or **firings**). We compile an SDF graph by first constructing a **valid schedule**, which is a finite schedule that fires each actor at least once, and does not lead to unbounded buffer accumulation (if the schedule is repeated indefinitely) nor buffer underflow (deadlock) on any edge. To avoid buffer overflow and underflow problems, the total amount of data produced and consumed is required to be matched on all edges. In [30], efficient algorithms are presented to determine whether or not a valid schedule exists for an SDF graph, and to determine the minimum number of firings of each actor in a valid schedule. We denote the **repetitions count** of an actor as this minimum number of firings, and we collect the repetitions counts for all actors in the **repetitions vector**. The repetitions vector is indexed by the actors in the SDF graph and it is denoted by q.

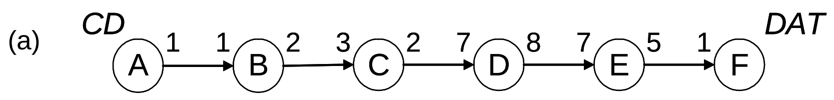
Given an SDF edge e and the repetitions vector q, the **balance equation** for e is written as

$$q(src(e))prd(e) = q(snk(e))cns(e)$$

and the quantity is denoted as **total number of samples exchanged (TNSE)** on e.

To save code space, actor firings can be incorporated within loop constructs to form **looped schedules**. Looped schedules group sequential firings into schedule loops; each such loop is composed of a loop iteration count and one or more iterands. In addition to being actor firings, iterands can also be subschedules, and therefore, it is possible to represent nested-loop constructs.

The notation we use for a schedule loop L is $L = (nI_1I_2\ldots I_m)$, where n denotes the iteration count and $I_1, I_2, \ldots, I_m$ denote the iterands of L. **Single appearance schedules (SASs)** are schedules in which each actor appears only once. In inlined code implementation, an SAS contains a single copy of code for every actor and results in minimal code space requirements. For an acyclic SDF graph, an SAS can easily be derived from a topological sorting of the actors. However, such an SAS often requires relatively high buffer cost. A more memory-efficient method of SAS construction is to perform a certain form of dynamic programming optimization (called **DPPO** for **dynamic programming post optimization**) over a topological sort to generate a buffer-efficient, nested looped schedule [7]. In this thesis, unless stated otherwise, we generally employ the **acyclic pairwise grouping for adjacent nodes (APGAN)** algorithm [7] for the generation of topological sorts and the DPPO method described above for the optimization of these topological sorts into various more buffer-efficient forms. Figure 2.3 is drawn to demonstrate an SDF graph representation of **CD (compact disc)** to **DAT (digital audio tape)** sample rate con-



(b) $q(A, B, C, D, E, F) = (147, 147, 98, 28, 32, 160)$

(c) $SAS = ((49(3AB)(2C))(4(7D)(8E(5F))))$

Figure 2.3 (a) An SDF graph modeling of CD to DAT rate conversion. (b) The repetitions vector. (c) A SAS computed by APGAN/DPPO.

version, along with the associated repetitions vector, and a SAS computed by APGAN and DPPO.

Chapter 3. Nested Procedure Synthesis

Synthesis of DSP software from dataflow-based formal models is an effective approach for tackling the complexity of modern DSP applications. In this chapter, an efficient method is proposed for applying subroutine call instantiation of module functionality when synthesizing embedded software from a dataflow specification. The technique is based on a novel recursive decomposition of subgraphs in a cluster hierarchy that is optimized for low buffer size. Applying this technique, one can achieve significantly lower buffer sizes than what is available for minimum code size inlined schedules, which have been the emphasis of prior software synthesis work. Furthermore, it is guaranteed that the number of procedure calls in the synthesized program is polynomially bounded in the size of the input dataflow graph, even though the number of module invocations may increase exponentially. This recursive decomposition approach provides an efficient means for integrating subroutine-based module instantiation into the design space of DSP software synthesis. The experimental results demonstrate a significant improvement in buffer cost, especially for more irregular multi-rate DSP applications, with moderate code and execution time overhead. A preliminary summary of part of this chapter is published in [25].

3.1. Introduction and Related Work

A significant body of theory and algorithms has been developed for synthesis of software from dataflow-based block diagram representations. Many of these techniques pertain to the SDF model. In [7], algorithms are developed to optimize buffer space while obeying the constraint of minimal code space. A multiple objective optimization is proposed in [57] to compute the full range of Pareto-optimal solutions in trading off code size, data space, and execution time. Vectorization can be incorporated into SDF graphs to reduce the rate of context switching and enhance execution performance [28][46].

In this chapter, an efficient method is proposed for applying subroutine call instantiation of module functionality to minimize buffering requirements when synthesizing embedded software from SDF specifications. The technique is based on a novel recursive decomposition of subgraphs in a cluster hierarchy that is optimized for low buffer size. Applying this technique, one can achieve significantly lower buffer sizes than what is available for minimum code size inlined schedules, which have been the emphasis of prior software synthesis work. Furthermore, it is guaranteed that the number of procedure calls in the synthesized program is polynomially bounded in the size of the input dataflow graph, thereby bounding the code size overhead. Having such a bound is particularly important because the number of module invocations may increase exponentially in an SDF graph. Our recursive decomposition approach provides an efficient means for integrating subroutine-based module instantiation into the design space of DSP software synthesis.

In [52], an alternative buffer minimization technique through transforming looped schedules is investigated. The transformation works on a certain schedule tree data structure, and the computational complexity of the transformation is shown to be polynomial in the number of leaf nodes in this schedule tree. However, since leaf nodes in the schedule tree correspond to actor appearances in the schedule, there is in general no polynomial bound in terms of the size of the SDF graph on the number of these leaf nodes. Therefore, no polynomial bound emerges on the complexity of the transformation technique in terms of SDF graph size. In contrast, our graph decomposition strategy extends and hierarchically applies a two-actor SDF graph scheduling theory that guarantees achieving minimal buffer requirements [7], and a number of theorems are developed in this chapter to ensure that the complexity of our approach is polynomially bounded in the size of the SDF graph.

Buffer minimization and use of subroutine calls during code synthesis have also been explored in the phased scheduling technique [23]. This work is part of the StreamIt language [55] for developing streaming applications. Phased scheduling applies to a restricted subset of SDF graphs, in particular each basic computation unit (called a filter in StreamIt) allows only a single input and output. In contrast, the recursive graph decomposition approach applies to all SDF graphs that have single appearance schedules (this class includes all properly-constructed, acyclic SDF graphs), and furthermore, can be applied outside the tightly interdependent components of SDF graphs that do not have single appearance schedules. Tightly interdependent components are unique, maximal subgraphs that exhibit a certain form of data dependency [7]. Through extensive experiments with single appearance scheduling, it has been observed that tightly interdependent components arise only very infrequently in practice [7]. Integrating phased scheduling concepts with the decomposition approach presented in this chapter is an interesting direction for further work.

Panda surveys data memory optimization techniques for compiling high level languages (HLLs), such as C, including techniques such as code transformation, register allocation, and address generation [40]. Due to the instruction-level parallelism capability found in many DSP processors, the study of independent register transfers is also a useful subject. The work of [32] investigates an integer programming approach for code compaction that obeys exact timing constraints and saves code space as well. Since code for individual actors is often specified by HLLs, several such techniques are complementary to the techniques developed in this thesis. In particular HLL compilation techniques can be used for performing intra-actor optimization in conjunction with the inter-actor, SDF-based optimizations developed in this thesis.

3.2. Recursive Decomposition of a Two-Actor SDF Graph

Given a two-actor SDF graph as shown on the left in Figure 3.1, we can recursively generate a schedule that has a buffer memory requirement of the least amount possible. The scheduling technique works in the following way: given the edge AB, and $prd(AB) = n > cns(AB) = m$, we derive the new graph shown on the right in Figure 3.1 where the actor set is $\{A_1, B_1\}$ and $prd(A_1B_1) = n \bmod m$. The actor A_1 is a hierarchical actor that represents the schedule $A(\lfloor n/m \rfloor B)$, and B_1 just represents B. Consider a minimum buffer schedule for the reduced graph, where we replace occurrences of A_1 by $A(\lfloor n/m \rfloor B)$, and occurrences of B_1 are replaced by B. For example, suppose that $n = 3$ and $m = 2$. Then $3 \bmod 2 = 1$, the minimum buffer schedule for the reduced graph would be $A_1A_1B_1$, and this would result in the schedule $ABABB$ after the replacement. As can be verified, this later schedule is a valid schedule for the original graph, and

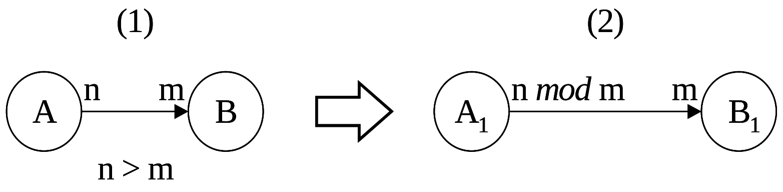


Figure 3.1 A two-actor SDF graph and its reduced version.

is also a minimum buffer schedule for it, having a buffer memory requirement of $n + m - 1$ as expected.

However, the advantage of the reduced graph is depicted in Figure 3.2: the schedule for the reduced graph can be implemented using procedure calls in a way that is more parsimonious than simply replacing each occurrence of A and B in $ABABB$ by procedure calls. This latter approach would require 5 procedure calls, whereas the hierarchical implementation depicted in Figure 3.2 requires only 3 procedure calls. The topmost procedure implements the SAS $A_1A_1B_1 = (2A_1)B_1$, where A_1 is really a procedure call; this procedure call implements the SAS AB, which in turn call the actors A and B. A procedure B_1 for actor B is also needed because it is called more than once: A_1 and the topmost procedure. Of-course, we could implement the schedule $ABABB$ more efficiently than simply using five procedure calls; for example, we could generate inline code for the schedule $(2AB)B$; this would have 3 blocks of code: two for B, and one for A. We would have to do a trade-off analysis to see whether the overhead of the 3 procedure calls would be less than the code-size increase of using 3 appearances (instead of 2). We would also like to clarify that **procedure calls** in this chapter refer to procedure instantiation in software synthesis, rather than run-time procedure invocation.

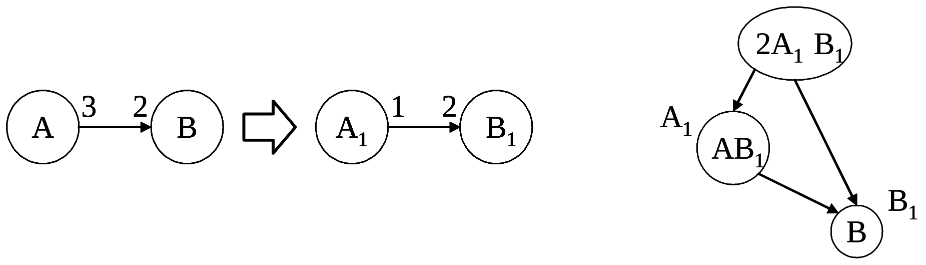


Figure 3.2 A hierarchical procedural implementation of a minimum buffer schedule for the SDF graph on the left.

We first state an important theorem from [7]:

Theorem 3.1. For the two-actor SDF graph depicted on the left in Figure 3.1, the minimum buffer requirement over all schedules is given by $n + m - \gcd(n, m)$.

Proof: See [7].

We denote $n + m - \gcd(n, m)$ for a two-actor SDF graph depicted on the left in Figure 3.1 as the **VBMLB (the buffer memory lower bound over all valid schedules)**. The definition of VBMLB also applies to an SDF edge. Similarly, for arbitrary SDF graphs, the VBMLB for a graph can be defined as the sum of VBMLBs over all edges.

Theorem 3.2 shows that the preservation of the minimum buffer schedule in the reduced graph in the above example is not a coincidence.

Theorem 3.2. The minimum buffer schedule for the reduced graph on the right in Figure 3.1 yields a minimum buffer schedule for the graph on the left when the appropriate substitutions of the actors are made.

Proof: Let $\gcd(n, m) = g$. The equation $\gcd(n \mathit{ mod } m, m) = g$ must hold since a fundamental property of the gcd is that $\gcd(n, m) = \gcd(n \mathit{ mod } m, m)$. So the minimum buffer requirement for the reduced graph is given by $n \mathit{ mod } m + m - g$ from Theorem 3.1. Now, when A_1 is replaced by $A(\lfloor n/m \rfloor B)$ to get a schedule for the original graph, we see that the maximum number of tokens is going to be reached after a firing of A since firings of B consume tokens. Since the maximum number of tokens reached in the reduced graph on edge A_1B_1 is $n \mathit{ mod } m + m - g$, the maximum number reached on AB when we replace A_1 by $A(\lfloor n/m \rfloor B)$ will be

$$n \mathit{ mod } m + m - g + \left\lfloor \frac{n}{m} \right\rfloor m = n + m - g.$$

Hence, the theorem is proved. **QED**.

Theorem 3.3. An SAS for a two-actor graph satisfies the VBMLB if and only if either $n \mid m$ (n is dividable by m) or $m \mid n$.

Proof: (Forward direction) Assume WLOG that $n > m$. Then the SAS is going to be $((m/(\gcd(n, m)))A)((n/(\gcd(n, m)))B)$. The buffering requirement of this schedule is $mn/\gcd(n, m)$. Since this satisfies the VBMLB, we have

$$\frac{m}{\gcd(n, m)}n = m + n - \gcd(n, m). \tag{3.1}$$

Since $n > m$, we have to show that (3.1) implies $m \mid n$. The contrapositive is that if $m \mid n$ does not hold then Equation (3.1) does not hold. Indeed, if $m \mid n$ does not hold, then $\gcd(n, m) < m$, and $m/(\gcd(n, m)) \geq 2$. In the R.H.S. of (3.1), we have $m - \gcd(n, m) < m < n$, meaning that the R.H.S. is $< 2n$. This shows that (3.1) cannot hold.

The reverse direction follows easily since if $m \mid n$, then the L.H.S. is n, and the R.H.S. is $m + n - m = n$. **QED**.

A 2-actor SDF graph where either $n \mid m$ or $m \mid n$ is called a **perfect SDF graph (PSG)** in this thesis.

Theorem 3.4. A minimum buffer schedule for a two-actor SDF graph can be generated in the recursive hierarchical manner by reducing the graph until either $n \mid m$ or $m \mid n$.

Proof: This follows by Theorems 3.2 and 3.3 since reduction until $n \mid m$ or $m \mid n$ is necessary for the terminal schedule to be an SAS by Theorem 3.3, and the back substitution process preserves the VBMLB by Theorem 3.2.

Theorem 3.5. The number of reductions needed to reduce a two-actor SDF graph to a PSG is polynomial in the size of the SDF graph and is bounded by $O(\log n + \log m)$.

Proof: This follows by Lame's theorem for showing that the Euclidean GCD algorithm runs in polynomial time. We repeat the proof here for completeness; it is taken from [24]. Suppose that $n > m > 0$, and there are $k \ge 1$ reductions to get the PSG. Then we show that $n > F_{k+2}$ and $m > F_{k+1}$, where F_k is the k^{th} Fibonacci number ($F_k = F_{k-1} + F_{k-2}, \forall k > 1, F_0 = 0, F_1 = 1$). This will imply that if $m \le F_{k+1}$, then there are fewer than k reductions to get the PSG. Since

$$F_k \approx \phi^k = \left(\frac{(1+\sqrt{5})}{2}\right)^k ,$$

the number of reductions is $O(\log\ m)$.

The proof is by induction on k. For the basis, let $k = 1$. Then $m > 1 = F_2$ since needing one reduction implies that m cannot be 1. Since $n > m$, we must have $n > 2 = F_3$. Now assume that it is true that if $k - 1$ reductions are required then $n > F_{k+1}$ and $m > F_k$. We will show that it holds for k reductions also. Since $k > 0$, we have $m > 1$, and the reduction process will produce a reduced graph with $n' = n \bmod m$ and $m' = m$. We will then make $k - 1$ additional reductions (the next reduction will result in a graph with $n'' = n'$, and $m'' = m' \bmod n'$ and so on). The inductive hypothesis implies that $m > F_{k+1}$ (since $m > n \bmod m$ after the first reduction), proving one part of the requirement, and that $n \bmod m > F_k$. Now, $m + n \bmod m = m + (n - \lfloor n/m \rfloor m) \le n$. Hence,

$$n \ge m + n \bmod m > F_{k+1} + F_k = F_{k+2},$$

as required.

We can also show that if $m = F_{k+1}$, then there are exactly $k - 1$ reductions. Indeed, if $n = F_4 = 3$ and $m = F_3 = 2$, $k + 1 = 3$ there is $k - 1 = 1$ reduction. For

$k \geq 2$, we have $F_{k+1} \, mod \, F_k = F_{k-1}$. Thus reducing the graph with $n = F_{k+1}, m = F_k$ results in a graph with $n' = F_{k-1}, m' = F_k$, which shows inductively that there will be $k-1$ reductions exactly. **QED**.

Thus, we can implement the minimum buffer schedule in a nested, hierarchical manner, where the number of subroutine calls is guaranteed to be polynomially bounded in the size of the original two-actor SDF graph.

3.3. Extension to Arbitrary SAS

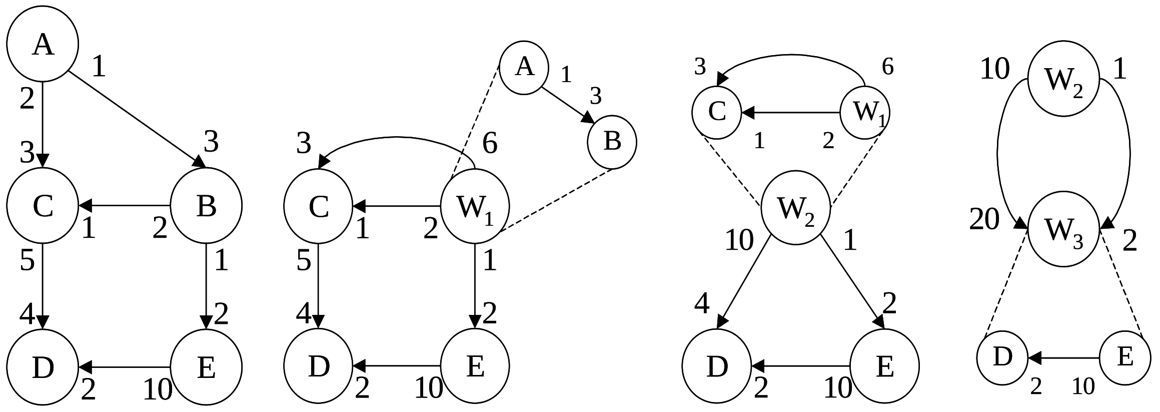
Any SAS S can be represented as an **R-schedule**,

$$S = (i_L S_L)(i_R S_R),$$

where S_L is the schedule for a "left" portion of the graph and S_R is the schedule for the corresponding "right" portion [7]. The schedules S_L, S_R can be recursively decomposed this way until we obtain schedules for two-actor graphs. In fact, the decomposition above can be represented as a clustered graph where the top level graph has two hierarchical actors and one or more edges between them. Each hierarchical actor in turn contains two-actor graphs with hierarchical actors until we reach two-actor graphs with non-hierarchical actors. Figure 3.3 shows an SDF graph, an SAS for it, and the resulting cluster hierarchy.

This suggests that the hierarchical implementation of the minimum buffer schedule can be applied naturally to an arbitrary SAS starting at the top-most level. In Figure 3.3, the graph in (d) is a PSG and has the SAS $(2W_2)W_3$. We then decompose the actors W_2 and W_3. For W_3, the graph is also a PSG, and has the schedule $E(5D)$. Similarly, the graph for W_2 is also a PSG with the schedule $W_1(2C)$. Finally, the graph for W_1 is also a PSG, and has the schedule $(3A)B$. Hence, in this example, no reductions are needed at any stage in the hierarchy at all, and the overall buffering requirement is $20 + 2 = 22$ for the graph in (d), 10 for W_3, 8 for W_2, and 3 for W_1, for a total requirement of 43. The VBMLB for this graph is 29. The reason that even the hierarchical decomposition does not yield the VBMLB is that the clustering process amplifies the produced/consumed parameters on edges, and inflates the VBMLB costs on those edges.

The extension to an arbitrary SDF graph, in other words, is to compute the VBMLB of the cluster hierarchy that underlies the given R-schedule. That is the goal the graph decomposition achieves and an algorithm overview is illustrated in Figure 3.4. In Figure 3.4, our approach is termed as **NEPS (NEsted Procedure Schedule)**. The VBMLB of the cluster hierarchy is calculated through summation over the VBMLB of all edges at each hierarchical level (e.g., W_1, W_2, W_3, and the top-most level comprising W_2 and W_3 in Figure 3.3). We denote this cost as the **VBMLB for a graph cluster hier-**



SAS: (2 ((3A) B) (2C)) (E (5D))

Figure 3.3 An SAS showing how an SDF graph can be decomposed into a series of two-actor subgraphs.

archy and for the example of Figure 3.3, the cluster hierarchy VBMLB is 43 as computed in the previous paragraph.

To obtain an R-schedule, DPPO is a useful algorithm to start with. As discussed in Chapter 2, DPPO is a dynamic programming approach to generating an SAS with minimal buffering cost. Because the original DPPO algorithm pertains to direct implementation in SAS form, the cost function in the dynamic programming approach is based on a buffering requirement calculation that assumes such implementation as an SAS. If, however, the SAS is to be processed using the decomposition techniques developed in this section, the VBMLB value for an edge e,

$$prd(e) + cns(e) - \gcd(prd(e), cns(e)),$$

is a more appropriate cost criterion for the dynamic programming formulation. This modified DPPO approach will evaluate a VBMLB-optimized R-schedule, which provides a hierarchical clustering suitable for our recursive graph decomposition.

Although we have shown that the number of decompositions required to reach a PSG is polynomial for a two-actor SDF graph, it is not obvious from this that the complexity of our recursive decomposition approach for arbitrary graphs is also polynomial. For

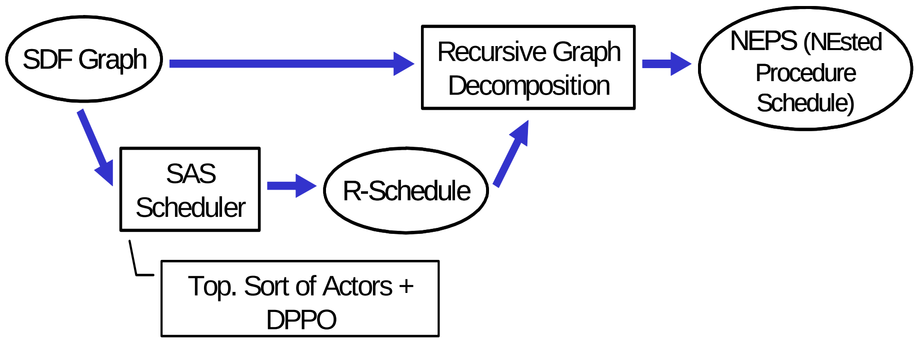

SDF Graph
Recursive Graph Decomposition
NEPS (NEsted Procedure Schedule)
SAS Scheduler
R-Schedule
Top. Sort of Actors + DPPO

Figure 3.4 An algorithm overview for arbitrary SDF graphs.

arbitrary graphs, the clustering process expands the produced/consumed numbers; in fact, these numbers can increase multiplicatively. Because of the nature of the logarithm operator, however, the multiplicatively-increased rates still keep the decomposition process of polynomial complexity in the size of the input SDF graph. First off, we have that the repetitions number for any actor v is $q(v) = O(P^{|E|})$, where $p = MAX_{e \in E}(prd(e), cns(e))$ and E is the set of edges in the SDF graph. If we cluster some set of actors $\{v_i, \ldots, v_j\}$ into a actor W, the produced parameter on each edge e_k leaving W [7] is increased by

$$\frac{q(v_k)}{\gcd(q(v_i), \ldots q(v_j))} prd(e_k) \le q(v_k) prd(e_k) .$$

Since the number of decompositions was $O(log(m))$, we see that if $m = q(v_k)prd(e_k)$, then

$$log(m) = log(q(v_k)) + log(prd(e_k)) = |E|\, log(P) + log(prd(e_k)),$$

and this is a polynomial function of the SDF graph.

Notice that we had to deal with multiple edges between actors in the above example. It is not immediately obvious whether there exists a schedule for a two-actor graph with multiple edges between the two actors that will simultaneously yield the VBMLB on each edge individually. We prove several results below that guarantee that there does exist such a schedule, and that a schedule that yields the VBMLB on any one edge yields the VBMLB on all edges simultaneously.

Consider the consistent two-actor graph shown in Figure 3.5. The repetitions vector satisfies the following equations:

$$q(W_1)p_i = q(W_2)c_i \qquad \forall i = 1, \ldots, k . \tag{3.2}$$

The fact that the graph is consistent means that (3.2) has a valid, non-zero solution.

Lemma 1. Suppose that $p_1 \le \ldots \le p_k$. Then $c_1 \le \ldots \le c_k$.

Proof: Suppose not. Suppose that for some i and j, we have $p_i \le p_j$ but $c_i > c_j$. Equation (3.2) implies that $c_i / p_i = c_j / p_j$. But $p_i \le p_j$ and $c_i > c_j$ implies $c_i / p_i > c_j / p_j$, contradicting Equation (3.2). **QED**.

Now consider the two graphs shown in Figure 3.6. Let these graphs have the same repetitions vector. Thus, we have

$$q(A)p_1 = q(B)c_1 \text{ and } q(A)p_2 = q(B)c_2. \tag{3.3}$$

Theorem 3.6. The two graphs in Figure 3.6 (I) and (II) have the same set of valid schedules.

Proof: Suppose not. Suppose there is a schedule for (I) that is not valid for (II). Let σ be the firing sequence $X_1 X_2 \ldots X_{q(A)+q(B)}$, where $X_i \in \{A, B\}$. Since σ is not valid for (II), there is some point at which a negative state would be reached in this firing sequence in

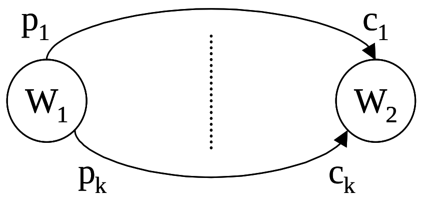


Figure 3.5 A two-actor SDF multi-graph.

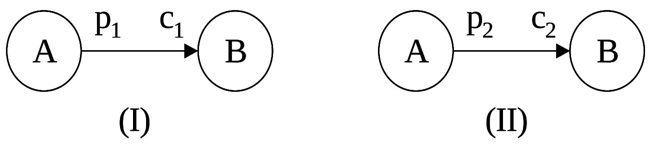


Figure 3.6 Two two-actor graphs with the same repetitions vector.

graph (II). By a negative state, we mean a state in which at least one buffer has had more tokens consumed from it than the number of tokens that have been produced into it. That is, after n_A, n_B firings of A and B respectively, we have $n_A p_2 - n_B c_2 < 0$ while $n_A p_1 - n_B c_1 \geq 0$. So,

$$0 \leq n_A p_1 - n_B c_1 < n_B \frac{c_2}{p_2} p_1 - n_B c_1 .$$

By (3.3), we have $c_1 / p_1 = c_2 / p_2$. Thus $n_B (c_2 / p_2) p_1 - n_B c_1 = 0$, giving a contradiction. **QED**.

Theorem 3.7. The schedule that yields the VBMLB for (I) also yields the same VBMLB for (II).

Proof: Let σ be the firing sequence $X_1 X_2 \ldots X_{q(A) + q(B)}$, where $X_i \in \{A, B\}$, that yields the VBMLB for (I). By Theorem 3.6, σ is valid for (II) also. Since σ is the VBMLB schedule for (I), at some point, after n_A^*, n_B^* firings of A and B respectively, we have

$$n_A^* p_1 - n_B^* c_1 = p_1 + c_1 - \gcd(p_1, c_1)$$

and for all other n_A and n_B in σ,

$$n_A p_1 - n_B c_1 \leq n_A^* p_1 - n_B^* c_1 . \tag{3.4}$$

We have that

$$n_A^* p_2 - n_B^* c_2 = n_A^* p_2 - n_B^* \frac{p_2}{p_1} c_1$$

$$= p_2 \left(n_A^* - \frac{c_1}{p_1} n_B^* \right)$$

$$= p_2 \left(n_A^* + \frac{p_1 + c_1 - \gcd(p_1, c_1) - n_A^* p_1}{p_1} \right)$$

$$= p_2\left(1 + \frac{c_1}{p_1} - \frac{\gcd(p_1, c_1)}{p_1}\right)$$

$$= p_2 + c_2 - \frac{\gcd(p_1, c_1)}{p_1}p_2$$

$$= p_2 + c_2 - \gcd\left(p_1\frac{p_2}{p_1}, \frac{c_1p_2}{p_1}\right)$$

$$= p_2 + c_2 - \gcd(p_2, c_2)$$

By Equation (3.4), we have

$$n_A \le n_B\frac{c_1}{p_1} + 1 + \frac{c_1}{p_1} - \frac{\gcd(p_1, c_1)}{p_1}.$$

Thus,

$$n_Ap_2 - n_Bc_2 \le \left(n_B\frac{c_1}{p_1} + 1 + \frac{c_1}{p_1} - \frac{\gcd(p_1, c_1)}{p_1}\right)p_2 - n_Bc_2 = p_2 + c_2 - \gcd(p_2, c_2).$$

Hence, this shows that σ yields the VBMLB for (II) also. **QED**.

Theorem 3.8. For the graph in Figure 3.5, there is a schedule that yields the VBMLB on every edge simultaneously.

Proof: Follows from the above results.

3.4. CD-DAT Example

Given the CD-DAT example in Figure 3.7, the DPPO algorithm returns the SAS $(7(7(3AB)(2C))(4D))(32E(5F))$. This schedule can be decomposed into two-actor clustered graphs as shown in Figure 3.8 . The complete procedure call sequence is shown in Figure 3.9, where each vertex represents a subroutine, and the edges represent caller-

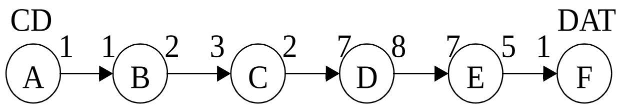


Figure 3.7 A CD-DAT sample rate converter example.

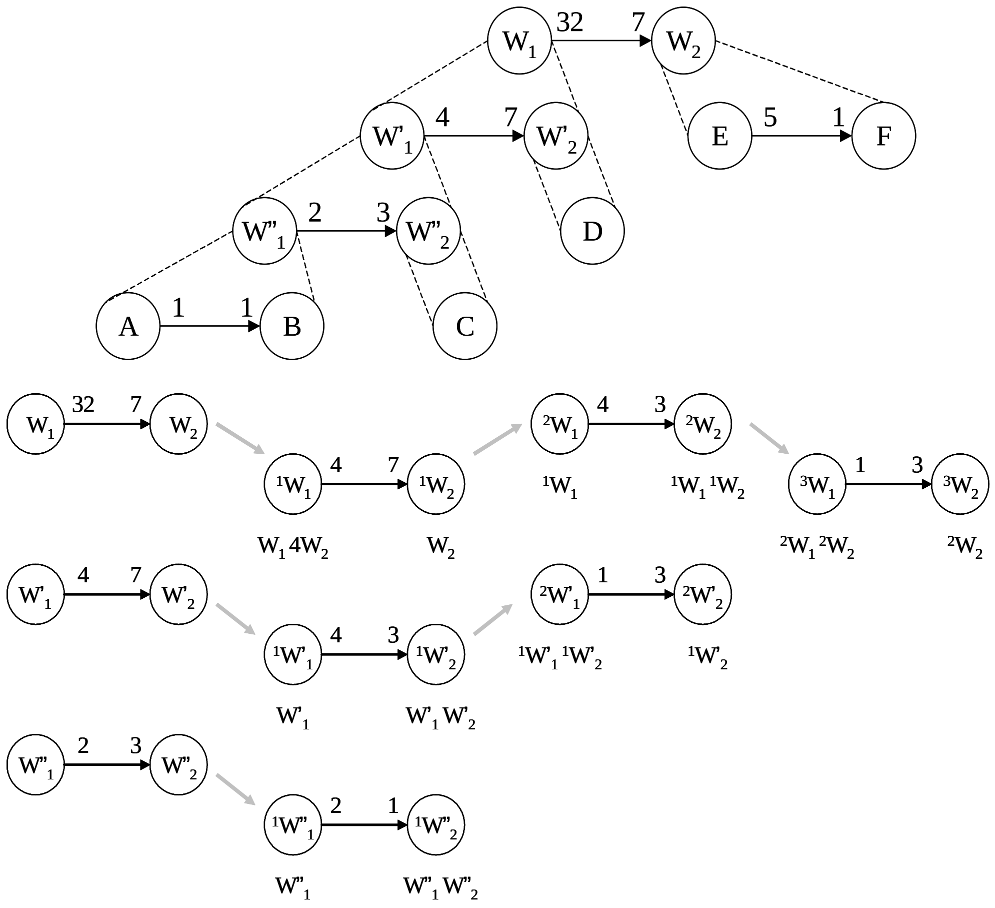


Figure 3.8 The recursive decomposition of the CD-DAT graph.

callee relationships. The generated C style code is shown in Figure 3.10 with 11 procedure calls. The number of required procedure calls is the **code cost** of the NEPS, while the **buffer cost** of the NEPS is the amount of buffer space required.

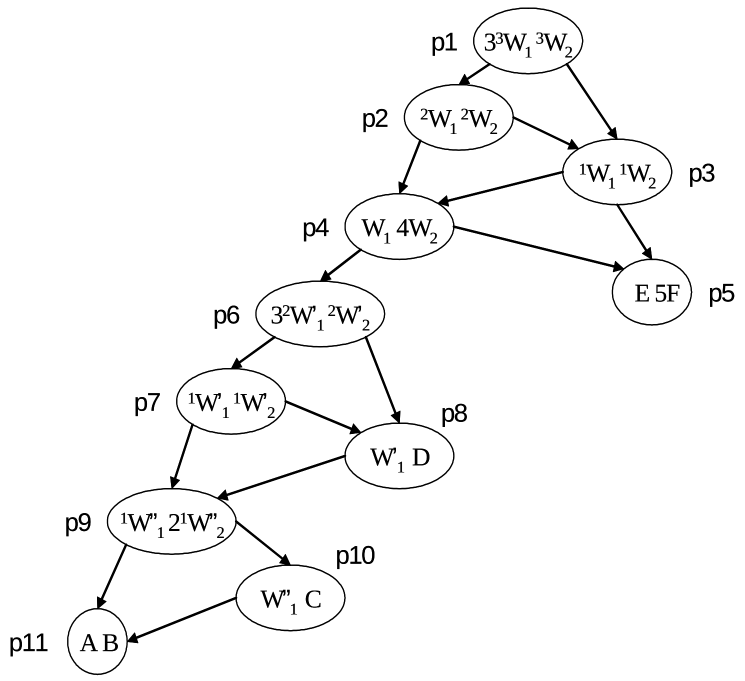


Figure 3.9 Procedure call sequence for the CD-DAT example.

```
p1() {
  for (int i=0; i<3; i++) {
    p2();
  }
  p3();
}

p2() {
  p4();
  p3();
}

p3() {
  p4();
  p5();
}

p4() {
  p6();
  for (int i=0; i<4; i++) {
    p5();
  }
}
```

```
p5() {
  inline of actor E;
  for (int i=0; i<5; i++) {
    inline of actor F;
  }
}

p6() {
  for (int i=0; i<3; i++) {
    p7();
  }
  p8();
}

p7() {
  p9();
  p8();
}

p8() {
  p9();
  inline of actor D;
}
```

```
p9() {
  p11();
  for (int i=0; i<2; i++) {
    p10();
  }
}

p10() {
  p11();
  inline of actor C;
}

p11() {
  inline of actor A;
  inline of actor B;
}
```

Figure 3.10 Generated C code for the CD-DAT example.

Definition 3.1. The number of procedure calls required for an NEPS implementation is the number of nodes in the procedure call dependence graph as in Figure 3.9.

The total buffer memory requirement of the CD-DAT is

$$(32 + 7 - 1) + (4 + 7 - 1) + (2 + 3 - 1) + 5 + 1 = 58 .$$

This is a 72% improvement over the best requirement of 205 obtained for a strictly inlined implementation of a SAS. The requirement of 205 is obtained by using a buffer merging technique [36].

3.5. Extension to Graphs with cycles

To deal with cycles, the **loose interdependence algorithm framework (LIAF)** [7] is a systematic approach to collaborate with NEPS. LIAF is basically a recursive strategy in deriving and scheduling (generating SAS) new graph topologies where nodes may be the original nodes (called **atomic** nodes) or subgraphs of the previous graph (called **composite** nodes). Composite nodes are solved in the same fashion until atomic nodes are reached or such recursion is infeasible. In LIAF, derivation of new graph topologies begins with breaking data dependences of certain edges as if they were removed, particularly those on cycles and with sufficient delays. Such delay sufficiency is determined by the requirement to make a cycle deadlock free and therefore schedule-able. When all the dependences bearing enough delays are broken, **strongly connected component (SCC)** decomposition is then applied to the graph. A new acyclic graph can be formed from newly created composite nodes in place of the SCCs and acyclic schedules can be hence efficiently computed. **Loose interdependence** refers to the situation that an acyclic structure can be transformed from a cycle in this way. It is **tight interdependence**, otherwise.

We call a graph a **tightly interdependent component** if tight interdependence exists in that graph. A summary of LIAF is sketched in Figure 3.11.

An example of LIAF scheduling is given in Figure 3.12. The SDF graph G in (a) is strongly connected and has three simple cycles. According to LIAF, the criterion to break a data dependence is

$$delay(e) \geq prd_G(e) \cdot q_G(src(e)), \tag{3.5}$$

where G is the associated graph containing e. The rationale behind the delay sufficiency is that $snk(e)$ can always obtain enough tokens to consume in an SAS scheduling and the associated data dependence does not impose any restriction on the firing sequence of $src(e)$ and $snk(e)$ anymore. Therefore, the dependence of B from C in the original graph G, C from E in SCC_Y can be broken because of

$$12 = delay(CB) \geq prd_G(CB) \cdot q_G(C) = 1 \cdot 12.$$

After dependence breaking, SCC decomposition is performed and non-trivial SCCs (SCC_X and SCC_Y) can be replaced with newly created composite nodes (X and Y). The

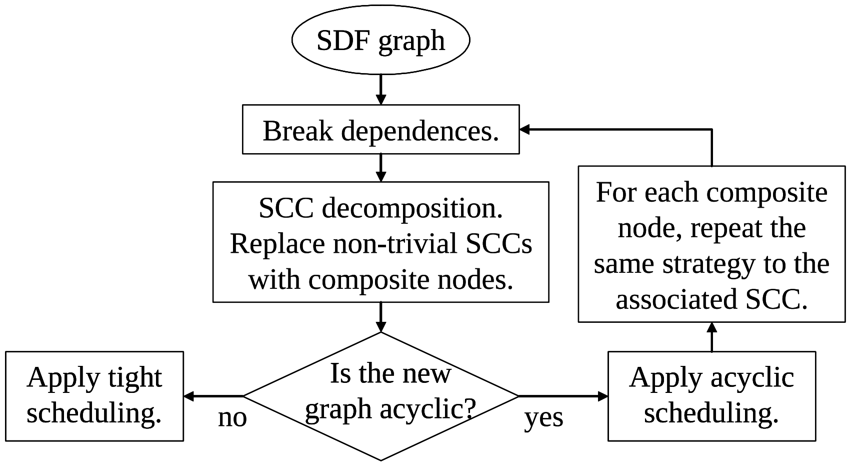

SDF graph
Break dependences.
SCC decomposition. Replace non-trivial SCCs with composite nodes.
For each composite node, repeat the same strategy to the associated SCC.
Is the new graph acyclic?
no
yes
Apply tight scheduling.
Apply acyclic scheduling.

Figure 3.11 Flow chart of LIAF approach.

new graph G' in (c) is then acyclic and the SAS is $(2X)(3Y)$. To complete the synthesis, each SCC is treated as an independent graph so that data dependence breaking, SCC decomposition, and acyclic scheduling can be applied recursively until stop condition is reached. In (e), the dependence of C from E can be broken because of

$$12 = delay(EC) \geq prd_{SCC_Y}(CB) \cdot q_{SCC_Y}(C) = 4 \cdot 3 .$$

Once again, acyclic structure is encountered in (f) and we can obtain the SAS $(2(2C)D)(3E)$ through the combination of APGAN and DPPO, which is efficient in deriving minimum buffer cost SAS. Unfortunately, the graph in (d) (also SCC_X) does not satisfy Equation (3.5),

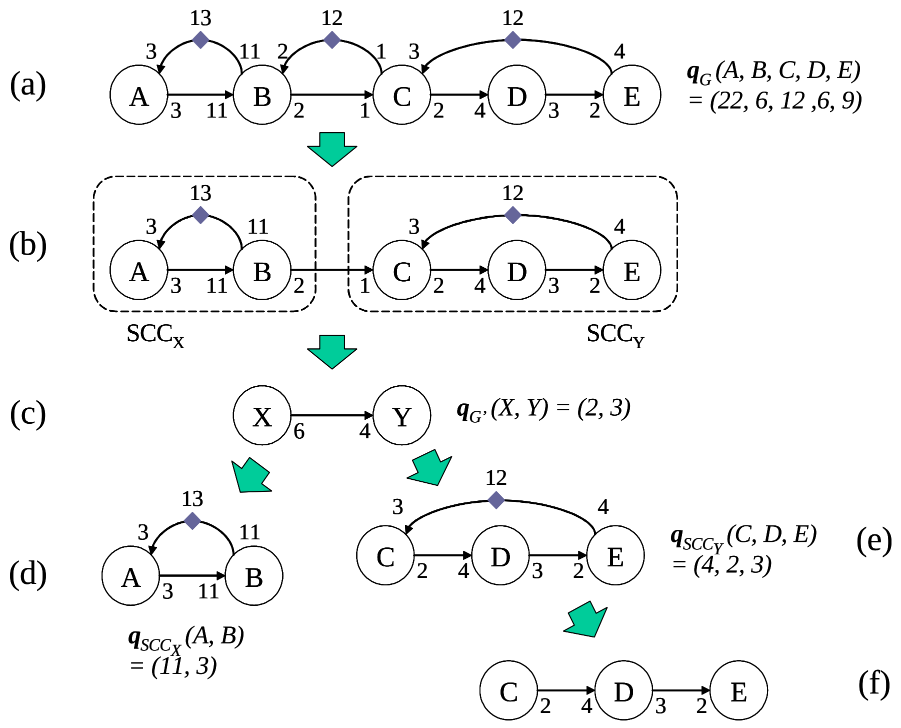

(a)
13
12
12
3
11
2
1
3
4
A
B
C
D
E
3
11
2
1
2
4
3
2
$q_G(A, B, C, D, E) = (22, 6, 12, 6, 9)$
(b)
SCC_X
SCC_Y
(c)
X
Y
6
4
$q_{G'}(X, Y) = (2, 3)$
(d)
$q_{SCC_X}(A, B) = (11, 3)$
(e)
$q_{SCC_Y}(C, D, E) = (4, 2, 3)$
(f)

Figure 3.12 An example demonstrating LIAF. Repetitions vectors are also given next to the associated graphs. The graph of (d) has tight interdependence.

$$13 = delay(BA) < (prd_{SCC_X}(B) \cdot q_{SCC_X}(B)) = 11 \cdot 3 = 33,$$

and is thus a tightly interdependent component. Since there is no efficient tight interdependence SAS scheduling so far, the LIAF will give us $(2X)(3(2(2C)D)(3E))$ with X unresolved.

To handle tight interdependence, strategies other than SAS scheduling are needed. One effective approach is the buffer cost minimization heuristic of [7], which gives us a **multiple appearance schedule (MAS)** $(AAAABAAAABAAAB)$ for SCC_X. The MAS can be compressed by CDDPO (more discussions will be given in Section 3.6.1) and we obtain $(2(4A)B)((3A)B)$, which is still a MAS for this case. Inlined implementation for MAS is costly due to multiple copies of code and procedural implementation is favored. Two ways of synthesizing SCC_X in procedures are presented in Figure 3.13: (a) is the implementation of $(2(4A)B)((3A)B)$ and (b) is our NEPS approach. Compared to a total of 5 procedures required in (a), our NEPS needs only 4 procedures while achieving the same buffer cost minimum of 13.

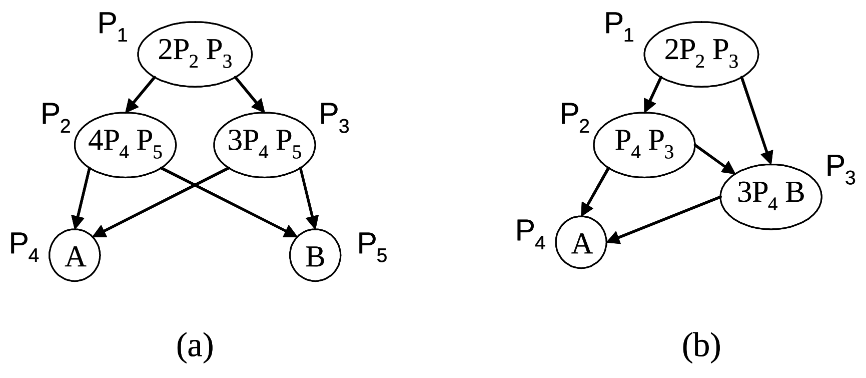


Figure 3.13 Two procedural implementations of the tight interdependent SCC in Figure 3.12: (a) minimum buffer cost heuristic plus CDDPO compression; (b) our NEPS approach.

The example shown in Figure 3.12(d) suggests less delays required by NEPS to break the dependence of A from B. To break the dependence, a minimum of 33 delays are required by LIAF according to Equation (3.5). In contrast, our NEPS demands a much lower minimum of 13 delays only. Further more, it is observed that $13 = 11 + 3 - gcd(11, 3)$. In other words, the NEPS delays requirement is equivalent to the VBMLB in this example. However, the truth of the lower delays requirement in general cases needs careful verification and is therefore left as future work.

In summary, NEPS fits into the LIAF framework neatly, and enables superior scheduling to be done for arbitrary SDF graphs. While tightly-interdependent components have been problematic for previous techniques, NEPS provides a way of scheduling these more efficiently as well.

3.6. Experimental Results

Our optimization algorithm is particularly beneficial to a certain class of applications. The statement of Theorem 3.3 tells us that no reduction is needed for edges with production and consumption rates that are multiples of one another. We call such edges **uniform edges**. Precisely, if an edge e has production and consumption rates m and n, respectively, then e is uniform if either $m|n$ or $n|m$. Our proposed strategy can improve buffering cost for **non-uniform** edges and generate the same buffering cost as existing SAS techniques for uniform edges.

We define two metrics for measuring this form of uniformity for a given SDF graph $G = (V, E)$ and an associated R-schedule S. For this purpose, we denote E_c as the set of edges in the cluster hierarchy associated with S. Thus, $|E_c| = |E|$ since every $e \in E$ has a corresponding edge in one of the clustered two-actor subgraphs associated with S. The set E_c can be partitioned into two sets: the uniform edge set E_u, which consists of the uniform edges, and the non-uniform edge set E_{nu}, which consists of the remaining edges.

Definition 3.2. Uniformity based on edge count (UEC):

$$U_1(G, S) = \frac{|E_u|}{|E|}.$$

Definition 3.3. Uniformity based on buffer cost (UBC):

$$U_2(G, S) = \frac{\sum_{e \in E_u} b(e)}{\sum_{e \in E} b(e)},$$

where $b(e)$ is the buffer cost on edge e for the given graph and schedule.

Our procedural implementation technique produces no improvement in buffering cost when uniformity is 100% (note that 100% uniformity for Definition 3.2 is equivalent to 100% uniformity for Definition 3.3). This is because if uniformity is 100%, then the two-actor graphs in the cluster hierarchy do not require any decomposition to achieve their associated VBMLB values.

We examined several SDF applications that exhibit uniformity values below 100%, and the results are listed in Figure 3.14. The first three columns give the benchmark names and graph sizes. Uniformity is measured by the proposed metrics and is listed in the fourth and fifth columns. The R-schedule in the uniformity computation is generated by the combination of APGAN and DPPO [7]. The last column is the **buffer cost ratio** of our procedural implementation over an R-schedule calculated by the combination of APGAN and DPPO. A lower ratio means that our procedural implementation consumes less buffer cost.

The first five *qmf* benchmarks are multirate filter bank systems with different depths and low-pass and high-pass components. Following those are three sample rate converters: *cd2dat*, *cd2dat2*, and *dat2cd*. The function of *cd2dat2* is equivalent to *cd2dat* except for an alternative breakdown into multiple stages. A two-channel non-uniform filter bank with depth of two is given in *filtBankNu*. The last benchmark *cdma2k_rev* is a CDMA example of a reverse link using HPSK modulation and demodulation under SR3.

Uniformity and buffer cost ratio are roughly in a linear relationship in Figure 3.14. To further explore this relationship between buffer cost ratio and uniformity, we experimented with a large set of randomly-generated SDF graphs, and the results are illustrated in Figure 3.15. Both charts in the figure exhibit an approximately proportional relationship between uniformity and buffer cost ratio. The lower the uniformity, the lower the buffer cost ratio.

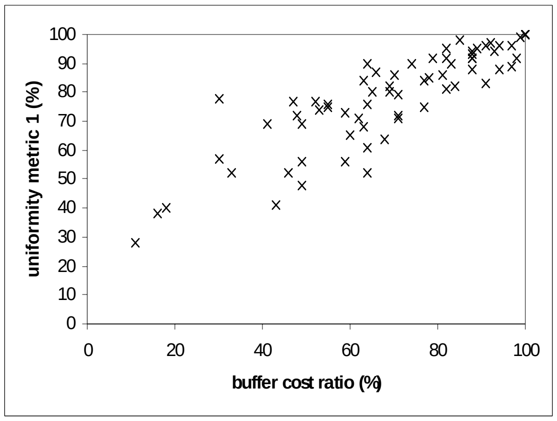
To better understand the overheads of execution time and code size for procedural over inlined implementation, we examined the cd2dat and dat2cd examples in more detail.

	actors count	edges count	*UEC* (%)	*UBC* (%)	buffer cost ratio (%)
aqmf235_2d	20	22	90	88	88
aqmf235_3d	44	50	76	70	76
aqmf23_2d	20	22	90	86	93
aqmf23_3d	44	50	80	70	87
nqmf23	32	35	82	84	86
cd2dat	8	7	42	4	9
cd2dat2	6	5	40	10	21
dat2cd	5	4	50	17	14
filtBankNu	26	28	82	83	90
cdma2k_rev	143	157	96	77	90

Figure 3.14 Experimental results for real applications.

In the experiment for *cd2dat*, we obtained 0.75% and 10.85% execution time and code size overheads, respectively, compared to inlined implementations of the schedules returned by APGAN and GDPPO. In the experiment for *dat2cd*, the overheads observed were 1.26% and 9.45% respectively. In these experiments, we used the Code Composer Studio by Texas Instruments for the *TMS320C67x* series processors. In general the overheads depend heavily on the granularity of the actors. In the applications of Figure 3.14, the actors are mostly of coarse granularity. However, in the presence of many fine-grained (low complexity) actors, the relative overheads are likely to increase; and for such applications, the procedural implementation approach proposed in this chapter is less favorable, unless buffer memory constraints are especially severe.

Employing VBMLB as the cost criterion for GDPPO gives better results than employing the BMLB as the cost criterion for GDPPO. Previously, the **BMLB (Buffer Memory Lower Bound)** has been used as the cost criteria for GDPPO in deriving buffer optimal SAS [7]. In the new formulation here, as suggested in Section 3.3, we use the



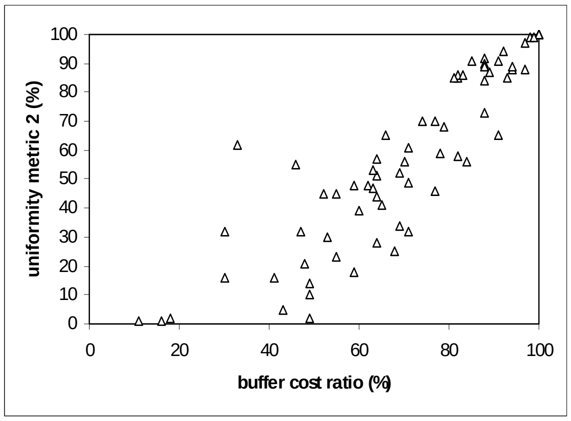


Figure 3.15 Relationship between uniformity and buffer cost ratio for random graphs.

VBMLB as the cost criterion for GDPPO in calculating R-schedules for NEPS implementations. Though the R-schedules generated by GDPPO for the CD-DAT example in Figure 3.7 are the same under either criterion, GDPPO with VBMLB as the cost criterion can result in smaller buffer costs as shown in Figure 3.16. We only show those benchmarks in Figure 3.16 for which there is a buffer cost reduction; the other examples all have the same cost under both criteria and hence no reduction. The buffer cost reduction caused by using VBMLB instead of BMLB is displayed in percentage as the bars' length. That is, the reduction is defined as

$$\frac{bufferCost(BMLB) - bufferCost(VBMLB)}{bufferCost(BMLB)}$$

where $bufferCost(X)$ means the buffer cost of NEPS taking DPPO results as the input R-schedule with the associated cost criterion X. The figure shows that there is a definite advantage in using VBMLB as the cost criterion for GDPPO under NEPS implementations.

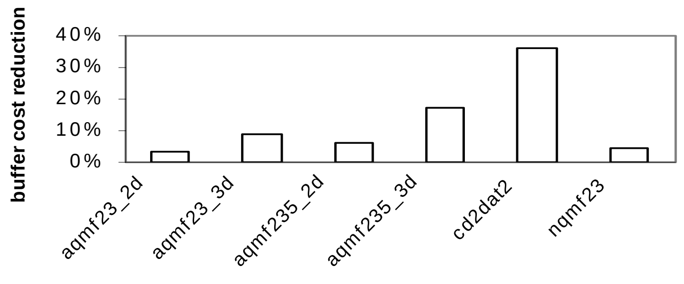


Figure 3.16 Buffer cost reduction due to VBMLB as the GDPPO cost criterion compared to BMLB criterion.

3.6.1. Comparison with Minimum Buffer Cost Schedules

So far, we have compared the NEPS implementation to the SAS implementation, and shown that while the SAS has minimum overall code cost, it has considerably higher buffer cost compared to NEPS. Here we compare NEPS with the converse of SAS: schedules of minimum overall buffer cost that necessarily have poor code costs.

An arbitrary, **raw** schedule for an SDF graph is an admissible actor firing sequence that does not have any form of schedule or code compression. For instance, organizing loops in the schedule is a form of schedule compression. For a raw schedule, the code cost is the sum of all actor firings counts (actors' repetitions). The repetitions of an actor may be exponential in the size of an SDF graph; hence, a raw schedule has exponential length in the size of the SDF graph. Whether the raw schedule is inlined or implemented via procedure calls does not matter; it will have very poor code cost. For example, the CD-DAT system in Figure 3.7 has 612 actor firings in a valid raw schedule. Even if each occurrence of a firing is replaced with a procedure call, the implementation will still demand large code space due to the 612 procedure calls required.

All known heuristics for generating minimum buffer schedules for SDF graphs generate them in raw form; see for example [7] and [14]. **CDPPO** [8] is an adaptation of GDPPO that minimizes schedule size for an arbitrary sequence of actor firings. Like GDPPO, CDPPO uses dynamic programming to build nested loops to compress repeated firings optimally. To demonstrate the effect of CDPPO, let us take the simple example in Figure 3.3(a). The minimum buffer schedule is given by

$$(AAABCAAABEDCDCDCDD)$$

CDPPO will organize loops in this schedule, thereby compressing it:

$$((3A)(B(C((3A)(B(E((3DC)(2D))))))))\tag{3.6}$$

Since CDPPO does not change the firing sequence of the schedule, the buffer cost of the schedule is the same as before compression. Both of the above schedules result in a buffer cost of 35, compared to 43 given by SAS and NEPS.

Now we want to investigate how NEPS implementations compare to optimally looped minimum buffer schedules returned by CDPPO applied to raw minimum buffer schedules. The loop hierarchy of the looped schedule suggests a natural way for a procedural implementation where the loop nests become procedures recursively. The procedure call sequence of Equation (3.6) is shown in Figure 3.17(a). As can be seen in the figure, sub-schedules are implemented in procedures. Actors A, B, C, and D are implemented

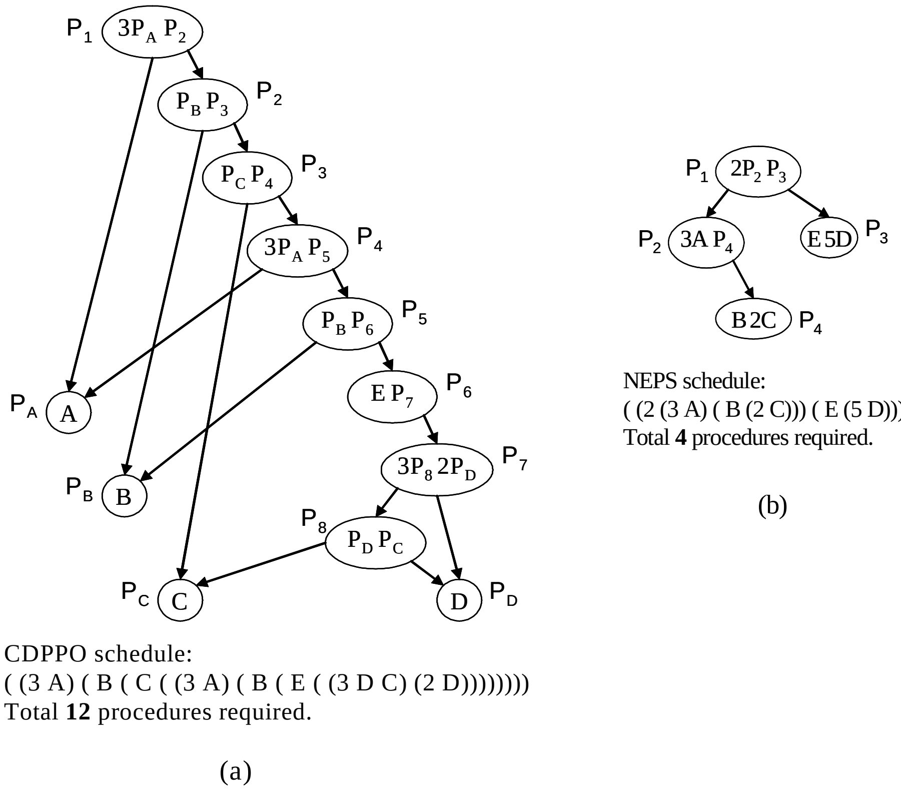


Figure 3.17 Procedure call sequence for CDPPO and NEPS schedules of the SDF graph in Figure 3.3(a).

as procedures as well because they appear multiple times in the schedule. Actor E is not implemented as a procedure but is inlined instead, in procedure P_6. This procedural implementation is arguably the most code efficient realization of the looped minimum buffer schedule. The code cost of this implementation is the number of nodes in the procedure call graph

Hence, the minimum buffer CDPPO schedule of Equation (3.6) requires a total of 12 procedure calls, while our NEPS implementation requires 4 procedure calls as shown in Figure 3.17(b). The buffer cost overhead of NEPS over the minimum buffer CDPPO schedule is 23%, but there is a 67% reduction of procedure count of NEPS over that of the minimum buffer CDPPO schedule.

Experiments on random graphs show that this advantage that NEPS has over minimum buffer CDPPO schedules (of having a small buffer cost overhead in return for a large code size savings), holds in general. The results are summarized in Figure 3.18. In the figure, buffer cost overhead is defined as

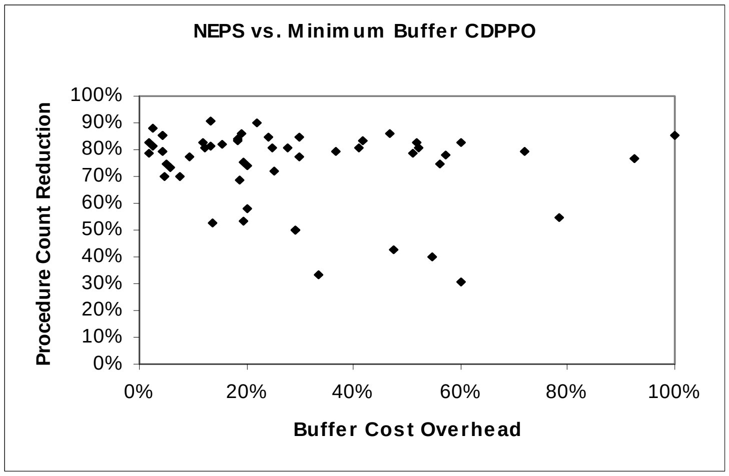


Figure 3.18 Comparison of NEPS with the minimum buffer CDPPO.

$$\frac{bufferCost(NEPS) - bufferCost(CDPPO)}{bufferCost(CDPPO)},$$

where $bufferCost(X)$ is the buffer cost computed by X and X is either NEPS or minimum buffer CDPPO schedule. Procedure count reduction is formulated as

$$\frac{procCount(CDPPO) - procCount(NEPS)}{procCount(CDPPO)},$$

where $procCount(X)$ is the procedure count of the schedule X. In Figure 3.18, most of the points fall in the area of less than 30% buffer cost overhead and 70% to 90% procedure count reduction. Thus, with minor buffer cost overhead, NEPS can result in many fewer procedure calls and therefore, a much smaller code space requirement.

Because the problem of computing minimum buffer schedules is NP-complete even for homogenous SDF graphs [7], we have to rely on heuristics for generating these schedules in general (if we want efficiency) [1][7][14]. While some heuristics might produce optimal results for some subclasses of graphs, like trees, they will be necessarily sub-optimal in general. So far, there does not appear to be a rigorous experimental evaluation of the quality of these minimum buffer heuristics on general graphs. In our experiment with random graphs, we observed that NEPS actually beats the minimum buffer heuristic of [7] in many cases. While this is yet another advantage for NEPS, in that it is apparently better than even a heuristic designed to purely return minimum buffer schedules, we believe that further evaluation is needed of minimum buffer heuristics before a more definitive claim is made of the ability of NEPS to actually function as a minimum buffer heuristic. Hence in our comparison above, we have omitted the cases where NEPS outperformed the minimum buffer heuristic, and only compared it to cases where it was worse. For future work we would like to do a thorough comparison of the heuristics of [1][7][14] against NEPS and see where NEPS fits in.

A major advantage of NEPS is the polynomial-time computational complexity. Though CDPPO is $O(n^4)$ where n is the number of actor firings [8], n is potentially exponential in the size of the SDF graph. Thus CDPPO computation is inefficient for large graphs while NEPS remains efficient.

Some more experiments are conducted in order to compare the performance of minimum buffer CDPPO schedules with NEPS and SAS. The results are depicted in Figure 3.19. In the bar chart where buffer costs are compared, buffer costs are normalized by the cost of SAS (computed by APGAN and GDPPO) because SAS has the worst buffer cost. That is, each bar has length

$$\frac{bufferCost(X)}{bufferCost(SAS)}$$

in percentage where $bufferCost(X)$ is the buffer cost of schedule X for the particular benchmark. Similarly, procedure counts are also normalized by the largest one, which are produced by the minimum buffer CDPPO schedule. Each bar in the procedure count comparison chart has length

$$\frac{procCount(X)}{procCount(MinBuff)}$$

in percentage where $procCount(X)$ is the number of procedure calls required in the procedural implementation of schedule X for the particular benchmark. The buffer cost chart in Figure 3.19 shows that NEPS is somewhere in between minimum buffer CDPPO and SAS, many times much closer to minimum buffer CDPPO than SAS. The procedure cost chart shows that NEPS is again between SAS and minimum buffer CDPPO, but much closer to SAS in all instances. Thus, NEPS gives us a nice trade-off between code size and buffer size, and gives implementations where the overhead over the best possible for both metrics is low. In the space of optimizing for two opposing criteria, namely buffer size and code size, the NEPS implementation is a useful and efficient Pareto point.

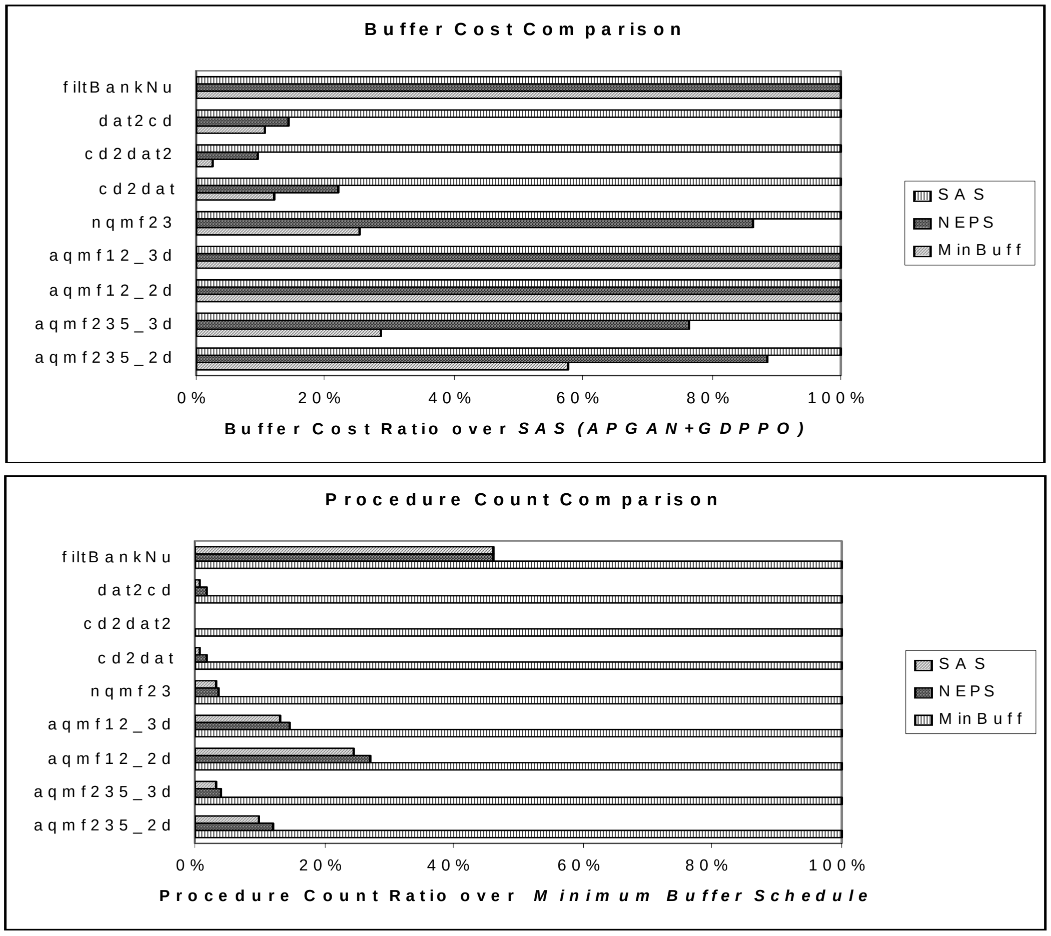


Figure 3.19 Comparison of minimum buffer schedule, NEPS, and SAS (APGAN+GDPPO).

Chapter 4. Block Processing Optimization

DSP applications involve processing long streams of input data. It is important to take into account this form of processing when implementing embedded software for DSP systems. Task-level **vectorization**, or **block processing**, is a useful dataflow graph transformation that can significantly improve execution performance by allowing subsequences of data items to be processed through individual task invocations. In this way, several benefits can be obtained, including reduced context switch overhead, increased memory locality, improved utilization of processor pipelines, and use of more efficient DSP-oriented addressing modes. On the other hand, block processing generally results in increased memory requirements since it effectively increases the sizes of the input and output values associated with processing tasks.

In this chapter, we investigate the memory-performance trade-off associated with block processing. We develop novel block processing algorithms that take carefully take into account memory constraints to achieve efficient block processing configurations within given memory space limitations. Our experimental results indicate that these methods derive optimal memory-constrained block processing solutions most of the time. We demonstrate the advantages of our block processing techniques on practical kernel functions and applications in the DSP domain. A preliminary summary of part of this chapter is published in [26]

4.1. Introduction

Indefinite- or unbounded-length streams of input data characterize most applications in the DSP and communications domains. As the complexity of DSP applications grows rapidly, great demands are placed on embedded processors to perform more and more intensive computations on these data streams. The multi-dimensional requirements that are emerging in commercial DSP products— including requirements on cost, time-to-market, size, power consumption, latency, and throughput — further increase the challenge of DSP software implementation.

Because of the intensive, stream-oriented computational structure of DSP applications, performance optimization for DSP software is a widely researched area. Examples of methods in this area include reducing context switching costs, replacing costly instructions that use absolute addressing, exploiting specialized hardware units or features, and using various other DSP-oriented compiler optimization techniques (e.g., see [32]).

Task-level **vectorization** or **block processing** is one general method for improving DSP software performance in a variety of ways. In this context, block processing refers to the ability of a task to process groups of input data, rather than individual scalar data items, on each task activation. Such a task is typically implemented in terms of a block processing parameter that indicates the size of the each input block that is to be processed. This way, the task programmer can optimize the internal implementation of the task through awareness of its block processing capability, and a task-level design tool can optimize the way block processing is applied to each task and coordinated across tasks for more global optimization.

In this chapter, we explore such global block processing optimization for dataflow-based design tools. More specifically, we examine the trade-off between block processing implementation and data memory requirements. Understanding this trade-off useful in memory-constrained software and design space exploration. Theoretical analysis and algorithms are proposed to efficiently achieve streamlined block processing configurations given constraints on data memory requirements. In addition, our approach is based on hierarchical loop construction such that code size is always minimized (i.e., duplicate copies of actor code blocks are not required). Experimental results show that our approach often computes optimal solutions. At the same time, our approach is practical for incorporation into software synthesis tools due its low polynomial run-time complexity.

4.2. Related Work

To strengthen the motivation for block processing, it has been shown that block processing improves regularity and thus reduces the effort in address calculation and context switching [20]. Block processing also facilitates efficient utilization of pipelines for vector-based algorithms, which are common in DSP applications [10].

Task-level, block processing optimization for DSP was first explored by Ritz et al. [46]. In this approach, a dataflow graph is hierarchically decomposed based on analysis of fundamental cycles. The decomposition is performed carefully to avoid deadlock and maximize the degree of block processing. While the work jointly optimizes block processing and code size, it does not consider data memory cost. Another limitation of this approach is its high complexity, which results from exhaustive search analysis of fundamental cycles.

Joint optimization of block processing, data memory minimization, and code size minimization is examined in [45]. Unlike the approach of [46], the work of [45] employs memory space sharing to minimize data memory requirement. However, again the techniques proposed are not of polynomial complexity, and therefore they may be unsuitable for large designs or during design space exploration, where one may wish to rapidly explore many different design configurations.

In both the methods of [45] and [46], the optimization problem is formulated without user-specified data memory constraints. Furthermore, although, overall memory sharing cost is minimized in [45], memory costs for individual program variables are fixed to be the largest. In fact, however, many configurations of program variable sizes — in particular, the sizes of buffers that implement the edges in the dataflow graph — are usually possible under dataflow semantics. Choosing carefully within this space of buffer configurations leads to smaller memory requirements and provides flexibility in memory cost tuning.

In contrast to these related efforts, the optimization problem that we target in this chapter is formulated to take into account a user-defined data memory bound. This corresponds to the common practical scenario where one is trying to fit the implementation within a given amount of memory (e.g., the on-chip memory of a programmable digital signal processor). Also, by iterating through different memory bounds, trade-off curves between performance and memory cost can be generated for system synthesis and design space exploration.

In this chapter, in conjunction with block processing optimization, memory sizes of dataflow buffers are efficiently configured through novel algorithms that frequently achieve optimum solutions, while having low polynomial-time complexity.

Various other methods address the problem of minimizing context switching overhead when implementing dataflow graphs. For example, the **retiming** technique is often exercised on single-rate dataflow graphs. In the context of context switch optimization, retiming rearranges delays (initial values in the dataflow buffers) so they are better concentrated in isolated parts of the graph [28][59]. As another example, Hong, Potkonjak, and Papaefthymiou [22] investigate throughput-constrained optimization given heterogeneous context switching costs between task pairs. The approach is flexible in that overall execution time or other objectives (such as power dissipation) are jointly optimized under a fixed schedule length through appropriate sequencing of task execution.

These efforts target different objectives and operate on **single-rate** dataflow graphs, which are graphs in which all task execute at the same average rate. In contrast, the methods targeted in this chapter operate on **multirate** dataflow graphs, which are common in many signal processing applications, including wireless communications, and multimedia processing. Our work is motivated by the importance of multirate signal processing, and the much heavier demands on memory requirements that are imposed by multirate applications.

4.3. Background

Any SAS can be transformed to an **R-schedule**, $S = (i_L S_L)(i_R S_R)$, where S_L (S_R) is the left (right) subschedule of S [7]. The binary structure of an R-schedule can be represented efficiently as a binary tree, which is called a **schedule tree** or just a **tree** in our discussion [37]. In this tree representation, every node is associated with a loop iteration count, and every leaf node is additionally associated with an actor. A schedule tree example is illustrated in Figure 4.1.

The loop hierarchy of an R-schedule can easily be derived from a schedule tree, and vice-versa. Therefore, R-schedules and schedule trees are referred to interchangeably in our work.

To avoid confusion when referring to terms for schedule trees and SDF graphs, some conventions are introduced here. When referring to general graph structure terms, such as “node” and “edge,” we refer to these terms in the context of schedule trees, unless otherwise specified.

Given a node a, the left (right) child is denoted as $left(a)$ $(right(a))$. We define the **association operator**, denoted $\alpha()$, as follows: $\alpha(A) = a$ maps the SDF actor A to its associated schedule tree leaf node a. The loop iteration count associated with a leaf node a is denoted as $l(a)$. The tree (or subtree) rooted at node r is denoted as $tree(r)$, and the corresponding set of internal nodes (the set of nodes in $tree(r)$ that are not leaf nodes) is represented as $\lambda(r)$ or $\lambda(tree(r))$.

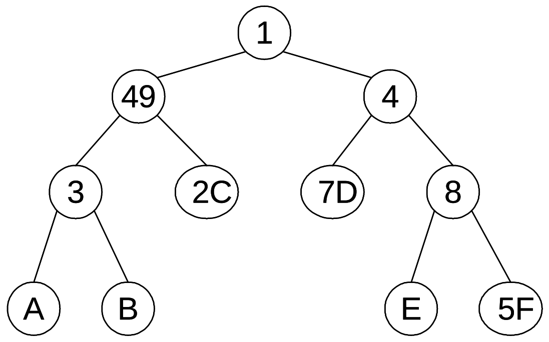


Figure 4.1 A schedule tree example corresponding to the SAS of Figure 2.3(c).

4.4. Block Processing Implementation

When a large volume of input data is to be processed iteratively by a task, a block processing implementation of the task can provide several advantages, including reduced context switch overhead, increased memory locality, improved utilization of processor pipelines, and use of more efficient DSP-oriented addressing modes. Motivation for block processing is elaborated on qualitatively in [46]. In this section, we add to this motivation with some concrete examples.

An example of integer addition is given in Figure 4.2 to illustrate the difference between conventional (scalar) and block processing implementation of an actor. From the perspective of the *main()* function, function *add_vector()* in Figure 4.2(b) has less procedure call overhead, fast addressing through auto-increment modes, and better locality for pipelined execution compared to *add_scalar()* in Figure 4.2(a).

To further illustrate the advantages of block processing, different configurations of *FIR*, *add*, *convolution*, *DCT-II* actors, which are important DSP kernel functions, are eval-

(a)

```
void add_scalar(int a, int b, int* sum) {
    *sum = a + b;
}
main() {
    int[] arrayA, arrayB, arraySum;
    for (int i=0; i<size; i++)
        add_scalar(arrayA[i], arrayB[i], &arraySum[i]);
}
```

(b)

```
void add_vector(int* a, int* b, int* sum, int size) {
    int* a2=a, b2=b, sum2=sum;
    for (int i=0; i<size; i++)
        *sum2++ = (*a2++) + (*b2++);
}
main() {
    int[] arrayA, arrayB, arraySum;
    add_vector(arrayA, arrayB, arraySum, size);
}
```

Figure 4.2 Integer addition (a) scalar version (b) vector version.

uated on the Texas Instruments TMS320C6700 processor. The results are summarized in Figure 4.3. Here, the left chart shows the number of execution cycles versus the number of actor invocations for both scalar and block processing implementations, where in the block processing case, all actor invocations are achieved with a single function invocation. The right chart gives the execution cycles reduced through application of block processing. Lines are drawn between dots to interpolate the underlying trend and do not represent real data.

By inspecting these charts, block processing is seen to achieve significant performance improvement, except when the actor invocation count (**vectorization degree**) is unity. In this case, one must pay for the overhead of block processing without being able to amortize the overhead over multiple actor invocations, so there is no improvement. Moreover, improvements are seen to saturate for sufficiently high vectorization degrees.

Charts of this form can provide application designers and synthesis tools helpful quantitative data for applying block processing during design space exploration.

4.5. Block Processing in Software Synthesis

To model block processing in SDF-based software synthesis, we convert successive actor invocations to inlined loops embedded within a procedure that represents an activation of the associated actor. Here, the number of loop iterations is equivalent to the number of successive actor invocations — that is, to the vectorization degree. Given an actor A, we represent the vectorization degree for A in a given block processing configuration as $vect(A)$. Thus, each time A is executed, it is executed through a unit of $vect(A)$ successive invocations. This unit is referred to as an **activation** of A.

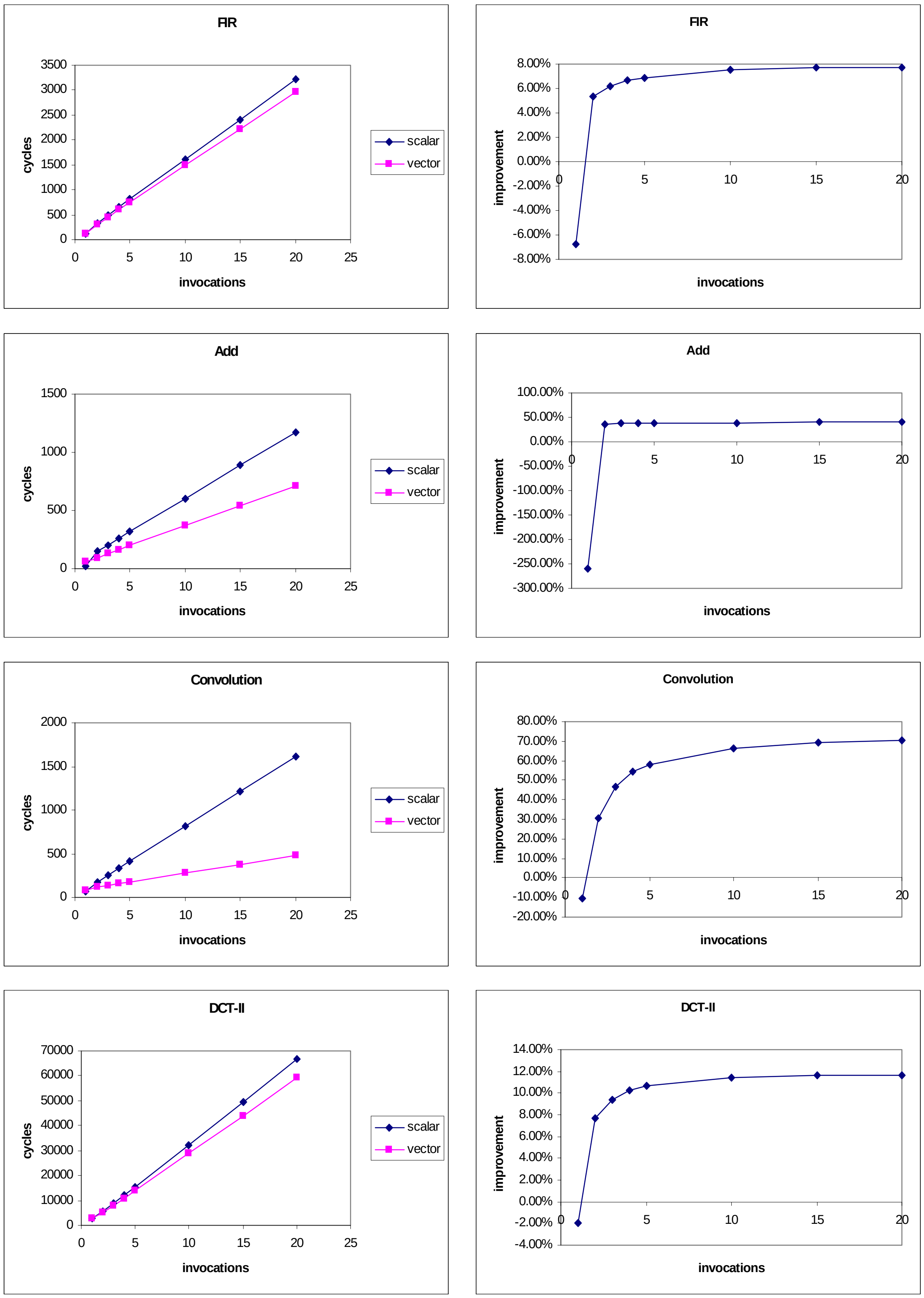


Figure 4.3 Performance comparison of vectorized and scalar implementation of FIR, add, convolution, and DCT-II operations.

Under block processing, the number of data values produced or consumed on each activation of A is $vect(A)$ times the number of data values produced or consumed per invocation of A (as represented by the $prd(e)$ and $cns(e)$ values on the edges that are incident to A).

Useful information pertaining to block processing can be derived from schedule trees. For this purpose, we denote $act(r)$ as the **activations number** for $tree(r)$ rooted at r. This quantity is defined as follows: $act(r) = 1$ if r is a leaf node, and otherwise, $act(r) = l(r)(act(left(r)) + act(right(r)))$.

If r is a leaf node and $\alpha(R) = r$, then $vect(R)$ successive invocations of actor R are equivalent to a single activation, and $act(r) = 1$. If r is an internal node, then based on the structure of SASs, an activation is necessary when $left(r)$ completes, and is followed by $right(r)$ at each of $l(r)$ iterations. An activation occurs also when $right(r)$ completes in one iteration and is followed by $left(r)$ in the next iteration. Therefore, we have $l(r)(act(left(r)) + act(right(r)))$ activations for $tree(r)$.

Given a valid schedule of an SDF graph S, there is a unique positive integer $J(S)$ such that S invokes each actor A exactly $J \times q(A)$ times, where q is the repetitions vector, as defined in Section 4.3. This positive integer is called the **blocking factor** of the schedule S. The blocking factor can be expressed as

$$J(S) = gcd(inv(A_1), inv(A_2), \ldots, inv(A_n)),$$

where gcd represents the **greatest common divisor** operation, $inv(A_i)$ represents the number of times that actor A_i is invoked in the schedule S, and n is the number of actors in the given SDF graph. We say that S is a **minimal** valid schedule if $J(S) = 1$, and a

graph iteration corresponds to the execution of a minimal valid schedule, or equivalently, the execution of $q(A)$ invocations of each actor A.

Increasing the blocking factor beyond the minimum required value of 1 can reduce the overall rate at which activations occur. For example, suppose that we have a minimal valid schedule $S_1 = ((2A)(3B))$ and another schedule $S_2 = ((8A)(12B))$, which has $J = 4$. Although both schedules result in 2 activations, the average rate of activations (in terms of activations per graph iteration) in schedule S_2 is one-fourth that of S_1. This is because S_2 operates through four times as many graph iterations as schedule S_1.

As motivated by this example, we define the **activation rate** of a schedule S as $rate(S) = act(S)/J(S)$, where $act(S)$ is the total number of actor activations in schedule S.

If S is represented as $tree(r)$, we have

$$rate(S) = rate(tree(r)) = act(r)/J(S).$$

The problem of optimizing block processing can then be cast as constructing a valid schedule that minimizes the activation rate. For example, S_2 has a lower activation rate ($rate(S_2) = 0.5$) than S_1 ($rate(S_1) = 1$), and S_2 is therefore more desirable under the minimum activation rate criterion.

The blocking factor is closely related to, but not equivalent to, the **unfolding factor**. Unfolding is a useful technique in DSP dataflow graph analysis [43], and both the unfolding factor and blocking factor are intended to help in investigating hardware/software configurations that encapsulate more than one graph iteration. While unfolding makes multiple copies of the original actors to enhance execution performance (in a manner analogous to loop unrolling), and generally allows executions of multiple graph iterations to overlap, use of blocking factors does not imply any duplication of actors, and usually does imply that each iteration of the given schedule will execute to completion before the next iteration begins.

4.6. Activation Rate Minimization with Unit Blocking Factors (ARMUB)

In this section we consider in detail the problem of minimizing the activation rate in a manner that takes into account user-defined constraints on buffer costs (data memory requirements). We restrict ourselves to unit blocking factor here because we are interested in memory-efficient block processing configurations, and increases in blocking factor generally increase memory requirements [7]. The resulting optimization problem, which we call *ARMUB* (Activation Rate Minimization with Unit Blocking factor), is the problem of rearranging the schedule tree of a minimal valid schedule such that the resulting schedule is valid and has a minimum number of total activations.

One more restriction in these formulations is that they assume that the input SDF graph is acyclic. Acyclic SDF graphs represent a broad and important class of DSP applications (e.g., see [7]). Furthermore, through the loose interdependence scheduling framework [7], which decomposes general SDF graphs into hierarchies of acyclic SDF graphs, the techniques of this chapter can be applied also to general SDF graphs.

The problem is formally described as follows. Assume that we are given an acyclic SDF graph G and a valid schedule S (and associated schedule tree $tree(r)$) for G such that $J(S) = 1$. Block processing is to be applied to G by re-arranging the loop iteration counts of tree nodes in the schedule tree for S. The optimization variables are the set

$\{l(x)\}$ of loop iteration counts in the leaf nodes of the rearranged schedule tree (recall that these loop iteration counts are equivalent to the vectorization degrees of the associated actors). The objective is to minimize the number of activations:

$$min(act(r)). \tag{4.1}$$

Changes to the loop iteration counts of tree nodes must obey the constraint that the overall numbers of actor invocations in the schedule is unchanged. In other words,

$$q(A) = \prod_{x \in path(a, r)} l(x) \quad \text{for each SDF actor } A, \tag{4.2}$$

where $\alpha(A) = a$, and $path(a, r)$ is the set of nodes that are traversed by the path from the leaf node a to the root node r. Intuitively, the equation says that no matter how the loop iteration counts are changed along the path, their product has to match the repetitions count of the associated actor.

In the ARMUB problem, we are also given a buffer cost constraint M (a positive integer), such that the total buffer cost in the rearranged schedule cannot exceed M. That is,

$$\sum_{e} buf(e) \le M, \tag{4.3}$$

where $buf(e)$ denotes the buffer size on SDF edge e.

The structure of R-schedules permits efficient computation of buffer costs.

Theorem 4.1. Given an acyclic SDF graph edge e, we have two leaf nodes a and b associated with the source and sink: $\alpha(src(e)) = a$ and $\alpha(snk(e)) = b$. Let p be the least common parent of a and b in the schedule tree. Then the buffer cost on e can be evaluated by the following expressions.

$$\begin{aligned} buf(e) &= prd(e)\left(\prod_{x \in path(a,\, left(p))} l(x)\right) \\ &= cns(e)\left(\prod_{y \in path(b,\, right(p))} l(y)\right). \end{aligned} \tag{4.4}$$

Proof: To determine the buffer cost, we need to figure out when data is produced and consumed on e. According to the semantics of single appearance schedules, at each of the $l(p)$ iterations of p, data produced on e by $left(p)$, which involves the firing of actor $src(e)$, will not be consumed until $right(p)$ (which involves the firing of actor $snk(e)$) starts to execute. In addition, based on the balance equations and the assumption of delayless edges, all of the data produced throughout a given iteration of $l(p)$ will be consumed without any data values remaining on the edge for the next iteration. This identical production and consumption pattern recurs at all $l(p)$ iterations in $tree(p)$ and any subtree where $tree(p)$ is contained. Therefore, the buffer cost is equivalent to the amount of data produced or consumed (Equation (4.4)). **QED.**

In summary, the ARMUB problem can be set up by casting Equations (4.1) through (4.4) into a **non-linear programming (NLP)** formulation, where the objective is given by (4.1), the variables are the loop iteration counts of the schedule tree nodes, and the constraints are given in (4.2), (4.3), and (4.4). Due to the intractability of NLP, efficient heuristics are desired to tackle the problem for practical use.

To determine an initial schedule to work on, we must consider the potential optimization conflicts between buffer cost and activations. While looped schedules that make extensive use of nested loops are promising in generating low buffer costs, activations minimization favors **flat schedules**, that is, schedules that do not employ nested loops.

We employ nested-loop SASs that have been constructed for low buffering costs as the initial schedules in our optimization process because flat schedules can easily be derived from any such schedule by the setting loop iteration counts of all internal nodes to one, while setting the loop iteration counts of leaf nodes according to the repetitions counts of the corresponding actors. Furthermore the construction of buffer-efficient nested loop schedules has been studied extensively, and the results of this previous work can be leveraged in our approach to memory-constrained block processing optimization. Specifically, the APGAN and GDPPO algorithms are employed in this work to compute buffer-efficient SASs as a starting point for our memory-constrained block processing optimization [7].

4.6.1. Loop Iteration Count Factor Propagation

As described earlier, activations values of leaf nodes are always equal to one and independent of their loop iteration counts. Hence, one approach to optimizing activations values is to enlarge the loop iteration counts of leaf nodes by absorbing the loop iteration counts of internal nodes. A similar approach is proposed in [46] to deal with cyclic SDF graphs with delays. The strategy in [46] is to extract integer factors out of a loop's iteration count and carefully propagate the factors to inner loops. Propagations are validated by checking that they do not introduce deadlock. However, as described in Section 4.2, the work of [46] does not consider buffer cost in the optimization process.

For acyclic SDF graphs, as we will discuss later, factors of loop iteration counts should be aggressively propagated straight to the inner most iterands to achieve effective block processing. Under memory constraints, such propagation should be balanced carefully against any increases in buffering costs.

Definition 4.1. FActor Propagation toward Leaf nodes (FAPL). Given $tree(r)$, the FAPL operation, when it can be applied, is to extract an integer factor $V > 1$ out of $l(r)$ and merge V into the loop iteration counts of all the leaf nodes in $tree(r)$. Formally, the new loop iteration count of r is $l(r)/V$, and for every leaf f in $tree(r)$ the new loop iteration count is $V \cdot l(f)$. Loop iteration counts of internal nodes remain unchanged. For notational convenience, a FAPL operation is represented as $\phi(r, V)$ for $tree(r)$ and factor V, or simply as ϕ when the context is known. We call r the **FAPL target internal node**, all leaf nodes in $tree(r)$ the **FAPL target leaf nodes**, and V the **FAPL factor**.

An example of FAPL is illustrated in Figure 4.4. FAPL reduces the number of activations and increases buffer costs.

Theorem 4.2. Given $tree(r)$ with $act(r)$, $\phi(r, V)$ reduces the activations of $tree(r)$ by a factor of V.

Proof: From the definitions of *act()* and FAPL, $act(left(r))$ and $act(right(r))$ are not affected (remain unchanged) from the operation $\phi(r, V)$. On the other hand, the loop iteration count of r turns into $l(r)/V$ as a result of $\phi(r, V)$. Therefore, the new activations, $act'(r)$, are updated as

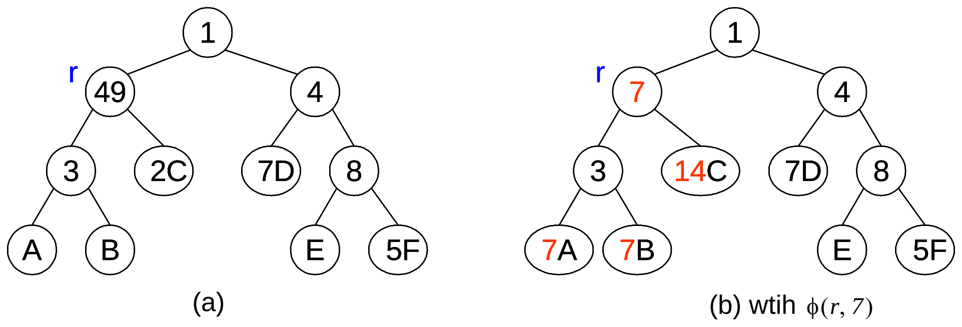


Figure 4.4 A FAPL operation example.

$$act'(r) = \frac{l(r)}{V}(act(left(r)) + act(right(r))) = \frac{act(r)}{V}.$$

QED.

Definition 4.2. Given an SDF edge e with $\alpha(src(e)) = a$, and $\alpha(snk(e)) = b$, we call $\phi(r, V)$ an **effective FAPL** on e if $a, b \in \lambda(r)$. Conversely, we call e an **effective edge** from ϕ.

Theorem 4.3. Given an SDF edge e and an effective FAPL $\phi(r, V)$ on e, the FAPL increases the buffer cost on e by a factor of V.

Proof: Suppose that $\alpha(src(e)) = a$, and $\alpha(snk(e)) = b$. With $\phi(r, V)$, we obtain new loop iteration counts $l'(a) = V \cdot l(a)$, $l'(b) = V \cdot l(b)$ for a, b, respectively. Suppose that w is the smallest parent of a and b. Then w must be contained in $tree(r)$. Along the paths of $path(a, left(w))$ and $path(b, right(w))$, the loop iteration counts of all the internal nodes of $tree(w)$ remain unchanged. Therefore, from Theorem 4.1, we can conclude that the buffer cost on e is increased by a factor of V. **QED.**

Theorem 4.4. Given a valid schedule, the new schedule that results from a FAPL operation is also a valid schedule.

Proof: First, a FAPL operation changes loop iteration counts only and in particular, such an operation does not change the topological sorting order associated with the initial schedule. Therefore, neither data dependencies nor schedule loop nesting structures are affected by FAPL operations. Given a FAPL operation effective on an SDF edge e, the same numbers of data values (Theorem 4.1) will be produced and consumed on the buffer associated with e and no buffer underflow nor overflow problems will be incurred. **QED.**

4.6.2. Properties of FAPL sequences

We call a sequence $\phi_1 \cdot \phi_2 \cdot \ldots \cdot \phi_m$ of FAPL operations applied sequentially (from left to right) as a **FAPL sequence**, and we represent such a sequence compactly by $\prod \phi_i$.

Definition 4.3. Given two FAPL sequences,

$$\Phi_1 = \prod_{i = 1 \ldots m} \phi_{1,i} \text{ and } \Phi_2 = \prod_{j = 1 \ldots n} \phi_{2,j},$$

we say that Φ_1 and Φ_2 are **equivalent** (denoted as $\Phi_1 \leftrightarrow \Phi_2$) if for every tree node a of $tree(r)$, we have $l_1(a) = l_2(a)$, where l_1 and l_2, respectively, give the loop iteration counts of tree nodes for the schedules that result from Φ_1 and Φ_2.

There are some useful properties about FAPL sequences, which we state here (without proof again, due to space constraints): **commutativity** $((\phi_1 \cdot \phi_2) \leftrightarrow (\phi_2 \cdot \phi_1))$ and **associativity** $(((\phi_1 \cdot \phi_2) \cdot \phi_3) \leftrightarrow (\phi_1 \cdot (\phi_2 \cdot \phi_3)))$.

Theorem 4.5. $(\phi_1 \cdot \phi_2) \leftrightarrow (\phi_2 \cdot \phi_1)$.

Proof: Suppose that $\phi_1 = \phi(r_1, V_1)$ and $\phi_2 = \phi(r_2, V_2)$. For any leaf node a and $a \in \lambda(r_1) \cap \lambda(r_2) \neq \varnothing$, the loop iteration count $l'(a)$ is equal to $l(a)V_1V_2$. If $r_1 = r_2$, the new loop iteration count, $l'(r_1)$, is updated as $l(r_1)/(V_1V_2)$. **QED.**

Theorem 4.6. $((\phi_1 \cdot \phi_2) \cdot \phi_3) \leftrightarrow (\phi_1 \cdot (\phi_2 \cdot \phi_3))$.

Proof: Suppose that $\phi_1 = \phi(r_1, V_1)$, $\phi_2 = \phi(r_2, V_2)$, and $\phi_3 = \phi(r_3, V_3)$. First, the right hand side can be reformatted as $(\phi_2 \cdot \phi_3) \cdot \phi_1$ according to the commutativity property provided in Theorem 4.5.

For any leaf node a such that $a \in \lambda(r_1) \cap \lambda(r_2) \cap \lambda(r_3) \neq \varnothing$, the new loop iteration count, $l'(a)$, is $l(a)V_1V_2V_3$. If $r_1 = r_2 = r_3$, the new loop iteration count, $l'(r_1)$, is updated as $l(r_1)/(V_1V_2V_3)$. If $a \in \lambda(r_1) \cap \lambda(r_2)$ and $a \notin \lambda(r_3)$, updating of the new loop iteration count $l'(a)$ is equivalent to $\phi_1 \cdot \phi_2 = \phi_2 \cdot \phi_1$. Similarly, if $a \in \lambda(r_1) \cap \lambda(r_3)$ and $a \notin \lambda(r_2)$, updating of $l'(a)$ is equivalent to $\phi_1 \cdot \phi_3 = \phi_3 \cdot \phi_1$.

Analogous reasoning applies to the updated value of $l'(r_1)$ if $r_1 = r_2 \neq r_3$ or if $r_1 = r_3 \neq r_2$. For the conditions of $a \in \lambda(r_2) \cap \lambda(r_3)$, $a \notin \lambda(r_1)$, or $r_2 = r_3 \neq r_1$, updating of the new loop iteration counts $l'(a)$ and $l'(r_2)$, is equivalent to that of $\phi_2 \cdot \phi_3$.

In summary, in all of the cases examined above, $(\phi_1 \cdot \phi_2) \cdot \phi_3 \leftrightarrow (\phi_2 \cdot \phi_3) \cdot \phi_1$ holds. **QED.**

4.6.3. FAPL-based heuristic algorithms

The problem of FAPL-based activations minimization is complex due to the interactions between the many underlying optimization variables. In this section, we propose an effective polynomial-time heuristic, called **GreedyFAPL**, for this problem.

GreedyFAPL, illustrated in Figure 4.5, performs block processing from pre-computed integer factorization results for the loop iteration counts of internal nodes. First, internal nodes are sorted in decreasing order based on their activations values. The sorted

```
Algorithm: GreedyFAPL
Input: An SDF graph, $tree(r)$, and buffer cost
        upper bound $M$
Output: Minimum activations

sort internal nodes by decreasing activations
for (each internal node $a$) {
    sort integer factors of $l(a)$ decreasingly
    for (each factor $\pi$ of $l(a)$) {
        compute overall buffer cost, $C$, as if
            $\phi(a, \pi)$ was run
        if ($C \le M$) {
            execute $\phi(a, \pi)$
        }
    }
}
return $act(r)$ as output
```

Figure 4.5 The GreedyFAPL algorithm for the ARMUB problem.

internal nodes are traversed one by one as FAPL target internal node targets. For each internal node target, the integer factors of the loop iteration count are tried in decreasing order as the FAPL factors until a successful FAPL operation (i.e., a FAPL operation that does not overflow the given memory constraint M) results. It is based on this consideration of integer factors in decreasing order that we refer to GreedyFAPL as a greedy algorithm.

Because the schedule tree is a binary tree and all leaf nodes are associated with unique actors, there are $|V|$ iterations in the outer *for* loop to traverse the internal nodes, where V is the set of actors in the input SDF graph. In the buffer cost constraint validation, we do not need to recompute the total buffer cost each time. Only the increase in buffer cost needs to be determined because we keep track of the overall buffer cost at each step.

In the worst case, for a given FAPL operation, all members of the edge set E in the input SDF graph are effective from the FAPL operation. Hence, the buffer cost constraint validation takes $O(|E|)$ time for each iteration of the inner *for* loop of GreedyFAPL. For every FAPL operation, updating the loop iteration counts of the target leaf nodes, and evaluating the buffer costs of the effective SDF edges costs involves $O(|V| + |E|)$ run-time complexity.

If Ω is the maximum number of factors in the prime factorization of the loop iteration count of an internal node, then the computational complexity of GreedyFAPL can be expressed as $O(\Omega|V|(|V| + |E|))$.

4.6.4. Handling of nonzero delays

As mentioned early in this section, the techniques developed here can easily be extended to handle delays. Specifically, given an SDF edge e with D units of delay, and a FAPL operation on e, the buffer cost induced from a FAPL operation on e can be determined as the sum of

$$D + F_{delayless}, \tag{4.5}$$

where $F_{delayless}$ represents the buffer cost of a FAPL operation on the "delayless version of" e — that is, the SDF edge obtained from e by simply changing the delay to zero. By using this generalized buffer cost calculation throughout (observe that when $D = 0$, the sum in Equation (4.5) still gives the correct result, from the definition of $F_{delayless}$), the methods developed earlier in this section can be extended to general acyclic SDF graphs (i.e., graphs with arbitrary edge delays). A similar extension can be performed for the techniques developed in the following section, but again for clarity, we will develop the technique mainly in the context of delayless graphs.

4.7. Activation Rate Minimization with Arbitrary Blocking Factors (ARMAB)

In this section, we generalize the problem formulation of the previous section to include consideration of non-unity blocking factors, called **ARMAB** for brevity. The resulting figure of merit of activation rate, $rate(S)$, provides an approximate measure to model the overall block processing performance enhancement of a schedule. In other words, schedules that have lower activation rates generally result in schedules with better performance. In Section 4.8, we include experimental results that quantify this claim that activation rate is a good indicator of overall schedule performance.

In this section we build on the insights developed in the previous section. Note that the problem we target in this section is strictly more general than that targeted in the previous section since ARMUB, as defined in Section 4.6, has the constraint $J(S) = 1$. Our study of the more restricted problem in Section 4.6 resulted in useful insights and techniques, most notably the GreedyFAPL approach, that are useful in the more general and practical context that we target in this section.

Our approach to ARMAB is to start with a buffer-efficient, minimal valid schedule as in Section 4.6, but unlike the approach of Section 4.6, we iteratively try to encapsulate the minimal valid schedule within an outer loop having different iterations counts, which correspond to different candidates for the target blocking factor. In this context, the blocking factor can be modeled in the schedule tree as the loop iteration count of the root node: i.e., by setting $l(r) = J$, where $tree(r)$ is otherwise equivalent to the schedule tree of the original (minimal) schedule.

We have derived the following useful result characterizing the effects of "pushing" an loop iteration count into the leaf nodes of a schedule tree from the root of the tree.

Theorem 4.7. Suppose that we are given a schedule tree $T = tree(r)$ with $l(r) = 1$ and buffer cost B. Suppose that we are also given an integer $J > 1$, and that we derive the tree T' from T by simply setting $l(r) = J$. If in T', we then apply $\phi(r, V)$, where V is an integer factor of J, then we obtain $rate(T') = (A_0 / V)$, where $A_0 = act(r)$ is the activations value for the original schedule tree T, and we also obtain a new buffer cost of $(V \times B)$.

Proof: With $\phi(r, V)$ applied, we have an updated loop iteration count $l(r) = J/V$, and an updated activation count $act(r) \cdot (J/V)$ of T' (according to Theorem 4.2). Thus, the activation rate of the overall graph is $act(r)/V$. The new buffer cost is then $(V \times B)$ based on Theorem 4.3. **QED.**

Theorem 4.7 tells us that with tree root r as the FAPL target internal node, the activation rate and new buffer cost are independent of J beyond the requirement that they be derived from a FAPL factor V that divides J. This allows us to significantly prune the search space of candidate blocking factors.

Given an acyclic SDF graph, an initial SAS, and a buffer cost upper bound, the minimal activation rate can be computed through a finite number of steps. Suppose that all of the possible FAPL sequences $\Phi_1, \Phi_2, \ldots, \Phi_n$ are provided through exhaustive search of the given instance of the ARMUB problem. We denote as a_i and b_i the activation rate (under unit blocking factor) and buffer cost, respectively, that are derived through Φ_i $(1 \le i \le n)$.

Now suppose that we are given a user-specified buffer cost upper bound M. Then for each Φ_i, the maximum FAPL factor for the tree root r is $\lfloor M/b_i \rfloor$. Therefore, the minimal activation rate for the given ARMAB problem instance is $min_i(a_i/\lfloor M/b_i \rfloor)$.

However, this derivation is based on an exhaustive search approach that examines all possible FAPL sequences. For practical use on complex systems, more efficient methods are needed. In the next subsection, we develop an effective heuristic for this purpose.

4.7.1. The FARM algorithm

Based on Theorem 4.7 and the developments of Section 4.6 we have developed an efficient heuristic to minimize the activation rate based on a given memory bound M. This heuristic, called **FARM (FAPL-based Activation-Rate Minimization)**, is outlined in Figure 4.6. The algorithm iteratively examines the FAPL factors $V = 1, 2, \ldots, \lfloor M/B \rfloor$, where B is the buffer cost of the original schedule. The original schedule is, as with the technique of Section 4.6, derived from APGAN and GDPPO, which construct buffer-efficient minimal schedules (without any consideration of block processing). Here, $\lfloor x \rfloor$ represents the largest integer that is less than or equal to the rational number x. For each of the candidate FAPL factors $V = 1, 2, \ldots, \lfloor M/B \rfloor$, the blocking factor is set to be equal to V, and the activation rate is minimized for that particular blocking factor.

The FARM algorithm keeps track of the minimal activation rate that is achieved over all iterations in which the memory constraint M is satisfied. The FAPL factor associated with this minimum activation rate and the resulting schedule tree configuration are returned as the output of the FARM algorithm.

The computational complexity of FARM is $O(\lfloor M/B \rfloor \Omega |V| (|V| + |E|))$.

```
Algorithm: FARM
Input: An SDF graph, $tree(r)$, and buffer cost
        upper bound $M$
Output: Minimum activation rate

let $B$ be the buffer cost induced from
$tree(r)$
set $MinRate = \infty$
for ($V = 1 \ldots \lfloor M/B \rfloor$) {
    copy $tree(r)$ to $tree'(r)$ with $l(r) = V$
    execute $\phi(r, V)$ on $tree'(r)$
    run $GreedyFAPL$ on $tree'(r)$ to obtain
        minimum activations $act(r)$
    if ($MinRate > (act(r)/V)$)
        set $MinRate = act(r)/V$
}
return $MinRate$ as output
```

Figure 4.6 The FARM algorithm for the ARMAB problem.

4.7.2. Integration of FARM and memory sharing techniques

Memory sharing is a useful technique to reduce the buffer cost requirement of an SDF application. In our ARMAB problem formulation, smaller activation rates are generally achievable if buffer sharing techniques are considered when specifying the buffer cost upper bound. In Figure 4.7, we propose a FARM-based algorithm, called *FARM-Sharing*, that integrates buffer sharing techniques into the basic FARM framework.

The FARM-Sharing algorithm feeds larger buffer cost bounds to FARM initially, and gradually reduces the bounds as the algorithm progresses by employing buffer sharing techniques. The algorithm employs two special parameters δ and ε, which can be tuned by experimentation on various benchmarks. In our experiments, after tuning these parameters, we have consistently used $\delta = 10$, and $\varepsilon = 0.5$.

Under buffer sharing, direct use of the buffer bound M will in general lead to buffer sharing implementations that over-achieve the bound, and do not use the resulting slack to

```
Algorithm: FARM-Sharing
Input: An SDF graph, $tree(r)$, buffer cost and buffer bound $M$
Output: Minimum activation rate

let $\delta$ be excess buffer cost ratio.
let $\varepsilon$ be reduction rate.
set $MinRate = \infty$.
set excess buffer cost $\Delta = M \times \delta$ and $\Delta \in Z$.
while ($\Delta > 0$) {
    run FARM on the SDF graph, $tree(r)$, and buffer cost bound $(M + \Delta)$.
    let $rate$ be the activation rate and $sched$ the SAS computed
        from FARM.
    let $\tau$ be the buffer sharing cost computed from any appropriate
        buffer sharing technique with the SDF graph and $sched$ as
        inputs.
    if ($\tau \le M$ and $rate \le MinRate$)
        $MinRate = rate$.
    else
        $\Delta = \Delta \times \varepsilon$.
}
return $MinRate$ as output.
```

Figure 4.7 A FARM-based algorithm integrating memory sharing techniques.

further reduce activation rate. Therefore, instead of directly using the user-specified bound of M, the FARM-sharing algorithm uses an internally-increased bound. This increased bound is initially set to o $M + \Delta$, where $\Delta = M\delta$. As the algorithm progresses, buffer sharing is used, and concurrently, the internally-increased bound is gradually reduced towards a final value of M. Any suitable buffer sharing technique can be employed in this framework; in our experiments we have used the lifetime-based sharing techniques developed in [37]. In this way, significantly smaller activation rates can generally achieved subject to the user-specified buffer bound M.

4.8. Experiments

To demonstrate the trade-off between buffer cost minimization and activation rate optimization, exhaustive search is employed to the CD to DAT sample rate conversion example of Figure 2.3. From the initial SAS of Figure 2.3(c), factor combinations of all loop iteration counts are exhaustively evaluated for the problem of ARMUB. The results are summarized in Figure 4.8. Each dot in this chart is derived from a particular factor

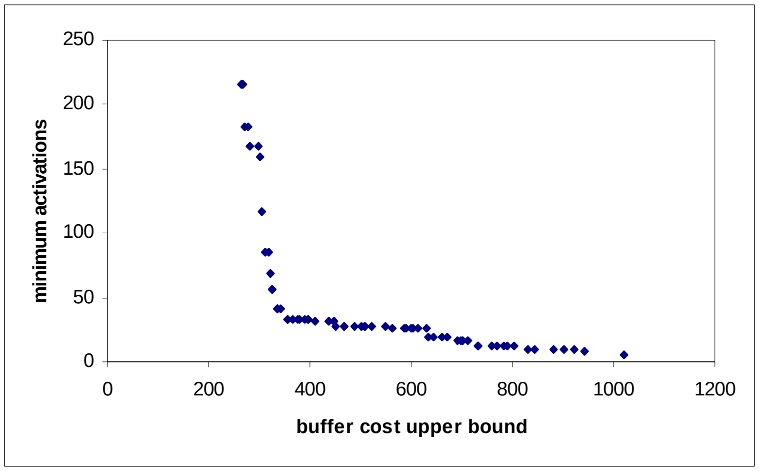


Figure 4.8 Trade-off between activations and buffer costs of CD to DAT.

combination and the activation rates are obtained by $min(rate(S))$ (vertical axis) subject to $buf(S) \leq M$, where M is the buffer cost bound (horizontal axis).

Experiments are set up to compare the activation rates achieved by our heuristics to the optimum achievable activation rates that we determine by exhaustive search. Exhaustive search is used here to understand the performance of our heuristics; due to its high computational cost, such exhaustive search is not feasible for optimizing large-scale designs nor for extensive design space exploration even on moderate-scale designs.

The following DSP applications are examined in our experimental evaluation: *16qam* (a 16 QAM modem), *4pam* (a pulse amplitude modulation system), *aqmf* (filter-bank), and *cd2dat* (CD to DAT sample rate conversion). These applications are ported from the library of SDF-based designs that are available in the Ptolemy design environment [17]. For each application, a number of buffer cost upper bounds (values of M) are selected uniformly in the range between the cost of the initial schedule (obtained from APGAN/GDPPO) to the cost of a flat schedule

Given a buffer bound M, the **degree of suboptimality (DOS)** of GreedyFAPL or FARM is evaluated as $(rate_{sub} - rate_{opt})/rate_{opt}$, where $rate_{opt}$ is the optimal activation rate observed for ARMUB or ARMAB, and $rate_{sub}$ is the activation rate computed by GreedyFAPL or FARM. The degrees of suboptimality thus computed are averaged over the number of buffer bounds selected to obtain an average degree of suboptimality. The results are summarized in Figure 4.9 and they demonstrate the abilities of GreedyFAPL, FARM to achieve optimum solutions for ARMUB, ARMAB, respectively, most of the time.

To further evaluate the efficiency of our algorithms, randomly-generated SDF graphs are experimented with. In these experiments, GreedyFAPL achieves optimal solutions approximately 90% of the time with 1.68% average DOS, and FARM achieves optimal solutions approximately 77% of the time with 0.97% average DOS.

To evaluate the impact of considering non-unity blocking factors, we performed experiments to compare GreedyFAPL and FARM for the CD to DAT example. In Figure 4.10, both algorithms are investigated with several large buffer cost bounds. Due to the restriction of unit blocking factor, it is shown in the figure that, at a certain point, GreedyFAPL reaches a limit and cannot reduce the activation rate any further with increases in

(a)

	16qam	4pam	aqmf1	aqmf2	aqmf3	aqmf4	cd2dat
DOS	0%	0%	1%	2%	0%	1%	3%

(b)

	16qam	4pam	aqmf1	aqmf2	aqmf3	aqmf4	cd2dat
DOS	0%	0%	0.18%	0.45%	0%	0.29%	1.25%

Figure 4.9 Effectiveness of (a) GreedyFAPL (b) FARM.

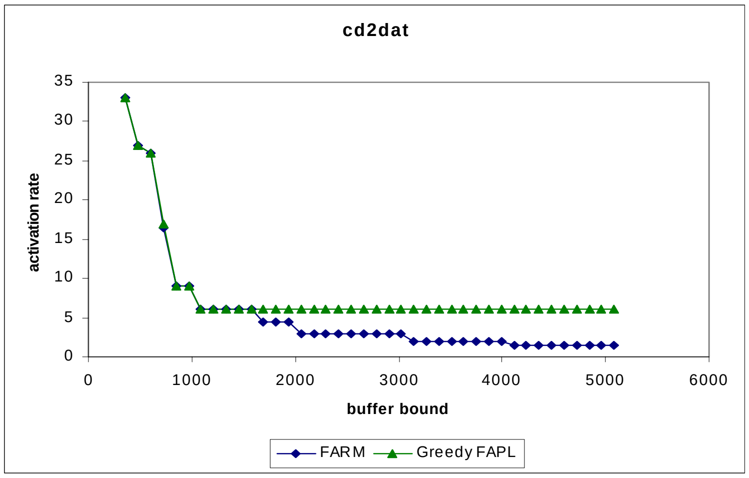


Figure 4.10 Comparison of GreedyFAPL and FARM.

allowable buffer cost (loosening of the bound M) beyond that point. In contrast, FARM generally keeps reducing activation rates as long the buffer bound M is loosened.

The effectiveness of FARM-Sharing is explored as well in our experiments for the CD to DAT benchmark. The observed activation rate improvements achieved by FARM-Sharing over FARM are summarized in Figure 4.11. Except for a few points, FARM-Sharing demonstrates over 20% activation rate reduction. The actual degree of improvement achieved varies significantly across applications.

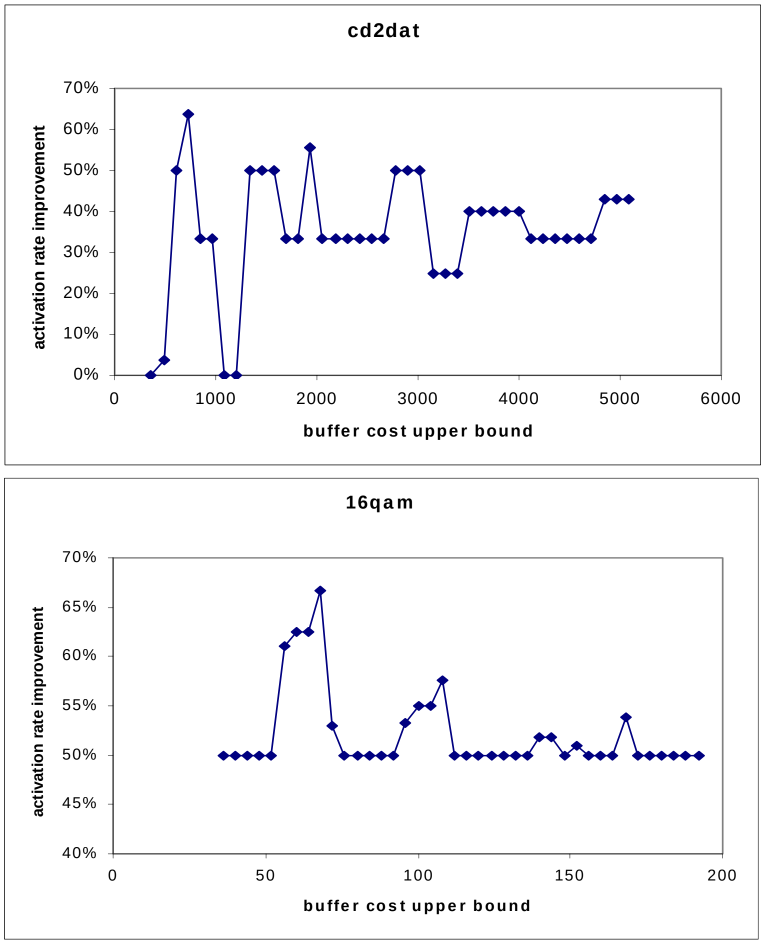


Figure 4.11 Activation rate improvement of FARM-Sharing over FARM.

Some experiments are also conducted to evaluate the suitability of the activation rate as a high-level estimator for performance. In our context, the performance can be characterized as the average number of execution cycles required per graph iteration, which we refer to as the **average latency**. Let $L = (l_1, \ldots, l_n)$ denote the average latencies of an application under various blocking factor settings (these latencies are measured experimentally), and let $R = (r_1, \ldots, r_n)$ denote the corresponding activation rates (these can be calculated directly from the schedules). To measure the accuracy of activation-rate driven performance optimization, we use the estimation **fidelity**, as defined by

$$fidelity = \frac{2}{n(n-1)}\left(\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} f_{ij}\right),$$

where $f_{ij} = 1$ if $sign(L_i - L_j) = sign(R_i - R_j)$, and $f_{ij} = 0$ otherwise.

Our fidelity experiments are summarized in Figure 4.12 for four kernel functions (FIR, add, convolution, and DCT), and also for a complete application (CD to DAT conversion). Except for the DCT function, the activation rate is seen to be a good high level model for comparing different design points in terms of average latency. Average latencies of block processing implementations of the DCT increase with increasing vectorization degrees, however, the vectorized forms still demonstrate better performance compared to the scalar implementation (Figure 4.12).

(a)

	FIR	add	conv	DCT	cd2dat
fidelity	0.79	1	1	0	1

(b)

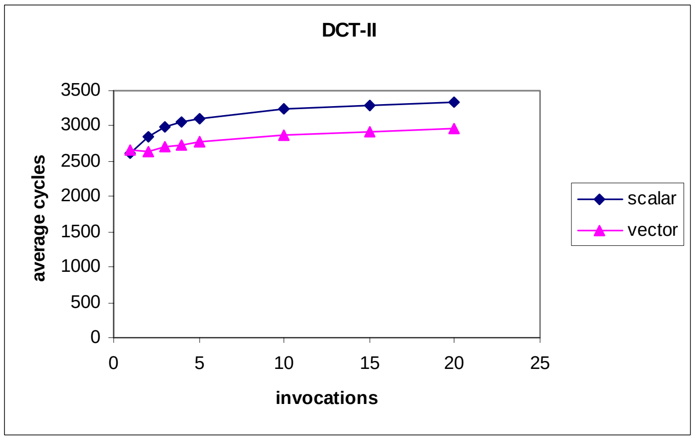


Figure 4.12 (a) Fidelity experiments. (b) Average latencies of DCT.

Chapter 5. Parameterized Looped Schedules

This chapter is concerned with the compact representation of execution sequences in terms of efficient looping constructs. Here, by a looping construct, we mean a compact way of specifying a finite repetition of a set of execution primitives. Such compaction, which can be viewed as a form of hierarchical **run-length encoding (RLE)**, has application in many system prototyping contexts, including efficient control generation for Kahn processes on FPGAs, and software synthesis for static dataflow models of computation. In this chapter we significantly generalize previous models for loop-based code compaction of DSP programs to yield a configurable code compression methodology that exhibits a broad range of achievable trade-offs. Specifically, we formally develop and apply to DSP hardware and software implementation a parameterizable loop scheduling approach with compact format, dynamic reconfigurability, and low-overhead decompression. In our experiments, this new approach demonstrates up to 99% storage saving (versus RLE) and up to 46% frequency enhancement (versus another parameterized approach) in FPGA synthesis, and an average of 11% code size reduction in software synthesis compared to existing methods for code size reduction.

5.1. Introduction

Due to tight resource constraints and the increasing complexity of applications, efficient program compression techniques are critical in the prototyping of embedded DSP systems. Hardware and software subsystems for DSP often present periodic and deterministic execution sequences that facilitate compile- or synthesis-time compression. In this chapter, we develop a methodology that exploits this characteristic of DSP subsystems through compact representation of execution sequences in terms of efficient looping constructs. The looping constructs provide a concise, parameterized way of specifying sequences of execution primitives that may exhibit repetitive patterns of arbitrary forms both at the primitive- and subsequence-levels. Such compaction provides a form of hierarchical run-length encoding as well as reconfigurability during DSP system prototyping. Moreover, exploitation of low-cost hardware features are considered to further improve the efficiency of the proposed methods. The power and flexibility of our approach is demonstrated concretely through its application to control generation for Kahn processes [16] on FPGAs, and to software synthesis for static dataflow models of computation, such as those developed in [7][29].

5.2. Related Work

Sequence compression techniques have been developed for many years in the context of file compression to save disk space, reduce network traffic, etc. One basic approach in this and other sequence compression domains is to express repeating strings of symbols in more compact forms. A typical example is run-length encoding, which replaces repeated instances of a symbol by a single instance of the symbol along with the repetition count. Several bitmap file formats, e.g., TIFF, BMP and PCX, adopt variants of run-length encoding. More elaborate compression strategies employ “dictionary” look-up mechanisms. Here, multiple instances of a symbol sequence are replaced by smaller-sized pointers that reference a single “master copy” of the repeated sequence. The collection of master copies can therefore be viewed as a dictionary for purposes of compression. An example is the LZ77 algorithm [58], variations of which are used in many data compression tools.

Code compression in embedded systems presents some unique characteristics and challenges compared to compression in other domains. First, code sequences depend heavily on the underlying control flow structures of the associated programs. Furthermore, the control flow structures of the associated programs can often be changed subject to certain restrictions, giving rise in general to a family of alternative code sequences for the same program behavior. Second, memory resources in embedded systems are particularly limited, and the temporary “scratch space” for decompression is usually very small. Third, decompression of embedded code must be fast enough to meet real-time demands.

There are several research works discussing reduction of code size through classical compiler optimizations such as strength reduction, dead code elimination, and common sub-expression elimination [15]. A particularly effective strategy is procedural abstraction [33], where procedures are created to take the place of duplicated code sequences. The work of [13] further reveals that procedural abstraction combined with classical compiler optimizations result in more compact code size than each approach can achieve alone.

For embedded DSP design, application representations are often based on dataflow models of computation. The work of [8] adopts a dynamic programming approach to reformat repeated dataflow executions in a hierarchical run-length encoding style. However, the computational complexity is relatively large, especially in rapid prototyping contexts. In [7], two complementary loop scheduling algorithms for dataflow-based DSP programs are proposed for joint code and data memory minimization. In the methods of [7] and [8], the constraint of static and fixed iteration counts in the targeted class of looping structures significantly restricts compression results. In [5], a meta-modeling approach is developed for incorporating dynamic reconfiguration capability into different dataflow modeling styles. When applied to SDF, this meta-modeling framework results in the **parameterized synchronous dataflow (PSDF)** model of computation. The developments in [5] center around a hybrid compile-time/run-time scheduling technique that is specialized to PSDF representations.

In this chapter, we propose a flexible and parameterizable looping construct, and associated analysis methods. Because the approach is formulated in terms of compressing fixed execution sequences, this looping construct is applicable to any representation, such as SDF and cyclo-static dataflow [9], from which static schedules can be derived. The looping construct provides compact format, dynamic reconfigurability, and fast decompression. The construct embeds functions in describing variable repetition lengths in a configurable form of run-length encoding to achieve better compression results.

As a consequence, appropriate execution subsequences can be derived by adjusting parameter values at run time without modifying the hardware implementation. Our proposed methodology applies looping constructs that provide flexibility in adapting execution sequences, as well as efficiency in managing the associated iteration control. In summary, we propose an approach for compact representation of execution sequences that is effective across the dimensions of conciseness, decompression performance, cost, and configurability.

5.3. Background: Static Looped Schedules

We denote the set of all integers by Z, and the set of non-negative integers by Z^+. Suppose $S = (s_1, s_2, \ldots, s_m)$ is a sequence of arbitrary elements and c is a non-negative integer. Then we define the product $S \times c$ to be the sequence that results from concatenating c copies of S. Thus, for example $S \times 0$ is the empty sequence; $S \times 1 = S$; $S \times 2 = (s_1, s_2, \ldots, s_m, s_1, s_2, \ldots, s_m)$; and so on. Furthermore, if $T = (t_1, t_2, \ldots, t_n)$ is another sequence, then we define the sum $S + T$ to be the concatenation of T to S: $S + T = (s_1, s_2, \ldots, s_m, t_1, t_2, \ldots, t_n)$. Note that in general $(S + T)$ does not equal $(T + S)$.

We occasionally abuse notation by overloading the definition of a function depending on the type of argument that is applied. For example, as explained fully in subsequent sections, if X is an instruction, then $c(X)$ defines the cost of that instruction, whereas if X is a schedule, then $c(X)$ denotes the total cost of that schedule (including the sum of instruction and loop costs). We abuse notation in this way to highlight relationships across closely-related functions, and to contain the total number of distinct symbols that are defined.

Suppose we are given a finite sequence of symbols $P = (p_1, p_2, \ldots, p_n)$ from a finite alphabet set $A = \{a_1, a_2, \ldots, a_m\}$. Thus, each $p_i \in A$. We refer to each p_i as an **instruction**, and we refer to the sequence P as the **program** that we wish to optimize. We define a **class 0 (static) schedule loop** over A to be a parenthesized term of the form $(cI_1I_2\ldots I_k)$, where $c \in Z^+$, and each I_i is either an element of A (i.e., an instruction) or a ("nested") class 0 schedule loop. The number c is called the **iteration count** of the schedule loop, and each I_i is called an **iterand** of the schedule loop. The concatenation

$I_1I_2\ldots I_k$ of iterands is called the **body** of the schedule loop. Such a schedule loop is called **static** because the iteration count is constant.

A **class 0 (static) looped schedule** over A is a sequence $S = (x_1, x_2, \ldots, x_n)$, where each x_i is either an element of A or a class 0 schedule loop over A. Note that by definition, if $L = (cI_1I_2\ldots I_k)$ is a class 0 schedule loop, then $S_L = (I_1, I_2, \ldots, I_k)$ and (L) are both class 0 looped schedules. We call S_L the **body schedule** of L.

Given a class 0 looped schedule S, a schedule loop L is **contained** in S if for some i, x_i is a schedule loop and $x_i = L$ or L is a schedule loop that is nested within x_i. For example, consider $S = ((3A(2B(3CD))), E, (3(2B)))$. This schedule contains the following schedule loops: $(3A(2B(3CD)))$, $(2B(3CD))$, $(3CD)$, $(3(2B))$, and $(2B)$. Note that in listing the set of schedule loops that are contained in a schedule, we may need to distinguish between multiple schedule loops that have identical iteration counts and bodies, as in the first and second appearances of $(3AB)$ in the looped schedule $((2A(3AB)), (5CD), (3AB))$. If L_1 and L_2 are schedule loops that are contained in the schedule $S = (x_1, x_2, \ldots, x_n)$, we say that L_1 is **contained earlier** than L_2 in S if there exist x_i and x_j such that $i < j$, x_i contains L_1, and x_j contains L_2. We say that L_1 **lexically precedes** L_2 in S if (a) L_1 is contained earlier than L_2 in S; (b) L_2 is nested within L_1; or (c) S contains a schedule loop L_3 so that L_1 is contained earlier than L_2 in the body schedule of L_3.

Example 5.1. Consider the looped schedule $((2A(3B))CD, (3A(2(3B)(2C))))$, let L_1 denote the first appearance of $(3B)$, let L_2 denote the second appearance of $(3B)$, let L_3 denote the schedule loop $(2A(3B))$, and let L_4 denote the schedule loop $(2C)$. Then L_1

lexically precedes L_2 due to condition (a); L_3 lexically precedes L_1 due to condition (b); and L_2 lexically precedes L_4 due to condition (c).

Consider an iterand I of a class 0 schedule loop. If I is an instruction, then we say that the **program generated by** I, denoted $P(I)$, is simply the one-element sequence (I). Otherwise, if I is a schedule loop — that is, I is of the form $I = (c_I X_1 X_2 \ldots X_p)$ — then $P(I)$ is defined recursively by

$$P(I) = (P(X_1) + P(X_2) + \ldots + P(X_p)) \times c\,. \tag{5.1}$$

Similarly, given a class 0 schedule $S = (x_1, x_2, \ldots, x_n)$, the program generated by S is (with a minor abuse of notation) denoted $P(S)$, and is given by

$$P(S) = P(x_1) + P(x_2) + \ldots + P(x_n)\,. \tag{5.2}$$

Example 5.2. Suppose that the set of instructions A is given by $A = \{a, b, c, d\}$, and suppose we are given a looped schedule $S = (a, (2c(2ad)d), b, (3c), d)$. Then we have

$$P(S) = (a, c, a, d, a, d, d, c, a, d, a, d, d, b, c, c, c, d)\,. \tag{5.3}$$

Static looped schedules have been studied extensively in the context of software synthesis from SDF representations of DSP applications (e.g., see [7]).

If costs are associated with individual actors and with loop construction in general, then we can express the degree of compactness associated with specific looped schedules. Suppose that in the context of looped schedule implementation, α_{loop} represents the overhead (cost) of a loop, and $\alpha(x)$ represents the cost of an instruction x. For example, for software implementation α_{loop} represents the code size cost associated with a loop in the target code. This value will normally depend on the processor on which the schedule is being implemented, and will include the code size of the instructions required to initialize the loop and update the loop counter at the beginning or end of each iteration. If the software is being implemented for a dataflow graph specification, then the “instructions” in the looped schedule correspond to actors in the dataflow graph, and the instruction code size values $\{\alpha(x)\}$ give the code size requirements of the different actors on the associated target processor.

The cost of a looped schedule S can be expressed as

$$\alpha(S) = n_{loop}(S)\alpha_{loop} + \sum_{x \in A} n_{app}(x, S)\alpha(x), \tag{5.4}$$

where $n_{loop}(S)$ denotes the number of schedule loops in S (including nested loops), and $n_{app}(x, S)$ denotes the number of times that instruction x appears in schedule S. For example, if $S = (a, (2c(2ad)d), b, (3c), d)$, the schedule illustrated above, then

$$\alpha(S) = 3\alpha_{loop} + 2\alpha(a) + \alpha(b) + 2\alpha(c) + 3\alpha(d). \tag{5.5}$$

To construct a static looped schedule from a sequence of instructions, a dynamic programming approach called **CDPPO** [8] provides an effective approach. The CDPPO algorithm adopts a bottom-up approach to fuse repetitive instruction sequences into hierarchical looping constructs. The objective of CDPPO is to minimize overall code size, including the costs for instructions and looping constructs. CDPPO has computational complexity that is polynomial in the number of instructions in the (uncompressed) input sequence.

5.4. Class 1 Looped Schedules

Static looped schedules provide a simple form of nested iteration where all iteration counts in the loops are static values, and loop counts implicitly progress from 1 to the corresponding iteration count limits in uniform steps of 1. However, static looped schedules do not always allow for the most compact representation of a static execution sequence. This motivates the definition of more flexible schedules in which more general updating of loop counters is integrated into the schedule. The class 1 schedules, which we define next, represent one such form of more general schedules. In class 1 schedules, the loop counter dimension is made explicit, and loop counters are allowed to have initial values, and update expressions specified for them. Because update expressions are processed frequently (once per loop iteration), their form is restricted in class 1 schedules to ensure efficient hardware and software implementation.

Formally, a class 1 schedule loop L has five attributes, a body, an index, an iteration count function, an initial index value, and an index update constant. The body of L is defined in a manner analogous to the body of a class 0 schedule loop. Thus, the body of L is of the form $I_1 I_2 \ldots I_n$, where each I_i, called an **iterand** of L, is either an instruction or a class 1 schedule loop. The **index** of a class 1 schedule loop L is a symbol that represents a loop index variable that is associated with L in an implementation of the loop. The **iteration count function** of L is an integer-valued function $f(y_1, y_2, \ldots, y_m)$ defined on Z^m, where each y_i is the index of some other class 1 schedule loop or is a parameter of a looped schedule that contains L. The value of f just before executing an invocation of L gives the minimum value of the index required for the loop to stop executing. In other words, L will continue executing as long as the index value is less than f. It is admissible to have $m = 0$, so that f represents a constant value v. In this case, we write $f() = v$. The **initial index value** of L is an integer to which the loop index variable associated with L is initialized. This initialization takes place before each invocation of L, just prior to the computation of f. The **index update constant** is a positive integer that is added to the index of L at the end of each iteration of L.

A class 1 schedule loop L is represented by the parenthesized term $([x_L, f_L, u_L]\ B_L)$, where x_L, f_L, B_L, and u_L are, respectively, the index, iteration count function, body, and index update constant of L. For brevity we omit the initial index value from this representation. The initial index value of L is denoted by $x_L(0)$; this value is specified separately when needed. Furthermore, when $u_L = 1$, we may suppress u_L from the schedule loop notation, and simply write $L = ([x_L, f_L] B_L)$. If x_L is not an argument of any relevant iteration count function, we may suppress x_L, and write $L = ([f_L] B_L)$ or $L = ([f_L, u_L] B_L)$; if, additionally, f_L is constant-valued (i.e., $f_L = c$), and $u_L = 1$, then we have a class 0 schedule loop, and we may drop the brackets and write $L = (cB_L)$, which is just the usual notation for class 0 schedule loops. We represent the arguments of the iteration count function by $args(f_L) = \{y_1, y_2, \ldots, y_m\}$. It is a fact that the number of iterations executed by an invocation of the class 1 schedule loop L is given by

$$iterations = \max\left(0, \left\lceil \frac{f_L(y_1{*}, y_2{*}, \ldots, y_m{*}) - x_L(0)}{u_L} \right\rceil\right), \tag{5.6}$$

where $y_i{*}$ denotes the value of index y_i just prior to initiation of I_L.

A **class 1 looped schedule** over A is an ordered pair $S = (params(S), body(S))$. The first member $params(S) = \{p_1, p_2, \ldots, p_r\}$ of this ordered pair is a finite set of elements called **parameters** of S, and the second member $body(S) = (x_1, x_2, \ldots, x_n)$ is a finite sequence where each x_i is either an element of A or a class 1 schedule loop over A.

We say that a looped schedule S is **syntactically correct** if the following three conditions all hold.

- Every loop $L = ([x_L, f_L, u_L]B_L)$ that is contained in S has a unique index x_L.
- $\{x_L | S \ contains\ L\} \cap params(S) = \varnothing$; that is, the parameters of S are distinct from the loop indices.
- For each loop L that is contained in S, the iteration count function f_L is either constant-valued, or depends only on parameters of S, and indices of loops that lexically precede L; that is,

$$args(f_L) \subseteq params(S) \cup \{x_{L'} | L'\ lexically\ precedes\ L\ in\ S\}. \tag{5.7}$$

Containment of a schedule loop earlier than another schedule loop, as well as lexical precedence between schedule loops, are defined for class 1 looped schedules in a manner analogous to that for class 0 looped schedules.

Syntactic correctness is a necessary but not sufficient condition for validity of a looped schedule. Overall validity in general depends also on the context of the looped schedule. For example, a syntactically correct looped schedule for an SDF graph may be invalid because the schedule is deadlocked (attempts to execute an actor before sufficient data has been produced for it).

Intuitively, the semantics of executing a class 1 schedule loop $([x_L, f_L, u_L]\ B_L\ )$, where $f_L = f_L(y_1, y_2, \ldots, y_m)$ and $u_L \in Z^+$, can be described as outlined in Figure 5.1. Using this semantics, we can define the program generated by an iterand of a class 1

```
$x_L = x_L(0)$
$limit_L = f_L(y_1, y_2, \ldots, y_m)$
while $(x_L < limit_L)$
        execute $B_L$
        $x_L = x_L + u_L$
end while
```

Figure 5.1 A sketch of the execution of a loop.

schedule loop, and the program generated by a class 1 looped schedule in a fashion analogous to the corresponding definitions for class 0 looped schedules. However, when determining these generated programs for class 1 looped schedules, we must specify an assignment of values to the schedule parameters. Thus, if $v : params(S) \to Z$ is an assignment of values to parameters of a class 1 looped schedule S, then we write $P(S, v)$ to represent the corresponding program generated by S.

Example 5.3. Suppose $\alpha = (A, B, C, D, E, F)$, and consider the class 1 looped schedule S specified by $params(S) = \{p_1\}$ and $body(S) = (F, ([x_1, f_1]AB([f_2, 2]CD)), E)$, where $f_1 = f_1(p_1) = p_1 - 3$ and $f_2 = f_2(x_1) = 5 - x_1$. Notice that this schedule contains a pair of nested schedule loops. If the initial index values in these loops are identically zero, and if $v(p_1) = 6$ (i.e., we assign the value of 6 to the schedule parameter p_1), then we have

$$P(S, v) = (F, A, B, C, D, C, D, C, D, A, B, C, D, C, D, A, B, C, D, C, D, E). \tag{5.8}$$

This simple example illustrates some of the ways in which more irregular programs can be generated by class 1 looped schedules as compared to their class 0 counterparts. In particular, in this example, we see that the number of iterations of the inner loop can vary across different invocations of the loop, and furthermore, the amount of this variation need not be uniform.

Theorem 5.1. Given a syntactically-correct PCLS S, and an assignment $v : params(S) \to Z$ of parameter values, the generated program $P(S, v)$ is finite.

Proof: Suppose that L is a schedule loop contained in S. Then there is a unique sequence $L_1, L_2, \ldots, L_n$, $n \geq 1$, of schedule loops contained in S such that L_1 is an iterand of S, $L_n = L$, and each L_{i+1} is an iterand of L_i. That is, $L_1, L_2, \ldots, L_{n-1}$ are the outer loops encapsulating L. Then the total number of invocations of L in an execution of S can be expressed as

$$\prod_{i=1}^{n} iterations(L_i).$$

This follows from (5.6), which specifies the number of iterations executed by a given schedule loop invocation I_L. Since $u_{L_i} > 0$ and f_{L_i} is integer-valued, this number of iterations will always be finite. **QED.**

Because of their potential for parameterization, in terms of schedule parameters and loop indices, and because of their restriction that loop indices be updated by constant additions, we also refer to class 1 looped schedules as **parameterized, constant-update looped schedules (PCLSs)**.

5.5. Affine Looped Schedules

One useful special case of looped schedules arises when f_L is a linear function of $args(f_L)$. We call this special case **affine parameterized looped schedules (APLSs)**.

5.5.1. Isomorphism of Looped Schedules

The ability to parameterize iteration counts in PCLSs is useful in expressing related groups of static schedule loops. In many useful design contexts, families of static schedule loops arise, such that within a given family, all loops are equivalent in a certain structural sense. We refer to this form of equivalence between loops as schedule loop **isomorphism**. Specifically, two class 0 schedule loops L_1 and L_2 are **isomorphic** if there is a bijection f between the set of loops contained in (L_1) and the set of loops contained in (L_2) such that for each L in the domain of f, $L = (cI_1I_2\ldots I_m)$ and $f(L) = (dJ_1J_2\ldots J_n)$ satisfy the following three conditions: 1) L and $f(L)$ have the same number of iterands (that is, $m = n$); 2) for each i such that I_i is not a loop (i.e., it is a "primitive" iterand), we have $J_i = I_i$; and 3) for each i such that I_i is a loop, we have that J_i is also a loop, and furthermore, I_i and J_i are isomorphic.

For each loop L contained in (L_1), the mapping $f(L)$ of L is called the **image** of L under the isomorphism. Furthermore, two static looped schedules S_1 and S_2 are said to be isomorphic if the loops $(1S_1)$ and $(1S_2)$ are isomorphic.

We can extend the definition of isomorphic looped schedules to a finite set of static looped schedules $S_1, S_2, \ldots, S_k$. In this case, we extract the loops from $(1S_i)$ for some arbitrary i. Then for all $j \neq i$ and for each loop L contained in $(1S_i)$, we define $f_j(L)$ to be the corresponding, structurally equivalent loop in $(1S_j)$.

5.5.2. Basics of APLS derivation

Using the concept of looped schedule isomorphism, we derive useful formulations in this section for the special case of APLSs where $args(f_L) = params(S)$ for every L contained in S.

For clarity in this discussion, we start with p as the only schedule parameter (i.e., $params(S) = \{p\}$). Under the APLS assumption, this means that the iteration count expression for each schedule loop will be of the form $ap + b$, where a and b are constants. Therefore, we need two instances of a given static schedule loop to fit the unknowns a and b. We simply need that these instances be for distinct values of p, say q_1 and q_2, and that these values of p be such that they reach beyond any transient effects (leading to negative, zero, or one-iteration loops when viewed from the final parameterized schedule). Note that functionally, a negative-iteration loop is just equivalent to a zero-iteration loop.

Let S_1 be the looped schedule instance corresponding to q_1 and let S_2 be the looped schedule instance corresponding to q_2. If S_1 and S_2 are not isomorphic, we need to increase $min(q_1, q_2)$, and try again.

Suppose now that we have an isomorphic schedule pair S_1 and S_2. We then take each loop L in S_1 and its image $f(L)$ in S_2. Let z_1 be the iteration count of L and z_2 be that of $f(L)$. We then set up the equations

$$z_1 = aq_1 + b\text{, and } z_2 = aq_2 + b\text{,} \tag{5.9}$$

and solve these equations for a and b. We repeat this procedure for all loops L that are contained in S_1.

Generalizing this to multiple schedule parameters, we start with a hypothesized APLS $S(p_1, p_2, \ldots, p_N)$ in $N \geq 1$ parameters. The iteration count expression for each schedule loop L is of the form $(a_1p_1 + a_2p_2 + \ldots + a_Np_N + b)$. We need $N + 1$ instances of L to fit the $N + 1$ unknowns in the iteration count expression for L. For $i = 1, 2, \ldots, N + 1$, let S_i be the ith element in our set of compacted looped schedule instances. Let $q_{i,1}, q_{i,2}, \ldots, q_{i,N+1}$ be the corresponding parameter values for $p_1, p_2, \ldots, p_{N+1}$, respectively. Furthermore, let L be a loop in S_i, and for each $i = 1, 2, \ldots, N + 1$, let z_i denote the iteration count of $f_i(L)$. We set up the following equations:

$$\begin{aligned} z_1 &= a_1q_{1,1} + a_2q_{1,2} + \ldots + a_Nq_{1,N} + b \\ z_2 &= a_1q_{2,1} + a_2q_{2,2} + \ldots + a_Nq_{2,N} + b \\ &\cdots \\ z_{N+1} &= a_1q_{N+1,1} + a_2q_{N+1,2} + \ldots + a_Nq_{N+1,N} + b\,. \end{aligned} \tag{5.10}$$

This can be expressed in matrix form as $\bar{z} = QA + \bar{b}$, where $\bar{z}$ is an $(N+1) \times 1$ constant column vector, A is an $N \times 1$ column vector composed of the unknown a_i's, Q is an $(N+1) \times N$ constant matrix composed of the parameter settings used in the selected schedule instances, and $\bar{b} = [b\ b\ \ldots\ b]^T$ is an $(N+1) \times 1$ column vector obtained by replicating the unknown offset term b.

By solving the linear equations, we obtain $a_1, a_2, \ldots, a_N$ and b to formulate the APLS loop implementation of L. If a solution cannot be obtained, we can increase the selected $\{q_{i,1}, q_{i,2}, \ldots, q_{i,N+1}\}$ (to more completely bypass transient effects, as described earlier), or we may change the hypothesized number of parameters in the looped schedule.

Example 5.4. Suppose we have a function unit (*FU*) with input data selected through a multiplexer (*MUX*) as in Figure 5.2. The input data sources are A, B, and the multiplexer has a control line (*CS*) for selecting one source at each instant. During the execution of FU, the multiplexer needs to determine a sequence of source executions to obtain proper input data. Suppose that this sequence is determined at run time by an integer parameter p, and that under parameter assignments $v_1(p) = 5$ and $v_1(p) = 6$, the corresponding sequences are $S_1 = (A, A, B, A, A, B)$ and

$$S_2 = (A, A, A, A, B, B, B, B, A, A, A, A, B, B, B, B, A, A, A, A, B, B, B, B),$$

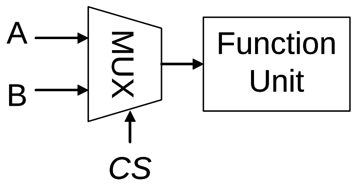


Figure 5.2 A function unit with input data selected through a multiplexer.

respectively. These sequences can be compacted into the static schedule loops $L_1 = (2(2A)(1B))$ and $L_2 = (3(4A)(4B))$. Then employing the APLS formulation techniques, the two loops can be unified into a singe expression $L = ([f_1]([f_2]A)([f_3]B))$, where $f_1(p) = p - 3$, $f_2(p) = 2p - 8$, and $f_3(p) = 3p - 14$.

5.5.3. Consolidating Loops within a Schedule

While the previous subsection focuses on isomorphism across schedules, this subsection discusses isomorphism within a schedule. Let us first look at a class 0 looped schedule, $S = ((1A), B, (2A), B, (3A), B)$. The schedule S cannot be compressed further by class 0 scheduling algorithms due to the heterogeneous iteration $(1A)$, $(2A)$, and $(3A)$. However, schedules $((1A), B)$, $((2A), B)$, and $((3A), B)$ contained in S are isomorphic to each other and our isomorphism-based compression technique is able to unify them in a single APLS loop. By inspection, we can easily evaluate this unified loop to be $L = ([y_1, f_1]([f_2]A)B)$, where $f_1 = 3$, $f_2 = y_1 + 1$, and $y_1(0) = 0$.

Motivated by this example, we now describe a formal method to compute loops of this kind in a general fashion. Given a static schedule $S = (x_1, x_2, \ldots, x_n)$, suppose that S contains z consecutive isomorphic subschedules and each subschedule contains w $(w \geq 1)$ elements of S. Let the z consecutive subschedules be represented successively (from left to right) as

$$S_0 = (x_i, x_{i+1}, \ldots, x_{i+w-1}),$$

$$S_1 = (x_{i+w}, x_{i+w+1}, \ldots, x_{i+2w-1}),$$

$$\ldots$$

$$S_{z-1} = (x_{i+(z-1)w}, x_{i+(z-1)w+1}, \ldots, x_{i+zw-1}),$$

where $i \geq 1$, and $i + zw - 1 \leq n$. In addition, given an integer j $(i \leq j \leq i + w - 1)$, any pair of elements in the subset $\{x_j, x_{j+w}, \ldots, x_{j+(z-1)w}\}$ are isomorphic, and we therefore call this subset the *isomorphism family* α_j. Furthermore, suppose that elements of each subschedule S_u $(0 \leq u \leq z - 1)$ — i.e., x_{i+uw}, $x_{i+uw+1}, \ldots, x_{i+(u+1)w-1}$ — are not uniformly isomorphic one another, and therefore, that $S_0, S_1, \ldots, S_{z-1}$ form an *isomorphism basis* (i.e., a decomposition into maximal isomorphic subschedules). Our goal is to consolidate these z subschedules into a single APLS schedule loop

$$L = ([y, f_L]x'_i x'_{i+1} \ldots x'_{i+w-1}), \tag{5.11}$$

where $f_L() = z$ and x_j' is an iterand evaluated from the isomorphism family α_j.

In our formulation, values of y are set as the subschedule subscripts, i.e., $y = t$ is for S_t. For any loop l contained in x'_j, we need to derive the loop iteration count function, say $f_l()$. For brevity, we discuss only the case of $args(f_l()) = \{y\}$; our treatment of this case can be extended however to more general case. After consolidating all the subschedules, the new S becomes $S = (S_L, L, S_R)$, where S_L (S_R) is a possibly-empty subschedule that immediately precedes S_0 (succeeds S_{z-1}) in the original schedule S.

Deriving L is then similar to that discussed in the previous subsection. The affine function of $f_l()$ is $ay + b$ and a, b are to be solved through the z isomorphic images in the isomorphism family α_j.

5.5.4. Further Consolidation of Loops by Incorporating Schedule Parameters

The APLS derivation techniques in the previous subsection can be incorporated with schedule parameters for further compaction. Consider the following two static schedule instances that involve an associated parameter p:

$$S_{v(p)=5} = (A, B, (2A), B, (3A), B) \text{ and}$$

$$S_{v(p)=6} = ((3A), B, (4A), B, (5A), B, (6A), B).$$

By employing the basic APLS derivation technique, we obtain

$$S_{v(p)=5} = (([x, f_1]([f_2]A)B)) \text{ and}$$

$$S_{v(p)=6} = (([y, g_1]([g_2]A)B)),$$

where $f_1 = 3$, $f_2 = x + 1$, $g_1 = 4$, and $g_2 = y + 3$. Since both APLSs are isomorphic to one another, there are chances to merge them into one. A new iteration count function $p - 2$ can replace both f_1 and g_1. For f_2 and g_2, the distinct constant shifts (i.e., 1 of $x + 1$ and 3 of $y + 3$) can be consolidated into one affine function $2p - 9$. Therefore, the compacted form is $S = (([i, h_1]([h_2]A)B))$, where $h_1 = p - 2$ and $h_2 = i + 2p - 9$.

In generalizing this kind of derivation, we assume for simplicity here that we are working with schedules that involve a single parameter p only. Multiple parameters can be handled with a straightforward (but more notationally cumbersome) extension.

Suppose that schedule loop instances of Equation (5.11) are provided. For any loop l contained in L, our goal is to relate the iteration count function $ay + b$ to p in affine functions. That is, $a = c'p + d'$ and $b = c''p + d''$, although this formulation may make ay a non-affine term. By solving for c', d', c'', and d'', we obtain the new iteration count function $(c'p + d')y + (c''p + d'')$ for l.

Example 5.5. Let us revisit the input selection example in Figure 5.2. Suppose the selection sequences are generated by the pseudocode in Figure 5.3(a) and P is an integer parameter. If the loop iteration space is drawn on a plane, it will look like Figure 5.3(b) (with P set to 6). The sequence goes from bottom to top, starting at the left-most column, and traversing the columns from left to right. For example, the sequence for $v(P) = 6$ is

$$(A, A, A, A, A, A, B, A, A, B, A, A, B, B, A, B, B, A, B, B, B, B, B, B). \tag{5.12}$$

Our task now is to compact in APLS form the raw sequences associated with this example, assuming no prior knowledge about the sequence generation mechanism. To this end, the sequence in Equation (5.12) can be first compacted as a static looped schedule, $((6A), (2B(2A)), (2(2B)A), (6B))$. The second and third elements are isomorphic and can be compacted further as $((6A), ([x, f_1](2([f_2]B)([f_3]A))), (6B))$, where $f_1 = 2$, $f_2 = x + 1$, and $f_3 = -x + 2$. Following the same procedure, we obtain a schedule for $v(P) = 7$, $((8A), ([y, g_1](2([g_2]B)([g_3]A))), (8B))$, where $g_1 = 3$, $g_2 = y + 1$, and $g_3 = -y + 3$. If P is incorporated, a unified schedule can be developed as

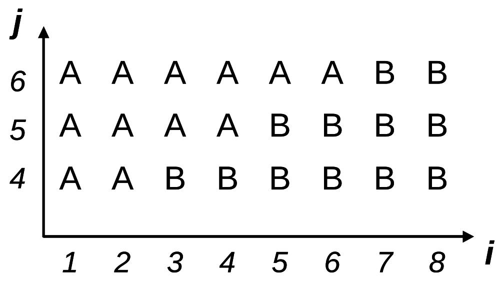
$$(([h_1]A), ([t, h_2](2([h_3]B)([h_4]A))), ([h_5]B)),$$

where $h_1 = h_5 = 2P - 6$, $h_2 = P - 4$, $h_3 = t + 1$, and $h_4 = -t + P - 4$.

If transient effects are carefully considered, we can introduce further compression through certain forms of “dummy” iterands. For example, $(6A)$ can be re-written as

```
for (int i = 1; i <= 2*P-4; i++) {
    for (int j = 4; j <= P; j++) {
        if ((j-4) < floor((i-1)/2))
            select B;
        else
            select A;
```

(a)



(b)

Figure 5.3 (a) A pseudocode demonstrating the input selection sequence for Figure 5.2. (b) The corresponding loop iteration space (p=6) of the pseudocode.

$(2(0B))(3A))$. Through this observation, the APLS can be reformulated as $(([t, h_1](2([h_2]B)([h_3]A))))$, where $h_1 = P-2$, $h_2 = t$, and $h_3 = -t+P-3$. Systematically exploiting transient effects in this way is an interesting direction for further work.

5.5.5. Pseudo-affine parameterized looped schedules

A useful generalization of APLS is to adopt pseudo-affine functions. In this case, the iteration count functions turn into (for a single parameter p) $\langle a_1, \ldots, a_m \rangle_p p + \langle b_1, \ldots, b_n \rangle_p$, where there are m (n) possibilities for a (b), and the choice depends on the value of p.

Example 5.6. Suppose that we are given a pseudo-affine loop $L = ([\langle 1, 2\rangle_p p + \langle 2, 3\rangle_p]A)$ such that if p is odd, then 1 will be chosen from $\langle 1, 2\rangle_p$, and if p is even, then 2 will be chosen, and similarly, 2 for odd p and 3 for even p will be selected from $\langle 2, 3\rangle_p$. It can be verified then that L will return $(5A)$ for $v(p) = 3$ and $(11A)$ for $p = 4$.

Example 5.7. Suppose that the upper bound of the outer loop in Figure 5.3(a) is changed to $P+2$, rather than $2P-4$, and assume that $v(P) \geq 8$. Then we will obtain

$$S_o = (([x, f_1](2([f_2]B)([f_3]A))), ([f_4]B), ([f_5]A))$$

for odd $v(P)$, where $f_1 = f_4 = (P+1)/2$, $f_2 = x$, $f_3 = -x+P-3$, and $f_5 = (P-7)/2$. For even $v(P)$, we have $S_e = (([y, g_1](2(([g_2]B)([g_3]A)))))$, where $g_1 = (P+2)/2$, $g_2 = y$, and $g_3 = -y+P-3$. The two APLSs can be consolidated into a single pseudo-affine formulation

$$S = (([t, h_1](2(([h_2]B)([h_3]A)))), ([h_4]B), ([h_5]A)),$$

where $h_1 = P/2 + \langle 1/2, 1 \rangle$, $h_2 = t$, $h_3 = -t + P - 3$, $h_4 = \langle 1/2, 0 \rangle_P P + \langle 1/2, 0 \rangle_P$, and $h_5 = \langle 1/2, 0 \rangle_P P + \langle -7/2, 0 \rangle_P$.

5.6. Application: Synthesis from Kahn Process Networks

The computation model of the **Kahn Process Network (KPN)** expresses applications in terms of distributed control and memory. The KPN model [16] assumes a network of concurrent autonomous processes that communicate in a point-to-point fashion over unbounded FIFO channels, using a blocking-read synchronization primitive. Each process in the network is specified as a sequential program that executes concurrently with other processes.

To facilitate migration from an imperative application specification, which is preferred by many programmers, to a KPN specification, a set of tools, **Compaan** and **Laura** [50], is being developed, as illustrated in Figure 5.4(a). This approach allows parts of an application written in a subset of MATLAB to be converted automatically to KPNs. The conversion is fast and correct-by-construction. The obtained KPN processes can be mapped to software or hardware.

5.6.1. Interface control generation

In the synthesis flow of Laura, a VHDL description of an architecture is generated from a KPN. Laura converts a process specification together with an IP core into an abstract architectural model, called a **virtual processor** [16]. Every virtual processor is composed of four units (Figure 5.4(b)): **Execution**, **Read**, **Write**, and **Controller**. Execution units contain the computational parts of virtual processors. To communicate data on FIFO channels, **ports** are devised, which connect FIFO channels and virtual processors.

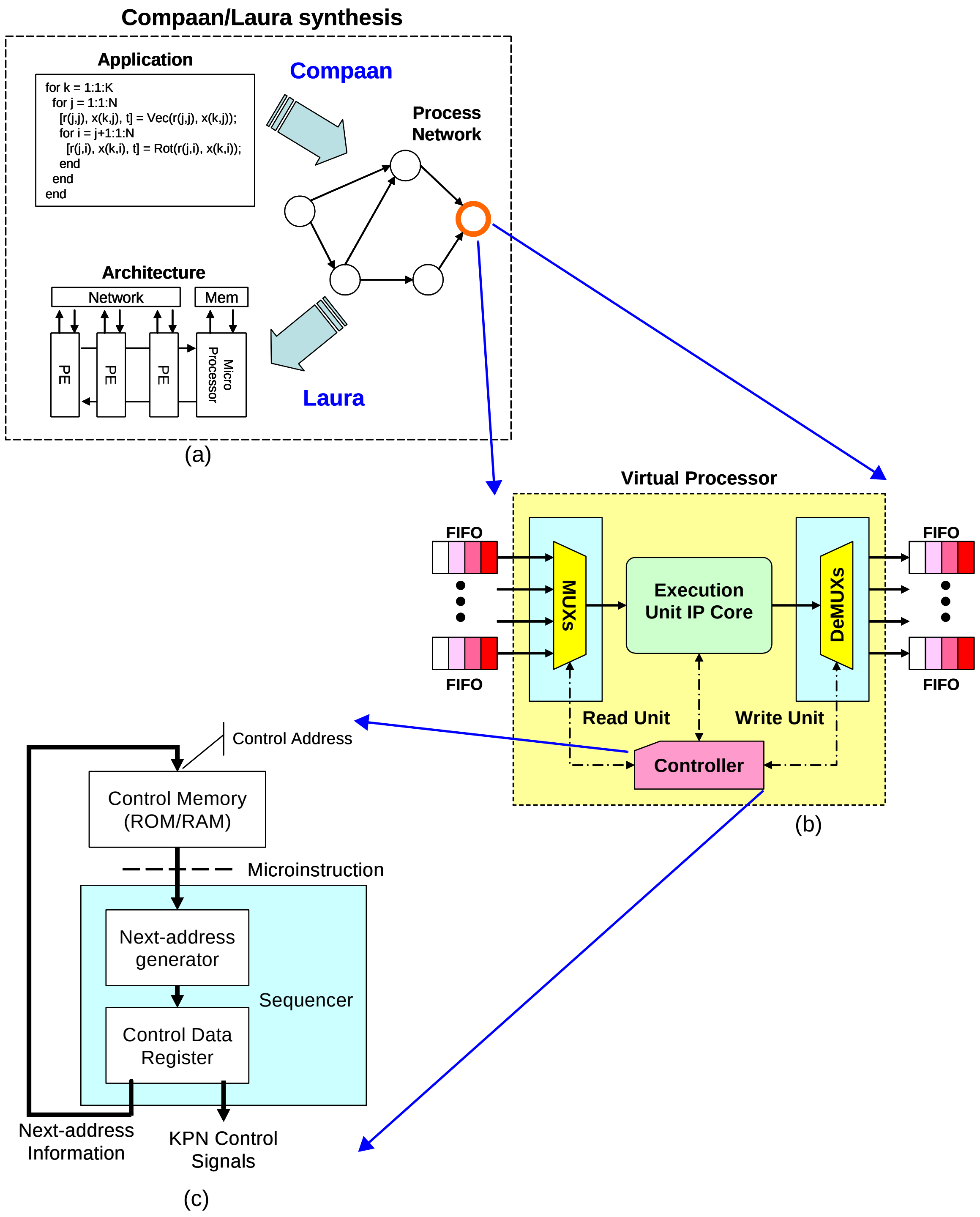


Figure 5.4 Synthesis overview of APLSs for Compaan traces.

Read/write units are in charge of multiplexing/de-multiplexing port accesses for execution units. Controller units provide valid port access sequences, or **traces**, to facilitate computation. The determination of traces, also called **interface control generation**, in a systematic way and compact form is our focus here.

A simple approach to implementing the distributed control is to use ROM tables to store the traces. However, this strategy is impractical because of large hardware costs. To reduce the complexity, several compile time techniques are proposed to compress these tables and to keep the flexibility offered by the parametric approach [16]. In this chapter, PCLSs are employed to compact traces to demonstrate the effectiveness of PCLS for hardware implementation.

To reduce hardware area costs of the ROM table approach, construction of looped schedules can be used. Moreover, applications specified in the KPN model may have parameters that can be configured at run time. The constructed looped schedules highly depend on the parameters values set dynamically. With the isomorphism formulation stated in Section 5.5 for APLS, groups of isomorphic looped schedules can be summarized by single APLSs if the formulation is possible (as in Figure 5.5). This is the way we generate the parameterizable and compact schedules, which result in significantly better performance than ROM tables.

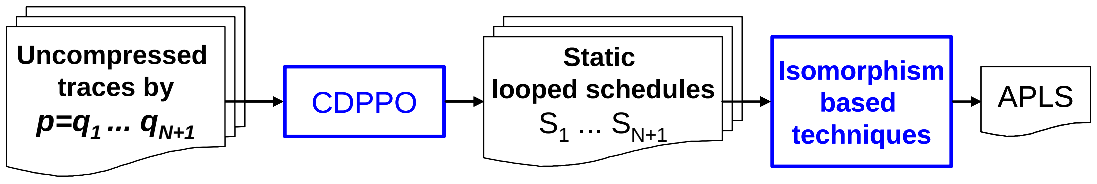

Uncompressed traces by $p=q_1 \ldots q_{N+1}$
CDPPO
Static looped schedules $S_1 \ldots S_{N+1}$
Isomorphism based techniques
APLS

Figure 5.5 APLS generation for Compaan traces.

5.6.2. FPGA Setup for Controller Units

KPN control generation using PCLS is implemented in a **micro-engine** architecture. Under the requirements of a virtual processor controller, the micro-engine has to perform a *for*-loop operation and generate a KPN control symbol in one cycle. As shown in Figure 5.4(c), a PCLS controller consists of two parts: a ROM/RAM memory and a **sequencer**. In the ROM/RAM memory is stored a compiled version of a PCLS, which describes a trace using micro-instructions. The sequencer uncompresses the PCLS trace and generates the desired KPN port through fetching and decoding micro-instructions from the control memory. The memory address of the next micro-instruction needs to be evaluated as well by the sequencer. To realize PCLSs in FPGA hardware implementation, the following two steps can be employed:

- Symbolic program compilation: The first step involves the compilation of the input PCLS using the micro-engine instruction primitives. This is done at the symbolic level.
- Hardware program generation: The second step takes the symbolic program and transforms it in a bit-stream suitable for an FPGA platform. This step takes into account the bitwidths of the loop count and the symbols used in the PCLS trace.

In hardware, encoding methods, such as one-hot or binary encoding, can be used for the program symbols. The choice of encoding schemes is done as a function of the dimension of the implementation and/or speed constraints.

5.6.3. Experiments

Our experiments are based on implementation costs of the controller units on an FPGA. The experiments apply the isomorphism-based APLS formulations developed in Section 5.5 to efficiently provide for dynamic reconfiguration across scalable families of KPN implementations.

In Figure 5.6, we show the FPGA area costs (number of FPGA slices) for a number of applications. Here, *QR* is a matrix decomposition algorithm, and *Optical* is an image restoration algorithm [34]. For each application, particular processes are selected for our experiments. We compare the area costs under ROM table and PCLS implementations. Also, FPGA area and maximum frequency in generating port accesses are shown. For example, virtual processor 3 (*VP3*) of the KPN representing Optical, requires a ROM table size of 944460 bytes with parameter values set to $W = 320$ and $H = 200$. The size reduces to only 160 bytes if the PCLS scheme is employed. All experiments are set up on a Xilinx Virtex-II 2000 equipped device.

The obtained results are promising in terms of area and frequency. For example, the largest PCLS trace occupies only 1% of the total FPGA slices while the ROM table approach, on contrast, uses approximately 9%. For the QR algorithm, we derived the ROM table for a set of typical parameters values ($N = 7$ and $K = 21$). The results with the compression technique applied show a considerable compression rate for this kind of application.

PCLS achieves enhancement also compared to advanced techniques experimented with in [16] (also on the Virtex-II 2000). In Figure 5.6, PCLS achieves up to 99% of byte savings over RLE and up to 46.4% frequency improvement over the **parameterized**

predicate controller (PPC) approach, with however, an overhead of 37.9% more slices required.

In this section, we have shown that our proposed PCLS methodology is effective for interface control generation. It offers the flexibility of a parametric controller with small hardware resource requirements. However, it is possible that the PCLS algorithm cannot optimally compress some execution sequences, and this can affect controller performance. We can see this trend from Figure 5.6, where the trace size difference affects the frequency of the entire design.

5.7. Application: Synthesis from Synchronous Dataflow

To save memory in storing actor execution sequences, previous studies have incorporated looping constructs to form static looped schedules. SASs in the form of static looped schedules, however, limit the potential for buffer minimization as shown in Figure 5.7(a). The SAS of Figure 5.7(a) has a higher buffer cost than the non-SAS does. The

	RLE (bytes)	PCLS (bytes)	saving	PPC (MHz)	PCLS (MHz)	impr.	slices ovhd.
QR VP4	400	20	*95%*	140	205	*46.4%*	*37.9%*
Optical VP3	14850	160	*98.9%*	129	150	*16.3%*	*36.3%*

Virtual Processor	ROM size (bytes)	PCLS size (bytes)	PCLS freq. (MHz)	PCLS slices
QR VP2	35	6	207	35
QR VP3	176	16	209	37
QR VP4	616	20	205	40
Optical VP3	944460	160	150	132

Figure 5.6 Experimental results for Compaan/KPN synthesis.

fixed iteration counts of static loops lack the flexibility to express irregular patterns, such as the non-SAS. In contrast, the more flexible iteration control associated with the PCLS approach naturally accommodates the non-SAS in Figure 5.7(a).

5.7.1. Minimizing Code and Data Size via PCLS

We start by considering two-actor SDF graphs to minimize buffer costs through PCLS. A useful lower bound on the buffer memory requirement of a two-actor SDF graph, as in Figure 5.7(b), is $m + n - gcd(m, n)$, and an algorithm is given in [7] to compute schedules that achieve this bound. Intuitively, this algorithm executes the source actor just enough times to trigger execution of the sink actor, and the sink actor executes as many times as possible (based on the available input data) before control is transferred back to the source actor.

Theorem 5.2. Given a two-actor SDF graph as in Figure 5.7(b), depending on the values of m and n, the buffer memory lower bound $m + n - gcd(m, n)$ can be reached through either of the following PCLSs:

- If $m \geq n$, $PCLS = (([x_1, f_1]A([f_2]B)))$, where

$$f_1 = n / \mathrm{gcd}(m, n) \text{ and } f_2 = \lfloor m(x_1 + 1)/n \rfloor - \lfloor mx_1/n \rfloor. \tag{5.13}$$

- If $m \leq n$, $PCLS = (([x_1, f_1]([f_2]A)B))$, where

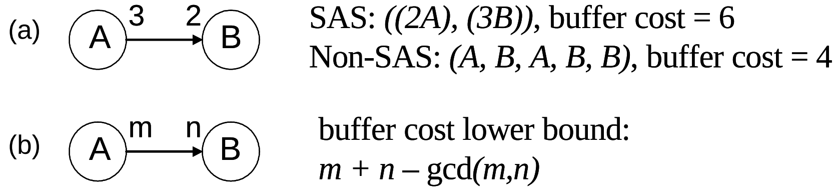

(a)
A
3
2
B
SAS: ((2A), (3B)), buffer cost = 6
Non-SAS: (A, B, A, B, B), buffer cost = 4
(b)
A
m
n
B
buffer cost lower bound:
m + n – gcd(m,n)

Figure 5.7 schedules and buffer costs for two-actor SDF graphs,

$$f_1 = m/\gcd(m, n) \text{ and } f_2 = \lceil n(x_1 + 1)/m \rceil - \lceil n x_1/m \rceil. \tag{5.14}$$

Proof: This proof generally follows the reasoning behind the algorithm in [7] that provably reaches the minimum buffer bound for two-actor SDF graphs as in Fig. 5.7(b). Let us start with the case $m \geq n$. Every execution of A produces more tokens than are consumed by B. To make the smallest buffer size feasible, B must be executed repeatedly in a way that the token consumption catches up to the production as closely as possible. This behavior is carried out by the inner loop $(A[x_2, f_2]B)$: B is consecutively executed to digest the live tokens to the full extent (i.e., any further execution of B at this point would lead to buffer underflow). The number of iterations of the outer loop is identical to the number of firings of A because A is executed only once in the inner loop. A valid SDF schedule requires that the total numbers of tokens produced and consumed have to be equal on an edge (this condition is referred to the **balance equation** for the edge). Therefore,

$$f_1 = \frac{\text{total tokens exchanged}}{m}$$

$$= \frac{mn}{gcd(m, n)} \cdot \frac{1}{m}$$

$$= \frac{n}{gcd(m, n)}$$

is necessary for the minimum schedule period. To determine f_2, we have to examine the total token consumption and production for an iteration. Up to the end of the $(x_1 + 1)$th iteration, there is a total of $(x_1 + 1)m$ tokens produced and $\lfloor (x_1 + 1)n/m \rfloor$ executions of B are required to maximally consume the tokens. Therefore, at the $(x_1 + 1)$th iteration, B

needs $\lfloor (x_1 + 1)n/m \rfloor$ executions minus $\lfloor x_1 n/m \rfloor$ executions that have occurred in the previous iteration. Therefore, we obtain the equation of f_2, as shown above.

Example 5.8. With a similar argument, we can derive the PCLS for the case $m \le n$. For the case of $m = n$, it does not matter which formulation ($m \ge n$ or $m \le n$) is used because both result in the same PCLS, (A, B). **QE**A PCLS can be derived for the SDF graph in Fig. 5.7(a) by applying Theorem 5.2:

$$\left([x_1, 2]\left(A, \left(\left[x_2, \left\lfloor \frac{3(x_1 + 1)}{2} \right\rfloor - \left\lfloor \frac{3x_1}{2} \right\rfloor\right] B\right)\right)\right).$$

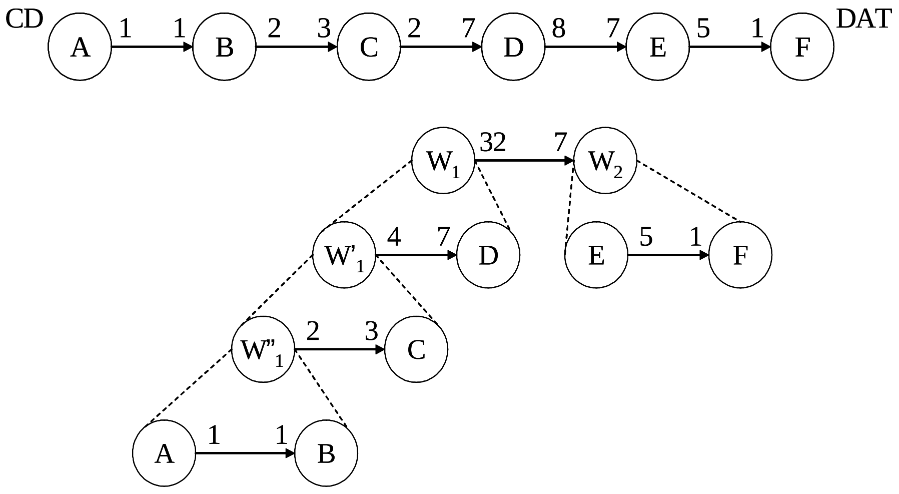
By expanding the loop hierarchy, we obtain the execution sequence (A, B, A, B, B), which results in the minimum buffer bound as shown in Fig. 5.7(b).

To extend this two-actor PCLS formulation to arbitrary acyclic graphs, we can apply the recursive graph decomposition approach in [25]. The work of [25] focuses on systematic implementation based on nested procedure calls, where both data and program memory size are considered in the optimization process. The work of [25] starts by effectively decomposing an SDF graph into a hierarchy of two-actor SDF graphs. The example of CD to DAT sample rate conversion is given in Figure 5.8 to demonstrate this decomposition process. To adapt the approach to PCLS implementation, the graph decomposition hierarchy can be mapped into a corresponding hierarchy of PCLS-based parenthesized terms.

5.7.2. Experiments

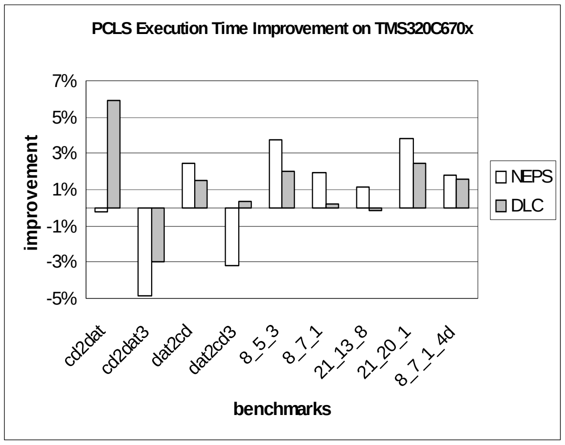
Experiments are set up to compare the results of PCLS-based inline synthesis with two other advanced techniques for joint code/data minimization, nested procedure synthesis (**NEPS**) [25] and dynamic loop-count inline synthesis (**DLC**) [39]. Our comparison is in terms of execution time and code size. Nine benchmarks available from the Ptolemy tool [17] are used in the experiments. The first four benchmarks are different multi-stage implementations of sample-rate conversion between CD and DAT formats. The other five, with labels of the form *x_y_z*, are for non-uniform filter banks, where the high (low) pass filters retain $y/x\,(z/x)$ of the spectrum. In the PCLS-based synthesis, iteration counts are pre-computed and saved in arrays so that they can be retrieved efficiently by indexing. The target processors are from the Texas Instruments TMS320C670x series.

Experimental results are summarized in Figure 5.9. We measure the performance improvement of PCLS over NEPS and DLC for both execution time and code size. Formally, percentage numbers are calculated by $(X - PCLS)/X$, where X is the result of NEPS or DLC. A positive (negative) percentage indicates that PCLS performs better



The PCLS is: $(([x_1, f_1]([x_2, f_2]([g_2]([x_3, f_3]([g_3]AB)C))D)([g_1](E(5F)))))$, where
$f_1 = 7, f_2 = 4, f_3 = 2$,
$g_1 = \lfloor 32(x_1 + 1)/7 \rfloor - \lfloor 32x_1/7 \rfloor$,
$g_2 = \lceil 7(x_2 + 1)/4 \rceil - \lceil 7x_2/4 \rceil$, and
$g_3 = \lceil 3(x_3 + 1)/2 \rceil - \lceil 3(x_3/2) \rceil$.

Figure 5.8 Two-actor graph decomposition for CD to DAT.



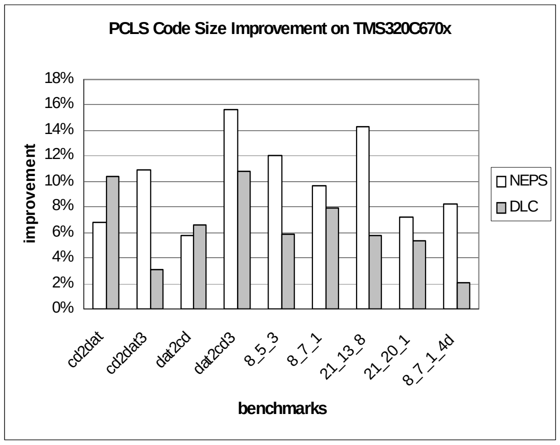


Figure 5.9 Comparison of PCLS, NEPS, and DLC synthesis.

(worse). PCLS synthesis demonstrates small advantages execution time for the filter bank examples, which require longer execution latency compared to the rate conversion benchmarks. Regarding code size efficiency, PCLS demonstrates more utility (average code size reduction of 11%, 7% over NEPS and DLC, respectively).

Our development of PCLS is further advantageous compared to alternative methods because it can naturally provide compaction for groups of static schedules, as demonstrated in Section 5.5, instead of just individual schedules in isolation. This advantage is especially useful for rapid prototyping, where designers may wish to experiment across a set of alternative implementations without having to re-synthesize for each experiment.

Chapter 6. Data Partitioning

Many modern DSP processors have the ability to access multiple memory banks in parallel. Efficient compiler techniques are needed to maximize such parallel memory operations to enhance performance. On the other hand, stringent memory capacity is also an important requirement to meet, and this complicates our ability to lay out data for parallel accesses. We examine these problems, data partitioning and minimization, jointly in the context of software synthesis from dataflow representations of DSP algorithms. Moreover, we exploit specific characteristics in such dataflow representations to streamline the data partitioning process. Based on these observations on practical dataflow-based DSP benchmarks, we develop simple, efficient partitioning algorithms that come very close to optimal solutions. Our experimental results show 19.4% average improvement over traditional coloring strategies with much higher efficiency than ILP-based optimal partitioning computation. This is especially useful during design space exploration, when many candidate synthesis solutions are being evaluated iteratively. A preliminary summary of part of this chapter is published in [27].

6.1. Introduction

Limited memory space is an important issue in design space exploration for embedded software. An efficient strategy is necessary to fully utilize stringent storage resources. In modern DSP processors, the memory minimization problem must often be considered in conjunction with the availability of parallel memory banks, and the need to place certain groups (usually pairs) of storage blocks (program variables or arrays) into distinct banks. This chapter develops techniques to perform joint data partitioning and minimization in the context of software synthesis from SDF. We report on insights on program structure obtained from analysis of numerous practical SDF benchmark applications, and apply these insights to develop an efficient data partitioning algorithm that frequently achieves optimum results.

The assignment techniques that we develop consider variable-sized storage blocks as well as placement constraints for simultaneous bank accesses across pairs of blocks. These constraints derive from the feature of simultaneous multiple memory bank accesses provided in many modern DSP processors, such as the Motorola DSP56000, NEC μPD77016, and Analog Devices ADSP2100. These models all have dual, homogenous parallel memory banks. For example, consider the architecture of the **Motorola DSP56000** programmable digital signal processor [35], which is illustrated in Figure 6.1.

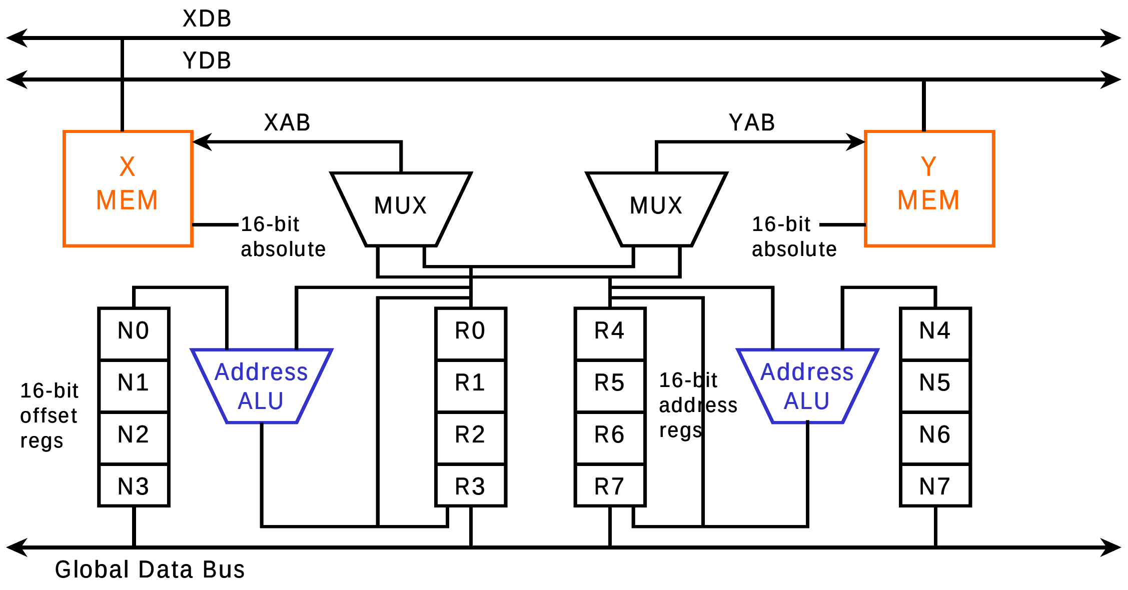


Figure 6.1 Motorola DSP56000 memory banks and address generation units.

The DSP56000 has two data memory banks that allow parallel data accesses. Accompanying the memory banks are two independent sets of **address generation units (AGU's)**, address register files, and address multiplexers. Address registers *R0* through *R3* and offset registers *N0* through *N3* are dedicated to one of the AGUs, and address registers *R4* through *R7* and offset registers *N4* through *N7* are dedicated to the other. Each AGU can post-increment or post-decrement a single address *Ri* by the constant one or the contents in the corresponding offset register *Ni*. Each multiplexer generates at most one effective address each cycle. Addresses generated by the two multiplexers must point to locations of different memory banks. For software programming, the DSP56000 allows up to two data move operations to be encoded in an instruction word. The data moves can be memory accesses, register transfers, or immediate loads. However, due to the nature of the DSP56000 micro-architecture, a set of restrictions are imposed on parallel data moves. These capabilities and constraints challenge the development of optimized compilers and several research works can be found that center around this topic [12][18][27][31][48][49][51][56].

Memory allocation techniques that consider this architectural characteristic can employ more parallelism and therefore speed up execution. The issue is one of performing strategic **data partitioning** across the parallel memory banks to map simultaneously-accessible storage blocks into distinct memory banks. Such data partitioning has been researched for scalar variables and register allocation [12][18][51]. However, the impact of array size is not investigated in those papers. Furthermore, data partitioning has not been explored in conjunction with SDF-based software synthesis. The main contribution of this chapter is in the development of novel data partitioning techniques for heterogeneous-sized storage blocks in the synthesis of software from SDF representations.

In this chapter, we assume that the potential parallelism in data accesses is specified by a high level language, e.g., C. Programmers of the SDF actor library provide possible and necessary parallel accesses in the form of language directives or pseudocode. Then the optimum bank assignment is left to software synthesis. Because of the early specifications, users can not foresee the parallelism that will be created by compiler optimization techniques, like code compaction and selection. It is neither our intention to explore such low level parallelism. From the benchmarks collected (in the form of undirected graphs), a certain structural pattern is found. The observations help in the analysis on practical applications and motivates a specialized, simple, and fast heuristic algorithm.

6.2. Related Work

Due to performance concerns, embedded systems often provide heterogeneous data paths. These systems are generally composed of specialized registers, multiple memory modules, and address generators. The heterogeneity opens new research problems in compiler optimization.

One such problem is memory bank assignment. One early article of relevance on this topic is [44]. This work presents a naive alternating assignment approach. In [48], interference graphs are derived by analyzing possible dual memory accesses in high level code. Interference edges are also associated with integer weights that are identical to the loop nesting depths of memory operations. The rationale behind the weight definition is that memory loads/stores within inner loops are called more frequently. The objective is to evaluate a maximum edge cut such that the induced node sets are accessed in parallel most often. A greedy heuristic is used due to the intractability of the maximum edge cut problem [19]. A similar problem is described in [31] though with an **Integer Linear Programming (ILP)** strategy employed instead.

Register allocation is often jointly discussed with bank assignment. These two problems lack orthogonality, and are usually closely related. In [51], a constraint graph is built after symbolic code compaction. Variables and registers are represented by graph nodes. Graph edges specify constraints according to the target architecture's data path as well as some optimization criteria. Nodes are then labelled under the constraints to reach lowest labelling cost. Because of the high intractability of the problem, a simulated annealing approach is used to compute solutions. In [18], an evolutionary strategy is combined with tree techniques and list scheduling to jointly optimize memory bank assignment and register allocation. The evolutionary hybrid is promising due to linear order complexity. Unlike phase-coupling strategies, a de-coupling approach is recently suggested in [12]. Conventional graph coloring is employed in this work along with maximum spanning tree computation.

While the algorithms described above are effective in parallel memory operations, array size is not considered. For systems with heterogeneous memory modules, the issue of variable size is important when facing storage capacity limitations. Generally, the optimization objective aims at promoting execution performance. Memory assignment is done according to features (e.g., capacity and access speed) of each module to determine a best running status [2]. Configurability of banks is examined in [42] to achieve an optimum working configuration. Furthermore, trade-offs between on-chip and off-chip memory data partitioning are researched in [41]. Though memory space occupation is investigated in those papers, parallel operations are not considered. The goal is to leverage overall execution speed-up by exploiting each module’s advantage.

A similar topic, termed memory bank disambiguation, can be found in the field of multiple processor systems. The task is to determine which bank a memory reference is accessing at compile-time. One example is the compiler technique for the **RAW** architecture from MIT [4]. The architecture of RAW is a two-dimensional mesh of tiles and each tile is composed of a processor and a memory bank. Because of the capability of fast static communication between tiles, fine-grained parallelism and quick inter-bank memory accesses can be accomplished. Memory bank disambiguation is rendered in compile time to support static memory parallelism as much as possible. Since each memory bank is with a processor, concurrent execution is assumed. Program segments as well as data layout are distributed in the disambiguation process. In other words, the design of RAW targets scalable processor level parallelism, which contrasts to our work of instruction level parallelism intrinsically.

In the data and memory management literature, manipulation of arrays is generally at a high level. Source analysis or transformation techniques are applied well before assembly code translation. Some examples are the heterogeneous memory discussion in [41][42]. For general discussions regarding space, such as storage estimation, sharing of physical locations, lifetime analysis, and variable dependencies, arrays are examined in high level code quite often [40]. This fact demonstrates the efficacy to explore arrays at the high level language level, which we explore in this chapter as well.

6.3. Problem Formulation

Given a set of variables along with the size, we would like to calculate an optimum bank assignment. It is assumed that there are two homogeneous memory banks of equal capacity. This assumption is practical and similar architectures can be found in products such as the Motorola DSP56000, NEC μPD77016, and Analog Devices ADSP2100. Each bank can be independently accessed in parallel. Such parallelism for memories enhances execution performance. The problem then is to compute a bank assignment with maximum simultaneous memory accesses and minimum capacity requirement.

To demonstrate an overview of our work, an SDF-based software synthesis process is drawn in Figure 6.2. First, applications are modeled by SDF graphs, which are effective at representing multirate signal processing systems. Scheduling algorithms are then employed to calculate a proper actor execution order. The order has significant impact on actor communication buffer sizes and makes scheduling a non-trivial task. For scheduler selection, APGAN and GDPPO are proven to reach certain lower bounds on buffer size if

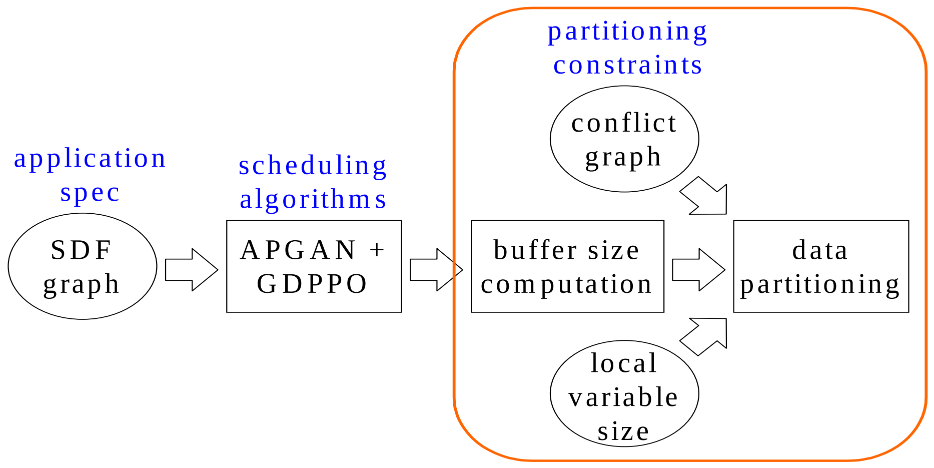


Figure 6.2 Overview of SDF-based software synthesis.

they are achievable [7]. Possible simultaneous memory accesses, partitioning constraints in the figure, together with actor communication buffer sizes and local state variable sizes in actors are then passed as inputs to data partitioning. Our focus in this chapter is on the rounded rectangle part in Figure 6.2.

One important consideration is that scalar variables are not targeted in this research. Mostly, they are translated to registers or immediate values. Compilers generally do so to promote execution performance. Memory cost is primarily due to arrays or consecutive data. As we described earlier, therefore, scalar variables and registers are often managed together. Since we are addressing data partitioning at the system design level, consecutive-data variables at a higher level in the compilation process are our major concern in this work.

The description above can be formalized in terms of graph theory. First, we build an undirected graph, called a **conflict graph** (e.g., see [12][31] for elaboration), $G = (V, E)$, where V and E are sets of nodes and edges respectively. Variables are represented by nodes and potential parallel accesses by edges. There is an integer weight $w(v)$ associated with every node $v \in V$. The value of a weight is equal to the size of the corresponding variable.

The problem of bank assignment, with two banks, is to find a disjoint bi-partition of nodes, P and Q, with each associated to one bank. The subset of edges with end nodes falling in different partitions is called an **edge cut**. Edge cut χ is formally defined as $\chi = \{e \in E | (v' \in P) \wedge (v'' \in Q)\}$ where v' and v'' are endpoints of edge e. Since a partition implies a collection of variables assigned to one bank, elements of the edge cut are the parallel accesses that can be carried out. Conversely, parallel accesses are not permissible for edges that do not fall in the edge cut. We should note that edges in the conflict graph represent possible parallelism in the application, and are not always achievable in any solution. Therefore, the objective is to maximize the cardinality of χ.

The other goal is to find minimum capacity requirement. Because of homogeneous size in both banks, we aim at storage balancing as well. That is, the capacity requirement is exactly the largest space occupation of either bank. Let $C(P)$ denote the total space cost of bank P. It is defined as

$$C(P) = \sum_{\forall v \in P} w(v).$$

Cost $C(Q)$ is defined in the same way. The objective is to reduce the capacity requirement M under the constraints of $C(P) \le M$ and $C(Q) \le M$. In summary, we have two objectives to optimize the partitioning problem:

$$min(M) \text{ and } max|\chi|. \tag{6.1}$$

Though there are two goals, priority is given to $max|\chi|$ in decision making. When there are contradictions between the objectives, a solution with maximum parallelism is chosen. In the following, we work on parallelism exploration first and then on examination of capacity. Alternatively, parallelism can be viewed as a constraint to fit. This is the view taken in the ILP approach proposed later.

Variables can be categorized as two types. One is actor communication buffers and the other is state variables local to actors. Buffers are for message passing in dataflow models and management over them is important for multirate applications. SDF offers several advantages in buffer management. One example is space minimization under single appearance scheduling constraint. As mentioned earlier, the APGAN and GDPPO algorithms in [7] are proven to reach a lower bound on memory requirements under certain conditions. However, buffer size is not our primary focus in this work though we do apply APGAN and GDPPO as part of the scheduling phase. The other type, state variables, is local and private to individual actors. State variables act as internal temporary variables or parameters in implementation and are not parts of dataflow expression. In this paper, however, variables are not distinguished by types. Types are merely mentioned to explain the source of variables in dataflow programs.

6.4. Observations on Benchmarks

We have found that benchmarks, in the form of conflict graphs, derived from several applications have sparse connections. For example, a convolution actor involves only two arrays in simultaneous accesses. Other variables to maintain temporary values, local states, loop control, etc. are not apparently beneficial, though no harm is inflicted either, if they are accessed in parallel.

Connected components (abbreviated as **CGCC, Conflict Graph Connected Component**) of benchmarks also tend to be acyclic and bipartite. We say a graph is **bipartite** if the node set can be partitioned into two sets such that all edges are with end nodes falling in distinct node partitions. This is good news to graph partitioning. Most of them have merely two nodes with a connecting edge. For those a bit more complicated, short chains account for the major structure. There are also many trivial CGCCs containing one node each and no edges. Typical topologies of CGCCs are illustrated in Figure 6.3 and an example is provided in Figure 6.4. Variable *singalIn* in Figure 6.4 is an input buffer of the actor and its size is to be decided by schedulers. Variables like *hamming* and *window* are arrays internal to the actor. For each iteration of the loop, *signalIn* and *hamming* are fetched to complete the multiplication and qualify for parallel accesses.

The characteristic of loose connectivity appears to high level relationships among consecutive-data variables. Though we did not investigate characteristics of the connectivity in the scalar case, it is believed that the connectivity is much more complicated than what we observe for arrays. In [12], though, the authors mention that the whole graph may not be connected and multiple connected components exist, and a heuristic approach is adopted to cope with complex topologies of the connected components. The topologies derived in [51] should be even more intricate because more factors are considered. Read-

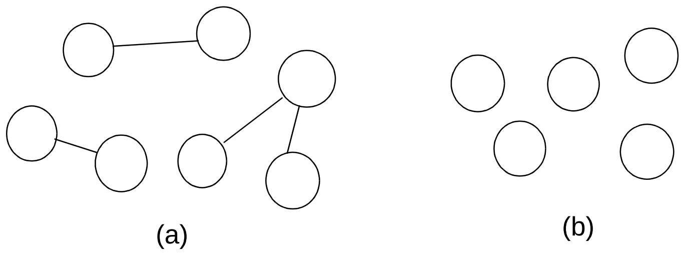


Figure 6.3 Features of conflict graph connected components extracted from realapplications. (a) short chains (b) trivial components, single nodes without edges.

```
#define N 320
float hamming[N];
float window[N];
for (m = 0; m < N; m++) {
    window[m] =
        signalIn[m] * hamming[m];
}
```

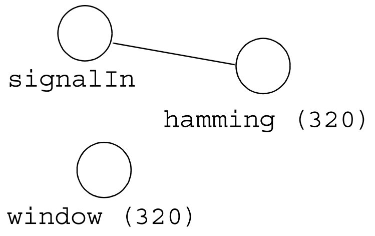


Figure 6.4 A conflict graph example of an actor that windows input signals.

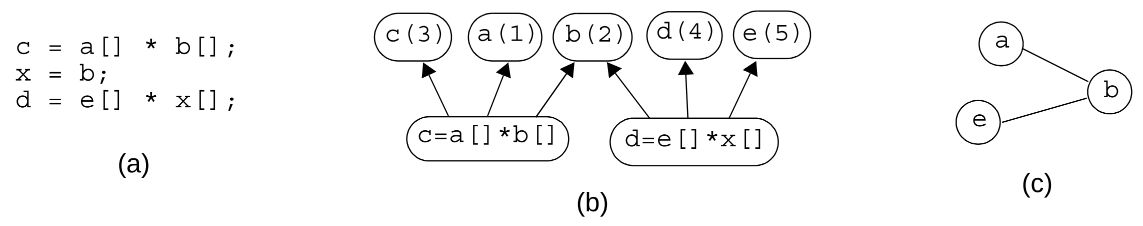
ers are reminded here once again that only arrays are focused on at high level in our context of combined memory minimization and data partitioning.

Another contribution to loose connectivity lies in the nature of coarse-grain dataflow graphs. Actors of dataflow graphs communicate with each other only through communication buffers represented by edges. State variables internal to an actor are inaccessible and invisible to that of other actors. This feature forces modularity of dataflow implementation and causes numerous CGCCs. Moreover, except for communication buffer purposes, any global variables are disallowed. This prevents their occurrences in arbitrary numbers of routines and hence reduces conflicts across actors. Furthermore, based on our observations, communication buffers contribute to conflicts mostly in read accesses. In other words, buffer writing is usually not found in parallel with other memory accesses. The phenomenon is natural in single assignment semantics.

In [4], to facilitate memory bank disambiguation, information about aliased memory references is required. To determine aliases, pointer analysis is performed. The analysis results are then represented by a directional bipartite graph. The graph nodes could be memory reference operations or physical memory locations. The edges are directed from operations to locations to indicate dependencies. The graph is partitioned into connected components, called **Alias Equivalence Classes (AEC)**, where any alias reference can only occur in a particular class. AECs are assigned to RAW tiles so that tasks are done independently without any inter-tile communication. Figure 6.5 is given to illustrate the concept of AECs. For the sample C code in (a), variable *b* is aliased by *x*. Memory locations and referencing code are expressed by a directional bipartite graph in (b). Parenthesized integers next to variables are memory location numbers (or addresses) and *b* and *x* are aliased to each other with identical location number 2. The connected component in (b) is the corresponding AEC of (a).

A relationship exists between AEC and CGCC, keeping in mind that conflict edges indicate concurrent accesses to two arrays. All program instructions issuing accesses to either array are grouped to an identical alias equivalence class. Therefore, both arrays can be found exclusively in that class. In other words, the node set of a CGCC can appear only in a certain single AEC instead of multiple ones. Take Figure 6.5 as an example. The node set in (c) can be found only in the node set of (b). For an application, therefore, the number of CGCCs is greater than or equal to that of AECs. The relationship between AEC and CGCC makes it promising in the automatic derivation of conflict graphs. This is an interesting topic for further work.

It is found in [4] that practical applications have several AECs. According to the relationship revealed in the previous paragraph, the number of CGCCs is bigger. If the modularity of dataflow semantics is considered, the number is even bigger. The fact of multiple AECs backs our discovery of numerous CGCCs and loose connectivity. However, the counts of AEC are not related to the simple topology, as demonstrated in Figure



```
c = a[] * b[];
x = b;
d = e[] * x[];
```

Figure 6.5 Example of the relationship between AECs and CGCCs.
(a) sample C code (b) AEC (c) CGCC.

6.3, of CGCC. Due to the feasibility of reducing CGCC from AEC, we believe that the graph structure of CGCC is much simpler than that of AEC.

6.5. Algorithms

In this section, three algorithms are discussed. The first one is a **0/1 ILP** approach, where all ILP variables are restricted to values 0 or 1. The second one is a coloring method, which is a typical strategy from the relevant literature. The third one is a greedy algorithm that is motivated by our observations on the structure of practical, SDF-based conflict graphs.

6.5.1. ILP

In this subsection, a 0/1 ILP strategy [3] is proposed to solve benchmarks with bipartite structure. Constraint equations are made for the bipartite requirement. If the conflict graph is not bipartite, it is rejected as failure. Fortunately, most benchmarks are bipartite according to our observations. On the other hand, the objective $min(M)$ in Equation (6.1) is translated to minimizing space cost difference, $min|C(Q) - C(P)|$, due to ILP restriction on single optimization equation. For each array u, there is an associated bank assignment b_u to be decided and $b_u \in \{0, 1\}$. Values of b_u denote banks, say B_P and B_Q respectively. A constant integer z_u denotes the size of array u. Memory parallelism constraints $b_u + b_y = 1$ are imposed if arrays u and y are to be accessed simultaneously and these constraints also act as the bipartite requirement. The constraints guarantee that distinct banks are assigned to the variables. Let us denote D as the capacity exceeding amount of bank B_Q beyond B_P. That is,

$$D = \sum_{\forall y \in B_Q} z_y - \sum_{\forall u \in B_P} z_u .$$

This equation can be further decomposed as follows.

$$D = \sum_{\forall y \in B_Q} z_y \cdot 1 + \sum_{\forall u \in B_P} z_u \cdot (0 - 1)$$

$$= \sum_{\forall y \in B_Q} z_y b_y + \sum_{\forall u \in B_P} z_u (b_u - 1)$$

$$= \sum_{\forall y} z_y b_y + \sum_{\forall u} z_u (b_u - 1)$$

$$= \sum_{\forall u} (z_u b_u + z_u (b_u - 1))$$

$$= \sum_{\forall u} z_u (2b_u - 1)$$

Finally, we end up with

$$D = 2\sum_{\forall u} z_u b_u - \sum_{\forall u} z_u .$$

Since the goal is to minimize the absolute value of D, one more constraint $D \geq 0$ is also required.

6.5.2. Two-Coloring and Weighted Set Partitioning

A traditional coloring approach is partially applicable for our data partitioning problem in Equation (6.1). If colors represent banks, a bank assignment is achieved while the coloring is done. Though minimum coloring is an NP-hard problem, it becomes polynomialy solvable for the case of two colors [19]. However, using a two-coloring approach, only the problem of simultaneous memory access is handled. Balancing of memory space costs is left unaddressed.

To cover space cost balancing, it is necessary to incorporate an additional algorithm. Among those integer set or weighted set problems that are similar to balancing costs, **weighted set partitioning** is chosen in our discussion because it searches for a solution with exact balancing costs. Weighted set partitioning states that: given a finite set A and a size $s(a) \in Z^{+}$ for each $a \in A$, is there a subset $A' \subseteq A$ such that

$$\sum_{a \in A'} s(a) = \sum_{a \in A - A'} s(a) ?$$

This problem is NP-hard [19]. If conflicts are ignored, balancing space costs can be reduced from weighted set partitioning and therefore balancing space costs with conflicts considered is NP-hard as well.

In the remainder of this sub-section, we establish the NP-hardness of the data partitioning problem addressed. As described earlier, data partitioning involves both bi-partitioning a graph and balancing of node weights. In other words, it is a combination of graph 2-coloring and weighted set partitioning, where the second problem is NP-hard. Therefore, for simplicity, we only prove that balancing node weights is NP-hard. Equivalently, we establish NP-hardness for the special case of data partitioning instances that have no conflicts.

The problem of space balancing is defined in Equation (6.3) and the objective is to minimize the capacity requirement M. The decision version of the optimization problem is to check whether both $C(P) \leq M$ and $C(Q) \leq M$ hold for a given constant integer M.

In the following paragraphs, we demonstrate the NP-hardness reduction from a known NP-hard problem, weighted set partitioning.

Weighted set partitioning states that: given a finite set A and a size $s(a) \in Z^+$ for each $a \in A$, is there a subset $A' \subseteq A$ such that

$$\sum_{a \in A'} s(a) = \sum_{a \in A - A'} s(a) \quad ? \tag{6.2}$$

The decision version of our space balancing problem can be rewritten as: Given a set of arrays U, the associated size $z(u) \in Z^+$ for every $u \in U$, and a constant integer $M > 0$, is there a subset $U' \subseteq U$ such that

$$\sum_{u \in U'} z(u) \leq M \text{ and } \sum_{u \in U - U'} z(u) \leq M \quad ? \tag{6.3}$$

Now given an instance (A, s) of weighted set partitioning, we derive an instance of space balancing by first setting

$$M = \left\lfloor \frac{\sum_{a \in A} s(a)}{2} \right\rfloor . \tag{6.4}$$

Then, for every element $a \in A$, we can have a corresponding array u and U is the set of all u. Moreover, $z(u) = s(a)$ for each corresponding pair of u and a. If a subset A' exists to satisfy Equation (6.2), the corresponding U' also makes Equation (6.3) true. If a subset of arrays U' exists for Equation (6.3), the corresponding A' also makes (6.26.2) true because

$$\sum_{a \in A'} s(a) + \sum_{a \in A - A'} s(a) = \sum_{a \in A} s(a) , \tag{6.5}$$

where

$$\sum_{a \in A'} s(a) \leq \left\lfloor \frac{\sum_{a \in A} s(a)}{2} \right\rfloor \text{ and } \sum_{a \in A - A'} s(a) \leq \left\lfloor \frac{\sum_{a \in A} s(a)}{2} \right\rfloor . \tag{6.6}$$

The above arguments justify the necessary and sufficient conditions of the reduction from Equation (6.2) to (6.3).

6.5.3. SPF — A Greedy Strategy

In this section, we develop a low-complexity heuristic called **SPF (Smallest Partition First)** for the heterogeneous-size data partitioning problem. Although 0/1 ILP calculates exact solutions, its complexity is non-polynomial, and therefore its use is problematic within intensive design space exploration loops, and for very large applications, it may become infeasible altogether. Coloring and weighted set partitioning each compute partial results. In addition, the efficacy of coloring is on heavily connected graphs. With the observations of loose connectivity in practice, coloring does not offer much contribution. A combination of coloring and weighted set partitioning would be interesting and is left as future work. In this article, the heuristic of SPF is proposed instead that is tailored to the restricted nature of SDF-based conflict graphs. The results and performance will be compared to that of 0/1 ILP and coloring in the next section.

A pseudocode specification of the SPF greedy heuristic is provided in Figure 6.6. Connected components or nodes with large weights are assigned first to the bank with least space usage. Variables of smaller size are gradually filled to narrow the space cost gap between banks. The assignment is also interleaved to maximize memory parallelism. Note that the algorithm is able to handle an arbitrary number of memory banks, and is applicable to non-bipartite graphs. Thus, it provides solutions to any input application with arbitrary bank count.

The SPF algorithm achieves a low computational complexity solution. In the pseudocode specification, the procedure **AlternateAssignment** performs the major function of data partitioning and is called exactly once for every node in a recursive style. First, the bank array $B[1 \dots K]$ is sorted in AlternateAssignment according to present storage usage. After that, internal edges linked to the input node are examined for every bank, keeping in mind that only cut edges are desired. The last step is querying the assignment

```
procedure SPFDataPartitioning
input: a conflict graph $G = (V,E)$ with integer node weights $W(V)$
    and an integer constant $K$ representing the number of banks.
output: node partitions $B[1...K]$.

set an array $B[1...K]$ representing $K$ node partitions.
let $C$ be the set of connected components of $G$.
sort $C$ decreasingly by total node weights.
for (each connected component $c \in C$) {
    get the node $v \in c$ with the largest weight.
    call AlternateAssignment($v$).
}
output array $B[1...K]$.


procedure AlternateAssignment
input: a node $v$.
set a boolean variable assigned to false.
sort $B[1...K]$ increasingly by total node weights.
for (each node partition $B[i]$) {
    if (no $u \in B[i]$ such that $u$ is a neighbor of $v$) {
        add $v$ to $B[i]$.
        set assigned to true.
        quit the for loop.
    }
}
if (assigned has value false) {
    add $v$ to $B[1]$, the node partition
        with the smallest total node weight.
}
for (each neighbor node $b$ of $v$)
    if (node $b$ has not been added to any of $B[1...K]$) {
        call AlternateAssignment($b$).
    }
}
```

Figure 6.6 The SPF data partitioning algorithm.

of neighbor nodes and a recursive call. Therefore, the complexity of AlternateAssignment is $O(K log K + KN + N)$, where N denotes the largest node degree in the conflict graph. Though the practical complexity of N can be $O(1)$ according to our observations, the worst case is of $O(|V|)$ complexity. In our assumption, K is a constant provided by the system. For the whole program execution, all calls to AlternateAssignment contribute $O(|V|^2)$ in worst case and $O(|V|)$ in practice. The remaining computations in SPF include strongly connected component decomposition, sorting connected components by total node weights, and building neighbor node lists. Their complexities are $O(|V| + |E|)$ [54], $O(|C| log|C|)$, and $O(|E|)$, respectively ($|C|$ denotes the number of connected components in the conflict graph.). In summary, the overall computational complexity is $O(|V|^2)$ in worst case and practically $O(max(|V| + |E|, |C| log|C|))$ for several real applications.

6.6. Experimental Results

Our experiments are performed for all three algorithms: ILP, 2-coloring, and our SPF algorithm. Since all conflict graphs from our benchmarks are bipartite, every edge falls in the edge cut and memory parallelism is maximized by all three algorithms. Therefore, only the capacity requirement is chosen as our comparison criteria. Improvement is evaluated for SPF over 2-coloring, a classical bank assignment strategy. Performance of SPF is also compared to that of ILP to give an idea of the effectiveness of SPF. For ILP computation, we use the solver OPBDP which is an implementation based on the theories of [3].

To decide bank assignment for coloring, the first node of a connected component is always fixed to the first bank and the remaining nodes are typically assigned in an alternate way because of the commonly-found bipartite graph structure. The order in which the algorithm traverses the nodes of a graph is highly implementation dependent and the result depends on this order. Thus, some results may become better while others may become worse if another ordering is tried. However, the average improvement of SPF is still believed to be high since numerous applications have been considered in the experiments with our implementation.

A summary of the results is given in Figure 6.7. The first column lists all the benchmarks that were used in our experiments. The second and third columns provide the number of variables and parallel accesses, respectively. Since the benchmarks are in the format of conflict graphs, those two columns represent node and edge counts, too. The fourth to sixth columns give the bank capacity requirement for each of the three algorithms. Capacity reduction for SPF over 2-coloring is placed in the last column as an improvement measure.

Most of the benchmarks are extracted from real applications in the Ptolemy environment [17]. Ptolemy is a design environment for heterogeneous systems and many examples of real applications are also included. A brief description of all the benchmarks follows. Two of them are sample rate conversion between CD and DAT devices, *cd2dat* and *dat2cd*. Filter bank examples are *filterBankNU*, *filterBankNU2*, *filterBankPR*, and *filterBankSub*. The first two are two-channel non-uniform filter banks with different depths. The third one is an eight-channel perfect reconstruction filter bank, while the last one is for four-channel subband speech coding with APCM. Modems of BPSK and QPSK are *bpsk10-100* and *qpsk10-100* with various intervals. A telephone channel simulation is represented by *telephone*. Filter stabilization using cepstrum is in *cep*. An analytic filter with sample rate conversion is *analytic*. A satellite receiver abstraction, *satellite*, is obtained from [45]. Because *satellite* is just an abstraction without implementation details, reasonable synthetic conflicts are added according to our benchmark observations.

Figure 6.7 demonstrates the performance of SPF. Not only does it generate less capacity requirement than the classical coloring method does, but also the results are almost equal to the optimality evaluated by ILP. The polynomial computational complex-

	variable counts	conflict counts	coloring	SPF	ILP	improvement (%)
analytic	9	3	756	448	448	40.7
bpsk10	22	8	140	90	90	35.7
bpsk20	22	7	240	156	156	35.0
bpsk50	22	8	300	228	228	24.0
bpsk100	22	8	500	404	404	19.2
cep	14	2	1602	1025	1025	36.0
cd2dat	15	7	1459	1343	1343	8.0
dat2cd	10	5	412	412	412	0.0
discWavelet	92	56	1000	999	999	0.1
filterBankNU	15	10	196	165	164	15.8
filterBankNU2	52	27	854	658	658	23.0
filterBankPR	92	56	974	851	851	12.6
filterBankSub	54	32	572	509	509	11.0
qpsk10	31	16	173	146	146	15.6
qpsk20	31	14	361	277	277	23.3
qpsk50	31	16	453	426	426	6.0
qpsk100	31	16	803	776	776	3.4
satellite	26	9	1048	771	771	26.4
telephone	11	2	1633	1105	1105	32.3
Average						19.4

Figure 6.7 Summary of the data partitioning exeprimental results.

ity (see subsection 6.5.3) is also lower than the exponential complexity of ILP. In our ILP experiments on a 1GHz Pentium III machine, most of the benchmarks finish within a few seconds. However, *discWavelet* and *filterBankPR* spend several hours to complete. In contrast, SPF finishes in less than ten seconds for all cases. In summary, SPF is effective both in the results and the computation time.

Chapter 7. Conclusion and Future Work

In this thesis, a number of important memory and performance optimization problems are addressed for translating high-level representations of signal processing applications into embedded software implementations. The problems studied in the thesis involve several inter-related features, and therefore an integrated approach is required to solve them effectively. This thesis proposes such an integrated approach, and develops the approach through formal problem formulations, in-depth theoretical analysis, and extensive experimentation.

The performance of the techniques proposed in the thesis is generally application-dependent, and some of the techniques will perform relatively better or worse for certain applications. The experiments developed in this thesis demonstrate how each technique can be useful for a broad range of useful DSP applications and subsystems. On the other hand, due to the highly multi-objective nature implementations in our targeted domain, this thesis has addressed particular regions of the design space with different techniques that are streamlined for these regions (e.g., in contrast to proposing a single generic technique that is to address the entire design space). Using our methods, designers can experiment with the different techniques to find the best combination for a particular set of implementation constraints and optimization objectives. This kind of interactive optimization and design space exploration — in which the expertise and preferences of the designer are flexibly integrated with a set of tools for synthesis and optimization — is common for design of embedded systems.

The problems investigated in this thesis are summarized in the remainder of this chapter, and several directions for future work are described.

7.1. Nested Procedure Synthesis

In this work, we have developed an efficient method for applying subroutine call instantiation of module functionality when synthesizing embedded software from an SDF specification. This approach provides for significantly lower buffer sizes, with polynomially bounded procedure call overhead, than what is available for minimum code size, inlined schedules. This approach also provides for significantly lower code space requirements than efficiently looped minimum buffer schedules with not much buffer cost overhead. This recursive decomposition approach thus provides an efficient means for integrating subroutine-based module instantiation into the design space of DSP software synthesis. We develop metrics for characterizing a certain form of uniformity in SDF schedules, and show that the benefits of the proposed techniques increase with decreasing uniformity.

Directions for future work include integrating the procedural implementation approach in Chapter 3 with existing techniques for inlined implementation. For example, different subgraphs in an SDF specification may be best handled using different techniques, depending on application constraints and subgraph characteristics (e.g., based on uniformity, as defined in Chapter 3, and actor granularity). Integration with other strategies for buffer optimization such as phased scheduling [23] and buffer merging [36] are also useful directions for further investigation.

7.2. Block Processing Optimization

In Chapter 4, we have first demonstrated the advantages of block processing implementation of DSP kernel functions. Then we have examined the integrated optimization problem of block processing, code size minimization, and data space reduction. We have shown that this problem can be modeled through a nonlinear programming formulation. However, due to the intractability of nonlinear programming, we have developed two efficient heuristics that are computationally efficient. We have evaluated both the Greedy-FAPL and FARM algorithms, and our results demonstrate that they consistently derive high quality results. Chapter 4 has presented a number of concrete examples and additional bodies of experimental results that provide further insight into the relationships among block processing, memory requirements, and performance optimization for DSP software.

7.3. Parameterized Looped Schedules

Chapter 5 has focused on the motivation for formally examining broader classes of looped schedules, and on the definition and application of parameterized, constant-update looped schedules (PCLSs) for generating static execution sequences (programs). PCLSs go beyond traditional static looped schedules by making the management of loop counters more explicit. This greatly enlarges the space of execution sequences that can be compactly represented, while requiring low overhead in most implementation contexts. As the terminology in Chapter 5 suggests, there are possibilities for further enriching the classes of looped schedules under investigation. For example, one might consider a more general class of schedules in which output values computed by “instructions” can be captured and used in the initialization or updating of loop counts, or in which the index update function can be more complex.

7.4. Data Partitioning

Bank assignment for arrays has great impact both on parallel memory accesses and memory capacity. Traditional bi-partitioning or two coloring strategies for scalar variables cannot be well adapted to applications with arrays. The variety of array sizes complicates memory management especially for typical embedded systems with stringent storage capacity. We propose an effective approach to jointly optimize memory parallelism and capacity when synthesizing software from dataflow graphs.

Surprisingly but reasonably, high level analysis presents a distinctive type of graph topology for real applications. Graph connections are sparse and connected components are in the form of chains, bipartite connected components, or trivial singletons.

Some possible future works follow. Our SPF algorithm generates results quite close to optimality. We are curious about the efficacy to graphs with arbitrary topology. Sparse connections found in dataflow models also arouses our interests in the applicability to procedural languages like C. Integration of high and low level optimization is also promising. An integrated optimization scheme involving arrays, scalar variables, and registers is a particularly useful target for further study. Automating conflict information through alias equivalence class calculation is a possible future work as well. Another potential work is to reduce storage requirements further by sharing physical space among variables whose lifetimes do not overlap.

BIBLIOGRAPHY

[1] M. Ade, R. Lauwereins, and J. A. Peperstraete, “Data Memory Minimization for Synchronous Data Flow Graphs Emulated on DSP-FPGA Targets,” *Proceedings of Design Automation Conference (34th DAC)*, Anaheim, California, pp.64-69, 1997.

[2] O. Avissar, R. Barua, and D. Stewart, “Heterogeneous memory management for embedded systems,” *ACM International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES)*, Atlanta, Georgia, pp.34-42, 2001.

[3] Peter Barth, *Logic-Based 0-1 Constraint Programming*, Kluwer Academic Publishers. 1996.

[4] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal, “Compiler Support for scalable and Efficient Memory Systems,” *IEEE Transactions on Computers*, 50(11):1234-1247, November 2001.

[5] B. Bhattacharya and S. S. Bhattacharyya, “Parameterized dataflow modeling for DSP systems,” *IEEE Transactions on Signal Processing*, 49(10):2408-2421, October 2001.

[6] S. S. Bhattacharyya, R. Leupers, and P. Marwedel, “Software synthesis and code generation for DSP,” *IEEE Transactions on Circuits and Systems -- II: Analog and Digital Signal Processing*, 47(9):849-875, September 2000.

[7] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, *Software Synthesis from Dataflow Graphs*, Kluwer Academic Publishers, 1996.

[8] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, “Optimal Parenthesization of Lexical Orderings for DSP Block Diagrams,” *Proceedings of the International Workshop on VLSI Signal Processing*, Sakai, Osaka, Japan, 1995.

[9] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete, “Cyclo-static dataflow”, *IEEE Transactions on Signal Processing*, 44(2):397-408, February 1996.

[10] R. E. Blahut, *Fast Algorithms for Digital Signal Processing*, Addison-Wesley, 1985.

[11] J. T. Buck and R. Vaidyanathan, “Heterogeneous modeling and simulation of embedded systems in El Greco,” *Proceedings of the International Workshop on Hardware Software Codesign*, San Diego, California, pp.142-146, May 2000.

[12] J. Cho, Y. Paek, and D. Whalley, “Efficient Register and Memory Assignment for Non-orthogonal Architectures via Graph coloring and MST Algorithms,” *LCTES’02-SCOPES’02*, pp.130-138, Berlin, June 2002.

[13] K. Cooper, N. McIntosh, “Enhanced code compression for embedded RISC processors,” *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp.139-149, 1999.

[14] M. Cubric and P. Panangaden, “Minimal memory schedules for dataflow networks,” *Proceedings of the 4th Intl. Conf. on Concurrency Theory (CONCUR ‘93)*, Lecture Notes in Computer Science 715:368-383, August 1993.

[15] S. Debray, W. Evans, R. Muth, B. de Sutter, “Compiler techniques for code compression,” *ACM Transactions on Programming Languages and Systems*, pp. 378-415, 2000.

[16] S. Derrien, A. Turjan, C. Zissulescu, B. Kienhuis, “Deriving efficient control in Kahn process networks,” *International Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS)*, 2003.

[17] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, “Taming heterogeneity — the Ptolemy approach,” *Proceedings of the IEEE*, January 2003.

[18] S. Frohlich and B. Wess, “Integrated Approach to Optimized Code Generation for Heterogeneous-Register Architectures with Multiple Data-Memory Banks,” *IEEE 14th Annual ASIC/SOC Conference*, pp.122-126, Arlington, September 2001.

[19] M. R. Garey and D. S. Johnson, *Computers and Intractability*, W. H. Freeman. 1979.

[20] L.Guerra, M.Potkonjak, J.Rabey, “System-level design guidance using algorithm properties,” *IEEE workshop on VLSI in signal processing*, pp.73-82, 1994

[21] C. Hsu, M. Ko, and S. S. Bhattacharyya, “Software synthesis from the dataflow interchange format,” *International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, pp.37-49, 2005.

[22] I. Hong, M. Potkonjak, and M. Papaefthymiou, “Efficient block scheduling to minimize context switching time for programmable embedded processors,” *Design Automation for Embedded Systems*, 4(4):311-327, Kluwer Academic Publishers, 1999.

[23] M. Karczmarek, W. Thies, and S. Amarasinghe, “Phased scheduling of stream programs,” *Proceedings of Languages, Compilers, and Tools for Embedded Systems (LCTES’03)*, pp.103-112, June 2003.

[24] D. E. Knuth, *Seminumerical algorithms, 2nd edition, The Art of Computer Programming Volume 2*, Addison-Wesley, Reading, MA, 1981.

[25] M. Ko, P. K. Murthy, and S. S. Bhattacharyya, “Beyond single appearance schedules: efficient DSP software synthesis using recursive procedure calls,” *ACM Transactions on Embedded Computing Systems (TECS)*, accepted, to appear in 2006.

[26] M. Ko, C. Shen, and S. S. Bhattacharyya, “Memory-constrained block processing optimization for synthesis of DSP software,” *International Conference on Embedded Computer Systems: Architectures, MOdeling, and Simulation*, accepted, to appear in 2006.

[27] M. Ko and S. S. Bhattacharyya, “Data partitioning for DSP software synthesis,” *International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, pp. 344-358, 2003

[28] K. N. Lalgudi, M. C. Papaefthymiou, and M. Potkonjak, “Optimizing computations for effective block-processing,” *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 5(3):604-630, July 2000.

[29] R. Lauwereins, M. Engels, M. Ade, and J. A. Peperstraete, “Grape-II: A system-level prototyping environment for DSP applications,” *IEEE Computer Magazine*, 28(2):35-43, February 1995

[30] E. A. Lee and D. G. Messerschmitt, “Synchronous dataflow,” *Proceedings of IEEE*, 75:1235–1245, September 1987.

[31] R. Leupers and D. Kotte, “Variable Partitioning for Dual Memory Bank DSPs,” *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, Salt Lake City, May 2001.

[32] R. Leupers and P. Marwedel, “Time-constrained code compaction for DSP's,” *IEEE Transactions on VLSI Systems,* 5(1):112-122, March 1997.

[33] S. Y. Liao, *Code Generation and Optimization for Embedded Digital Signal Processors*, PhD thesis, MIT, Cambridge, Massachusetts, June 1996.

[34] E. Memin, T. Risset, “On the study of VLSI derivation for optical flow estimation,” *International Journal of Pattern Recognition and Artificial Intelligence,* 2000.

[35] Motorola, *DSP56000/DSP56001 Digital Signal Processor User’s Manual*, Motorola Inc., Phoenix, AZ, 1990.

[36] P. K. Murthy and S. S. Bhattacharyya, “Buffer merging: a powerful technique for reducing memory requirements of synchronous dataflow specifications,” *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 9(2):212-237, April 2004.

[37] P. K. Murthy and S. S. Bhattacharyya, “Shared buffer implementations of signal processing systems using lifetime analysis techniques,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(2):177-198, February 2001.

[38] P. K. Murthy, E. G. Cohen, and S. Rowland, “System Canvas: A new design environment for embedded DSP and telecommunication systems,” *Proceedings of the International Workshop on Hardware/Software Co-Design*, April 2001.

[39] H. Oh, N. Dutt, S. Ha, “Single appearance schedule with dynamic loop count for minimum data buffer from synchronous dataflow graphs,” *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pp. 157-165, September 2005.

[40] P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, and P. G. Kjeldsberg, “Data and Memory Optimization Techniques for Embedded Systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 6(2):149-206, April 2001.

[41] P. R. Panda, N. D. Dutt, and A. Nicolau, “On-Chip vs. Off-Chip Memory: The Data Partitioning Problem in Embedded Processor-Based Systems,” *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 5(3):682-704, July 2000.

[42] P. R. Panda, “Memory Bank customization and Assignment in Behavioral Synthesis,” *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 477-481, San Jose, November 1999.

[43] K. K. Parhi, *VLSI digital signal processing systems: design and implementation*, A Wiley-Interscience publication, 1999.

[44] D. B. Powell, E. A. Lee, and W. C. Newman, “Direct Synthesis of Optimized DSP Assembly Code from Signal Flow Block Diagrams,” *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, 5:23-26, March 1992.

[45] S. Ritz, M. Willems, and H. Meyer, “Scheduling for optimum data memory compaction in block diagram oriented software synthesis,” *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, 4:2651-2654, May 1995.

[46] S. Ritz, M. Pankert, V. Zivojnovic, and H. Meyer, “Optimum vectorization of scalable synchronous dataflow graphs,” *Proceedings of the International Conference on Application Specific Array Processors,* pp.285-296, October 1993.

[47] C. B. Robbins, *Autocoding Toolset software tools for automatic generation of parallel application software.* Technical Report of Management, Communications & Control Inc. (MCCI), 2002.

[48] M. A. R. Saghir, P. Chow, and C. G. Lee, “Exploiting Dual Data-Memory Banks in Digital Signal Processors,” *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp.234-243, October 1996.

[49] V. Sipkova, “Efficient Variable Allocation to Dual Memory Banks of DSPs,” *International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, Vienna, Austria, pp. 359-372, September 2003.

[50] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprettere, “System design using Kahn process networks: the Compaan/Laura approach,” *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*, February 2004.

[51] A. Sudarsanam and S. Malik, “Simultaneous Reference Allocation in Code Generation for Dual Data Memory Bank ASIPs,” *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 5(2):242-264, April 2000.

[52] W. Sung and S. Ha, “Memory Efficient Software Synthesis using Mixed Coding Style from Dataflow Graph,” *IEEE Transactions on VLSI Systems*, 8:522-526, October 2000.

[53] W. Sung, M. Oh, C. Im, and S. Ha, “Demonstration of hardware software codesign workflow in PeaCE,” *International Conference on VLSI and CAD*, October 1997.

[54] R. E. Tarjan, “Depth first search and linear graph algorithms,” *SIAM Journal on Computing*, 1(2):146-160, 1972

[55] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A language for streaming applications," *International Conference on Compiler Construction*, 2002.

[56] Q. Zhuge, B. Xiao, E. H.-M. Sha, and C. Chantrapornchai, "Efficient Variable Partitioning and Scheduling for DSP Processors with Multiple Memory Modules," *IEEE Transactions on Signal Processing*, 52(4):1090-1099, April 2002.

[57] E. Zitzler, J. Teich, and S. S. Bhattacharyya, "Multidimensional exploration of software implementations for DSP algorithms," *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, pp.83-98, February 2000.

[58] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, 23(3):337-343, 1977.

[59] V. Zivojnovic, S. Ritz, and H. Meyr, "Retimimg of DSP programs for optimum vectorization," *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, 2:19-22, 1994.