From Cyclo-Static Process Networks to Code Generation for Multidimensional Software Pipelining

Mohammed Fellahi

THÈSE DE DOCTORAT

SPECIALITE : Informatique

Ecole Doctorale d'Informatique de Paris-sud

Présenté par

Mohammed Fellahi

Sujet de la thèse

From Cyclo-Static Process Networks to Code Generation for Multidimensional Software Pipelining

Des réseaux de processus cyclo-statiques à la génération de code pour le pipeline multi-dimensionnel

Soutenue le 22 Avril 2011 devant les membres du jury:

Pr. Yannis	Manoussakis	Président
Pr. Pierre	Boulet	Rapporteur
Pr. Alain	Girault	Rapporteur
Pr. Albert	Cohen	Directeur de thèse
Pr. Daniel	Etiemble	Examinateur
M. Lionel	Lacassagne	Examinateur

Université Paris Sud

Résumé

Les applications de flux de données sont des cibles importantes de l'optimisation de programme en raison de leur haute exigence de calcul et la diversité de leurs domaines d'application: communication, systèmes embarqués, multimédia, etc. L'un des problèmes les plus importants et difficiles dans la conception des langages de programmation distinés à ce genre d'applications est comment les ordonnancer à grain fin à fin d'exploiter les ressources disponibles de la machine.

Dans cette thèse on propose un "framework" pour l'ordonnancement à grain fin des applications de flux de données et des boucles imbriquées en general. Premièrement on essaye de paralléliser le nombre maximum de boucles en appliquant le pipeline logiciel. Après on fusionne le prologue et l'épilogue de chaque boucle (phase) parallelisée pour éviter l'augmentation de la taille du code. Ce processus est un pipeline multidimensionnel, quelques occurrences (ou instructions) sont décalées par des itérations de la boucle interne et d'autres occurrences (instructions) par des itérations de la boucle externe.

Les expériences montrent que l'application de cette technique permet l'amélioration des performances, extraction du parallélisme sans augmenter la taille du code, à la fois dans le cas des applications de flux de données et des boucles imbriquées en général.

Abstract

Applications based on streams, ordered sequences of data values, are important targets of program optimization because of their high computational requirements and the diversity of their application domains: communication, embedded systems, multimedia, etc. One of the most important and difficult problems in special purpose stream language design and implementation is how to schedule these applications in a fine-grain way to exploit available machine resources.

In this thesis we propose a framework for fine-grain scheduling of streaming applications and nested loops in general. First, we try to pipeline steady state phases (inner loops), by finding the repeated kernel pattern, and executing actor occurrences in parallel as much as possible. Then we merge the kernel prolog and epilog of pipelined phases to move them out of the outer loop. Merging the kernel prolog and epilog means that we shift actor occurrences, or instructions, from one phase iteration to another and from one outer loop iteration to another, a multidimensional shifting.

Experimental shows that our framework can improve performance, parallelism extraction without increasing the code size, in streaming applications and nested loops in general.

Contents

Chapter 1

Introduction

1.1 Introduction

Many architectures have been designed and constructed to exploit the parallelism of streaming applications. Specialized hardware is used in the Cheops video processing system for streaming operation [7]. The Imagine architecture for media processing also takes into account stream operations but without specialized functional units [51]. Raw architecture may support stream codes because of its conception, replicated processing elements and a fast compiler-controlled interconnection network [65]. Other architectures exist too, like the Cell processor.
Despite all these architectures, few of them have compilers and languages for streaming application programming. Such a compiler or language should help the programmer to express stream computations in a natural way and without the need to have information about the target machine. The programmer also doesn't need to take care about stream operations scheduling; it is the compiler task. Language design and paradigms should be easy enough to make it popular, especially when the language users are from different scientific domains as stream applications.
An interesting question may be asked here: why don't we use general purpose languages, like the C language, when it is already easy and popular? General purpose languages are inadequate for stream programming: They don't provide an intuitive representation of streams which reduces readability, robustness and programmer productivity. Moreover because of the widespread parallelism and regular communication patterns of data, streams are left implicit in general purpose languages; compilers are not stream-conscious and can not perform stream specific optimizations. Also most general purpose languages are built for Von-Neumann architectures which is not suitable for streaming applications and their architectures [4].
Surprisingly, there are few special-purpose stream languages. They are theoretically sound but they are not practical enough to be used for the development of any streaming applications. Sometimes, developing a streaming application using one stream language is a research project. The reason is that streaming applications are not like general applications: the nature of streams affects directly their design and development. The data can be streamed in a static or dynamic way. This means the data flow can be uniform or not and model of actor (independent operations)

communication can be static or dynamic depending on whether it can change during the execution time or not. It depends on the system where the application is running. An example of that is: a mobile phone user leaves a station zone and goes to another one is an asynchronous event. So the application environment also affects the application design and implementation.
Theoretically, a lot of work and research have been done. Researchers have tried to classify different streaming applications according to their stream and communication natures and there are models of computation for them like Kahn networks, data flow networks, Synchronous Data Flow (SDF), Boolean Data Flow (BDF), Dynamic Data Flow networks (DDF) and others [42]. For instance, SDF is for uniform streams application and DDF is for non uniform streams applications. Finding the model that suits your application is not enough to develop it. Executing the applications means the execution of its different actors, each one a number of times at a number of instances. Actors are independent in that they depend only on their inputs. If there are enough input data, they can be executed. So do all actor execution sequences are valid schedules? Are there invalid actor orders? what are the constraints restricting this selection? Is it always possible to find a valid schedule? Should this schedule be found at compile time? Surprisingly target architecture affects the choice of a valid schedule. A compiled code in a machine may not be executed on another one. The execution of a streaming application means the consumption of resources and time. Schedule size and channel size depend directly on the free memory and buffer sizes. Extraction of parallelism means the exploitation of machine resources, including registers and operational units, as good as possible. If the application has time constraints, for example a real time application, the research for a valid schedule becomes more difficult. We are focusing on SDF application static scheduling, scheduling during compilation, and we try to answer the questions asked above.

1.2 State of the Art

So the problem is how to schedule SDF applications. As far as we know, 'StreamIt' and 'Lucid Synchrone' are the latest works on Streaming Application languages. They come from very different classes of language paradigms. StreamIt is designed and developed to be an imperative language for streaming applications, exactly for SDF applications although StreamIt researchers have started to look at DDF applications. Its goals are to provide a high-level stream abstraction to improve programmer productivity. Also it is intended to be a general language for different stream target architectures which lets researchers work on portability too[4][57][58][59]. The scheduling task provides a coarse-grained schedule, Phased Scheduling [30][29], actors are coarse-grained actors and the schedule as well. Therefore, it doesn't take advantage of offered parallelism. Also, the application graph model (SDF) is restricted: it is built using a set of predefined structures [4][57][58][59] and not all SDF applications can be modeled using them. Lucid Synchrone is designed for streaming applications too but it is built on a functional language (OCAML), Lus-

tre family of stream languages. It is up to the programmer to use language syntax to build a synchronized application. Using keywords and scheduling instructions one by one provides a scheduled source code, but it is a hard task and it demands from the programmer to be well experienced with the language and obliges him to know the application and its environment very well to schedule it [12][47][46][10]. All difficult tasks are performed by the user himself which is not practical. Without forgetting that functional programming is not easy as well by itself. Other languages are cited in [55]. So how to schedule SDF applications in a fine-grain manner to profit from available machine resources?

1.3 Our Contributions

The goal of my work is to focus on the problem of streaming application scheduling. One of the most important and difficult problems in special purpose stream language design and implementation is to answer this question: how to schedule SDF applications in a fine-grain way to exploit available machine resources? *The main idea:* developing a fine-grained scheduling approach: an actor is a block of a few instructions without loop or branch statement, this is useful to build a fine-grained schedule and to exploit offered parallelism; an SDF schedule is an infinite execution of periodic sequence, steady state and this sequence schedule looks exactly like nested loops schedule. Hence, all our scheduling work is based on this similitude between nested loop schedule and SDF schedule, in this way we take advantage of nested loops scheduling methods and exploit machine parallelism. Also, loop scheduling and parallelism can take advantage of our proposed techniques and algorithms because they are applicable for both nested loops and steady state.

The contributions of this thesis are:

- A coarse-grain scheduling algorithm. It takes care of code and buffer size.
- A fine-grain scheduling technique INMS.
- Shifting idea: it breaks down dependence relations between actors to exploit parallelism.
- Prolog-epilog merging technique.
- Prolog-epilog merging with renaming.
- Code generation for prolog-epilog merging technique.

1.4 Thesis Overview

This thesis will address the problem of streaming application scheduling. It is divided into two parts. The first part will introduce the basic concepts. In chapter 2 we present Stream applications and their models of computation, especially the SD model. Chapter 3 is a survey on loop basic notions and its transformations that

are necessary for readers to understand our work. Chapter 4 describes "Dependence Removal Techniques" especially "Renaming" that we use to extent our scheduling technique, "Prolog-epilog Merging". In chapter 5 we present the Knapsack Problem because the "Prolog-epilog Merging" solution is based on solving this problem. The second part of the thesis focuses on our work. Chapter 6 is an overview of the problem. It cites different specifications and characteristics of applications we focus on. Then it presents "Imperfectly Nested Multidimensional Shifting(INMS)" framework. Chapter 7 talks about Pattern Tables Shifting, our first attempt to propose a fine-grained schedule for streaming applications. In chapter 8 we present the "Prolog-epilog Merging" scheduling technique and in chapter 9 we explain how to generate code for it. Chapter 10 shows different experiments and results. Finally, we conclude in chapter 11 and suggest future work.

Part I

Chapter 2

Stream Theory

2.1 Introduction

Applications based on streams, ordered sequences of data values, are becoming important and widespread because of their high computational requirements and the diversity of their domains: communication, embedded systems, multimedia, etc. In embedded domain, applications for hand-held computers, cell phones and DSPs are centered around streams of voice or video data. The stream abstraction is also fundamental to high performance applications such as intelligent software routers, cell phone base station and HDTV editing consoles.

One important property of these applications is that they are composed of independent repetitive operations. This takes the form of a pipeline. As data are streamed through the pipeline, each stage can operate concurrently on a different portion of the input.

We will see in this chapter some basic stream notions, in section 2.2 and 2.3 then in section 2.4 we will take a look at different models of computation, and we will finish by choosing one of them, the appropriate one for our case.

2.2 Basic Notions

Stream Processing: it refers to the active research area of a number of disparate systems such as dataflow systems, reactive systems, synchronous concurrent algorithms, signal processing systems and certain classes of real-time systems. They all deal with streams [55].
Stream: it is an infinite list of elements $a0, a1, a2, ...$ taken from some data of interest A which can be a set of integers, reals, booleans, or any other type, or stream itself[55].
Stream Processing Systems Basic Conceptual Model: a Stream Processing System (SPS) can be seen as collections of modules, also called actors or agents depending on the system and its specific model, that compute in parallel and that communicate data via channels. A module can be a source that passes data into the system, a filter (or agent) that performs atomic computations or a sink that passes data from

the system [55]. This basic model can be represented by a directed graph where nodes are modules and arcs are channels. Fig. 2.1 shows a typical SPS. The characteristic of modules and channels provide more specific models such as Kahn process networks, dataflow networks and others. The three main characteristics are :

(1) synchronous or asynchronous actors: actor are synchronized or not. Do they operate in synchronized manner with respect to other filters, or they compute with no synchronization?

(2) deterministic or non-deterministic actors: actor either do or do not compute a function.

(3) uni-directional or bi-directional channels.

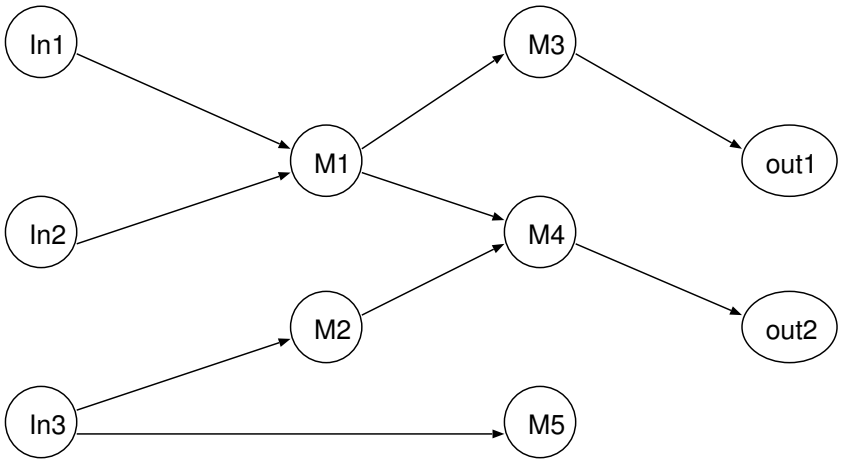


Fig. 2.1. Typical SPS

2.3 Streaming Applications

Applications that use stream abstractions are diverse but they have some common characteristics identifying them as " Streaming Applications" and making them distinct from other classes of programs [59]. These characteristics are as follows:

1. large streams of data: it is the most important characteristic of streaming applications. Streams are the input and the output data. They may be infinite. This makes a great difference between streaming applications and scientific codes which process a set of data with a great degree of data reuse.

2. independent stream filters: an Actor is the basic operational unit. If we look at a streaming computation as a sequence of transformations on the data streams, an actor will be the basic unit of this transformation. All communications between actors are explained in the graph. Actors do not communicate through shared variables. The streaming application is the composition of these filters in a stream graph where the output of a filter is connected to the input of another.

3. a stable computation pattern: generally the graph structure is static during the computation or precisely during the steady state sequence which is a predefined period ordering filter executions. In Synchronous Data Flow (SDF) case, the graph structure is always static.

4. occasional modification of stream structure: occasional change of the graph structure occurs because of some asynchronous events. These events will break the ordinary execution by introducing some new filters to the graph, omitting some others or modifying some data value, like, initialization. For instance, a software radio re-initializes a portion of stream graph when a user switches from AM to FM or when a network protocol changes from bluetooth to 802.11 during transmission.

5. occasional out-of-stream communication: in practice, filters may exchange another kind of data, a small amount of control information on an infrequent and irregular basis. For example to change the cell phone volume or printing an error message to a screen.

6. high performance expectations: many streaming applications, like in embedded system, have real time constraints and others have their own appropriate constraints depending on the streaming application kind. For example power consumption in mobile environment is a critical constraint. Also, memory requirement and code size in some specific architectures where memory size is not very big, in some specific embedded architectures where memory is very expensive (cache or ROM).

We are interested in this thesis in streaming applications that have a static graph, Synchrounous DataFlow (SDF) applications. They are presented in detail in section 2.4.3.

2.4 Models of Computation

As we have said above, characteristics of channels, actors and streams define the model of computation. We will not talk here about models, like reactive system model, where the channel is bi-directional. We will present only dataflow system models: kahn networks and its restricted models. Kahn Process Networks is the natural model for describing dataflow systems. It is for ADU-SPS (Asynchronous Determinate Unidirectional SPS) and generally dataflow systems are ADU-SPS too. This section presents Kahn network model and its derived models, especially SDF model, their mathematical representations and boundedness and termination problems.

2.4.1 Kahn Process Networks

Like the basic conceptual model Kahn Process Network (KPN) is a model of computation where processes are connected by channels to form a network. The interesting

properties that make it a suitable model for computation are [42]:

- each process is a sequential program that consumes tokens from its inputs and produces tokens to the output queues.
- channel is the only way of information exchange and it is unidirectional.
- each process is blocked when it tries to read from an empty channel. It can not examine a channel to test for the presence of data
- writing is non blocking but in practice we are interested in channels that have limited size.
- systems that can be modeled by this model are deterministic. Results or token produced on channels do not depend on the execution order [42].

The goal is to execute process network programs always with bounded channel whenever it is possible. The real differences between streaming applications and scientific computing are: infinite stream and infinite program execution. Because each process can be seen as a Turing machine, this model is a Turing machine network and because termination of a Turing machine is an undecidable problem, we can't know always in finite time if this process will terminate or not, then termination of process network programs is undecidable too. Bounded buffering is another problem, in practice buffer or memory have limited sizes. It can be transformed to a termination problem. Hence it is undecidable. But buffering depends on the execution order so if we find a valid execution order, which does not increase buffer size infinitely, then the problem is solved. While both of these properties are undecidable for Kahn Process Network programs, for some restricted models, like SDF, they are decidable as we will see later.

Mathematical Representation: Kahn's formal mathematical representation of process networks is very easy, efficient and it proves determinism of KPN programs. Produced tokens depend only on program definition and not on execution order. In this formalism, channels are represented by streams and processes are functions that map streams into streams and the process network is this set of equations. The least fixed point of these equations is unique and it corresponds to the histories of the streams in the network which proves the determinism of the KPN program [42].

Stream: a stream is a sequence of finite or infinite elements: $X = [x1, x2, x3, ..]$. X, Y are streams, $X \sqsubseteq Y$ means X is a prefix or it is equal to Y, for example $X = [0]$ is a prefix of $Y = [0, 1]$. $\perp$ is the empty stream and $\forall X, \perp \sqsubseteq X$. Each increasing chain $\vec{X} = (X_1, X_2, ...)$ where $X_1 \sqsubseteq X_2 \sqsubseteq ...$ has a least upper bound $\cup\vec{X} = \lim_{i\to\infty} X_i$.

Process: A process is a function that maps input streams into output streams. This functional mapping can be described by an equation. For example, the process in Fig. 2.2 can be described by this equation:

$$(Y1, Y2) = f(X1, X2, X3) \tag{2.1}$$

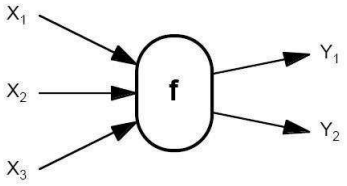


Fig. 2.2. Graphical Representation of a Process

A functional mapping is continuous iff

$$\forall \vec{X}, X1 \sqsubseteq X2 \sqsubseteq ... \qquad f(lim_{i\to\infty} X_i) = lim_{i\to\infty} f(X_i) \tag{2.2}$$

Fixed Point Equations: as we have said before, we can represent the process network by a set of equations representing different function mappings (different processes). If the functions are continuous mappings over a complete partial order then there is a unique least fixed point for this set of equations, and that solution corresponds to the histories produced on the communication channels which proves the determinism of KPN programs. KPN in Fig. 2.3 can be described by :

- $(T_1, T_2) = g(X)$
- $X = f(Y, Z)$
- $Y = h(T_1, 0)$
- $Z = h(T_2, 1)$

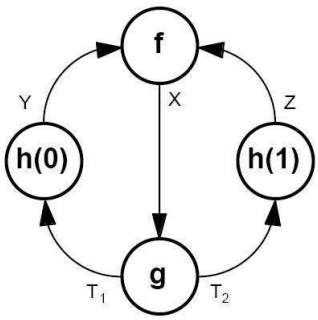


Fig. 2.3. Graphical Representation of the Process Network

This system can be reduced to one equation: $(T_1, T_2) = g(f(h(T_1, 0), h(T_2, 1)))$ Because the solution corresponds to the channels' histories, so it is the set of channel's least upper bound, and if it is a terminating program all streams will be finite, else at least one of them is infinite.

In our example and by induction:

- $(T_1, T_2)^0 = (\perp, \perp)$
- $(T_1, T_2)^1 = g(f(h(\perp, 0), h(\perp, 1))) = ([0], [1])$

- $(T_1, T_2)^2 = g(f(h([0], 0), h([1], 1))) = ([0, 0], [1, 1])$
- $(T_1, T_2)^3 = g(f(h([0, 0], 0), h([1, 1], 1))) = ([0, 0, 0], [1, 1, 1])$
- $(T_1, T_2)^{j+1} = g(f(h(T_1^j, 0), h(T_2^j, 1))) = ([0, 0, 0, ...], [1, 1, 1, ..])$

The least fixed point solution is: $T_1 = [0, 0, 0...], T_2 = [1, 1, 1, ...]$. $Y = h(T_1, 0) = [0, 0, 0, ...]$ and $Z = h(T_2, 1) = [1, 1, 1, ...]$. $X = f(Y, Z) = [0, 1, 0, 1, ...]$.

Determinism, Termination and Boundedness.

Determinism : A PN is determinate if the result of the computation doesn't depend on the execution order. KPN programs are determinate because they are based on the fact that the least fixed point, which represents channels' histories, is unique [42].

Termination: It is closely related to determinism. The least fixed point solution determines stream values of each channel in the program. If we know the stream values, we know each length. Then we can know if the stream is finite or not. A complete execution of the KPN program corresponds to the least fixed point. A terminating KPN program is a program where all complete executions have a finite number of operations (finite streams) and a non-terminating KPN program is one where all its complete executions have an infinite number of operations.

Boundedness: Although the length of produced token in each channel is defined by the program, the number of tokens unconsumed in channels depends on the execution order. A channel strictly bounded by b is a channel where the number of unconsumed tokens doesn't exceed b for any complete execution. A channel is bounded by b if there is a complete execution where the number of unconsumed tokens in this channel doesn't exceed b. This last definition is a weaker condition but it is not always easy or possible to find this complete execution or this execution order.

2.4.2 Dataflow Networks

Dataflow networks, like KPN, can be represented by a graph where arcs represent the FIFO channels and nodes represent actors. The difference between these two models is that dataflow networks don't use blocking reading semantics. They have firing rules instead of it. These firing rules specify the number of tokens that should be ready in each actor input channel, to enable the execution of the actors and actors are atomic. A process is formed by firing an actor many times, infinitely, for an infinite stream [42].

Stream here has the same meaning. It is represented the by tokens, it can be finite or infinite. In contrast to the process (functional mapping from streams to streams), actors are functions that map input tokens to output tokens. They specify how many tokens should be available in each input channel for the actor to fire. When it fires it consumes some input tokens and produces some output tokens.

Firing rules: an actor can have one or more firing rules:

$$R = \vec{R}_1, \vec{R}_2, ..., \vec{R}_N \tag{2.3}$$

R is the set of firing rules of the actor. It will fire if at least one firing rule R_i is satisfied. And for an actor with p input channels:

$$\vec{R_i} = R_{i,1}, R_{i,2}, ..., R_{i,p} \tag{2.4}$$

For $\vec{R_i}$ firing rule to be satisfied, each $R_{i,j}$ should be a prefix of the stream X_j because $R_{i,j}$ is a sequence of tokens specifying what tokens should be available on the channel "j" to let the actor fire.
$R_{i,j} = \perp$ means there is no condition on this channel and the actor can fire for any token on this channel but it doesn't mean the channel should be empty. A pattern $R_{i,j}$ can specify input tokens to have some particular values. For instance, the select actor firing in Fig. 2.4 depends on the control token values. It has two firing rules: $\vec{R_1} = ([*], \perp, [F])$, $\vec{R_2} = (\perp, [*], [T])$. If the control token equals true then R_1 is selected else R_2. So $R_{i,3}$ pattern specifies the value of this token. $[*]$ means the alone condition is: at least one token exists and $\perp$ means no condition even if the channel is empty the actor can fire.

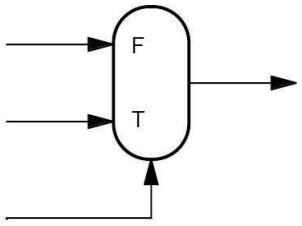


Fig. 2.4. The Select Actor

Sequential Firing Rules: any firing rule that can be implemented as a sequence of blocking reads is a sequential firing rule. For example, selector firing rules are sequential: a blocking read of the control input is followed by a blocking read of the appropriate data input. But with non-determinate merge, firing rules are not sequential, see Fig. 2.5: The blocking read of one input doesn't mean the same for the other. If tokens are available in both channels so there is an ambiguity. With sequential rules this problem doesn't appear at all.

Execution Order: there is another difference between Kahn Process Networks and Data Flow Networks. Because the actor firing is atomic, ordering actor firings will implicitly order put and get operations. In a channel , the producer actor put tokens in it, "put" operation and the consumer actor get tokens form it, "get" operation.

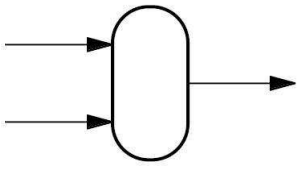


Fig. 2.5. The Non-determinate Merge Actor

2.4.3 Synchronous Dataflow Networks:

Synchronous Dataflow Networks, SDF, is a restricted dataflow network model . SDF, like dataflow networks, consists of actors and channels but the initial number of tokens on each channel, the number of consumed and produced tokens by each actors are all predefined. Because the number of consumed tokens is static so we have only one firing rule which means it is a determinate model. It can be represented by a matrix Γ where rows represent arcs and columns represents nodes. $\Gamma_{i,j}$ is the number of tokens produced by the j^{th} node on the i^{th} arc. It is negative if the node j is a consumer and not a producer and equals zero if the channel is not connected with an actor. If an actor is both producer and consumer, $\Gamma_{i,j}$is the difference between the number of tokens produced and the number of token consumed.
Steady State Making the system balanced means finding a sequence of actor firings. How much each actor firing should be repeated such that it returns the system to its initial state. By state we mean the number of tokens in each channel. So after the execution of this sequence, the number of consumed tokens equals the number of produced tokens on each channel. To find this sequence we should resolve this equation and find the vector $\vec{r}$:

$$\Gamma\vec{r} = \vec{0} \tag{2.5}$$

r_j means how much the actor j will fire. We call this sequence a "Steady State" or "Complete Cycle". For example in Fig. 2.6 actor1 produces 2 tokens each time it fires and actor2 consumes 3 tokens from the same channel on each firing. So to make the system balanced, we should execute actor1 3 times and actor2 twice and continue repeating this sequence. In this example $\Gamma = (2 \quad -3)$ and $\vec{r} = (3 \quad 2)^T$.
Bounded Channel Because the number of actor firings is finite in the complete cycle and the number of tokens produced by an actor is finite so channels are always bounded. We can repeat the sequence forever by consuming only a bounded number of tokens on the arcs. For this reason this model doesn't suffer from bounded channel problem. Also because the steady state tells us only how much each actor should fire, we can find many execution orders for this complete cycle. For our example of Fig. 2.6 1,1,2,1,2 and 1,1,2,1,2 are two valid execution orders for the complete cycle but the first is better, in buffer consumption, because the number of un-consumed tokens in the first case equals 4 and in the second case equals 6. Hence, in this model, the choice of execution order is important. It is even difficult if there are other constraints like schedule size and resources [42][29].

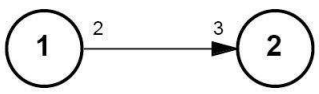


Fig. 2.6. Bounded Synchronous Dataflow Program

Initial State Finding the steady state, non-null $\vec{r}$, is not enough for the SDF program to execute forever with bounded channels. Having enough initial tokens on channels to execute a complete cycle without deadlock is necessary too. In the

example of Fig. 2.7, the SDF program has a steady state (complete cycle) but it is deadlocked because it doesn't have enough tokens at its initial state:

$$\mathbf{\Gamma} = \begin{pmatrix} 2 & -3 & 0 \\ -1 & 0 & 3 \\ 0 & 1 & -2 \end{pmatrix} \text{ and } \vec{r} = \begin{pmatrix} 3 & 2 & 1 \end{pmatrix}^T$$

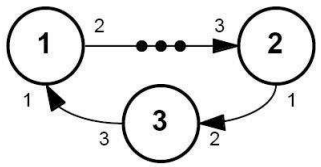


Fig. 2.7. A Deadlocked Synchronous Dataflow Program

2.5 Conclusion

To conclude, KPN is the natural representation of dataflow systems but termination and boundedness are undecidable. A restricted model of KPN is Dataflow Networks, that use firing rules instead of the blocking read and atomic actor firings. With these, we have more chance to find a valid execution order, no deadlock and bounded channel. However both properties termination and boundedness are still undecidable. In contrast to KPN, dataflow network models can be un-determinate as well when two or more firing rules are satisfied. The model which can verify these two properties is synchronous dataflow networks(SDF), a restricted model of data flow networks. The initial number of tokens in each channel, the consumed and produced tokens by each actor firing are all predefined. So it is always possible to find a valid schedule (execution order). Of course it is possible only if the application is an SDF program. Not all dataflow systems are SDF applications but many interesting applications are so, in many interesting domains: digital signal processing, embedded systems, mobile phones, etc. We have tried to restrict our work to SDF applications as a first step. Other interesting models exist like Dynamic Dataflow (DDF) where the input streams are not static, different actor firings may consume different amount of tokens.

Chapter 3

Dependence Analysis and Loop Transformations

3.1 Introduction

A steady state is exactly a loop nest where actor occurrences replace instructions. This similitude between steady state and nested loops let us think about steady state scheduling problem as a loop nest scheduling problem, especially because there are a lot of works on loop nest scheduling. Because of this importance of loop scheduling in our work and because our proposed scheduling technique, Prolog-epilog Merging, is for both steady state and loop nest we will try to do a survey on loop basic notions and its transformations that are necessary for readers to understand our work.

3.2 Dependence Analysis

This section briefly introduces dependence analysis, its terminology and the underlying theory. A dependence is a relationship between two computations that places constraints on their execution orders. Dependence analysis identifies these constraints, which are then used to determine whether a particular transformation can be applied without changing the semantics of the computation [27][2].

3.2.1 Type of Dependences

There are two kinds of dependences: data dependence and control dependence [27][2].

- **Data Dependence** A data dependence is a situation in which a program statement (instruction) refers to the data of a preceding statement. Assuming statement $S1$ and $S2$, $S2$ depends on $S1$ if:
  $[I(S1) \cap O(S2)] \cup [O(S1) \cap I(S2)] \cup [O(S1) \cap O(S2)] \neq \emptyset$ where:$I(Si)$ is the set of memory locations read by Si and $O(Sj)$ is the set of memory locations written by Sj and there is a feasible run-time execution path from $S1$ to $S2$

This Condition is called Bernstein Condition, named by A. J. Bernstein. Three cases exist:

- **True Dependence:** $O(S1) \cap I(S2), S1 \rightarrow S2$ and $S1$ writes something read by $S2$. It occurs when an instruction depends on the result of a previous instruction:

```
1.  a = 3
2.  b = a
3.  c = b
```

  Instruction 3 is truly dependent on instruction 2, as the final value of c depends on the instruction updating b. Instruction 2 is truly dependent on instruction 1, as the final value of b depends on the instruction updating a. Since instruction 3 is truly dependent upon instruction 2 and instruction 2 is truly dependent on instruction 1, instruction 3 is also truly dependent on instruction 1. Instruction level parallelism is therefore not an option in this example.

- **Anti-dependence:**$I(S1) \cap O(S2)$, mirror relationship of true dependence. It occurs when an instruction requires a value that is later updated. In the following example, instruction 3 anti-depends on instruction 2. The ordering of these instructions cannot be changed, nor can they be executed in parallel (possibly changing the instruction ordering), as this would affect the final value of a.

```
1.  b = 3
2.  a = b + 1
3.  b = 7
```

  An anti-dependence is an example of a name dependence. That is, renaming of variables could remove the dependence, as in the next example:

```
1.  b = 3
N. b2 = b
2.  a = b2 + 1
3.  b = 7
```

  A new variable, $b2$, has been declared as a copy of b in a new instruction, instruction N. The anti-dependence between 2 and 3 has been removed, meaning that these instructions may now be executed in parallel. However, the modification has introduced a new dependence: instruction 2 is now truly dependent on instruction N, which is truly dependent upon instruction 1. As true dependences, these new dependences are impossible to safely remove.

- **Output Dependence:** $O(S1) \cap O(S2), S1 \rightarrow S2$ and both write the same memory location. It occurs when the ordering of instructions will affect the final output value of a variable. In the example below, there is an output dependence between instructions 3 and 1. Changing the ordering of instructions in this example will change the final value of a, thus these instructions can not be executed in parallel.

```
1.  a = 2 ×x
2.  b = a / 3
3.  a = 9 ×y
```

As with anti-dependences, output dependences are name dependences. That is, they may be removed through renaming of variables, as in the below modification of the above example:

```
1 a2 = 2 ×x
2 b = a2 /3
3 a = 9 ×y
```

A commonly used naming convention for data dependences is the following: Read-after-Write (true dependence), Write-after-Write (output dependence), and Write-After-Read (anti-dependence).

- **Control Dependence** An instruction B is control dependent on a preceding instruction A if the latter determines whether B should execute or not. In the following example, instruction $S2$ is control dependent on instruction $S1$.

```
S1.  if a == b goto AFTER
S2.  a = 2 ×x
S3.  AFTER
```

Intuitively, there is a control dependence between two statements $S1$ and $S2$ if $S1$ could be possibly executed before $S2$. The outcome of S1 execution will determine whether $S2$ will be executed. A typical example is that there is a control dependence betweenthe if statement's condition part and the statements in the corresponding true/false bodies. A formal definition of control dependence can be presented as follows: A statement $S2$ is said to be control dependent on another statement $S1$ if and only if there exists a path P from $S1$ to $S2$ such that every statement $Si \neq S1$ within P will be followed by $S2$ in each possible path to the end of the program and $S1$ will not necessarily be followed by $S2$, i.e., there is an execution path from $S1$ to the end of the program that does not go through $S2$. Expressed with the help of (post-)dominance the two conditions are equivalent to:

  - $S2$ post-dominates all Si
  - $S2$ does not post-dominate $S1$

Implications: Conventional programs are written assuming the sequential execution model. Under this model, instructions execute one after the other, atomically (i.e., at any given point of time only one instruction is executed) and in the order specified by the program. However, dependences among statements or instructions may hinder parallelism parallel execution of multiple instructions, either by a parallelizing compiler or by a processor exploiting instruction level parallelism (ILP). Recklessly executing multiple instructions without considering related dependences may cause danger of getting wrong results, namely hazards.

3.2.2 Representing Dependences

To capture the dependence for a part of code, the compiler creates a dependence graph; each node in the graph typically represents one statement. An arc between two nodes indicates that there is a dependence between the computation they represents. Because it is cumbersome to account for both data and control dependence during analysis, compiler often convert control dependences into data dependences using a technique called *if-conversion* [3]. If-conversion introduces additional boolean variables that encode the conditional predicates; every statement whose execution depends on the conditional is then modified to test the boolean variable. In the transformed code data dependence subsumes control dependence.

3.2.3 Loop Dependence Analysis

Loop dependence analysis is the task of determining whether statements within a loop body form a dependence, with respect to array access and modification, induction, reduction and private variables, simplification of loop-independent code and management of conditional branches inside the loop body. Loop dependence analysis is mostly done to find ways to do automatically parallelization, by means of vectorization, shared memory or others. Loop dependence analysis occurs on a normalized loop of the form:

```
for i1 to U1
    for i2 to U2
    ...
        for in to Un
        body
```

where "body" may contain:

```
S1:  a[f1(i1, ..., in), ..., fm(i1, ..., in)] := ...
    ...
S2 :  ...  := a[h1(i1, ..., in), ..., hm(i1, ..., in)]
```

Where a is an n-dimensional array and fn, hn, etc. are functions mapping from all iteration indexes in to a memory access in a particular dimension of the array. For example, in C:

```
for(i = 0; i < u1; i++)
    for(j = 0; j < u2; j++)
        a[i+4-j] = b[2×i − j] + i;
```

$f1$ would be $i+4-j$, controlling the write on the first dimension of a and $f2$ would be $2-j$, controlling the read on the first dimension of b.
In general, the body of a loop contains many instructions and the scope of the problem is to find all possible dependences between them. To be conservative, any dependence which cannot be proved false must be assumed to be true.
Independence is shown by demonstrating that no two instances of two instructions $S1$ and $S2$ access or modify the same variable or the same box in arrays. When a possible dependence is found, loop dependence analysis usually makes every attempt

to characterize the relationship between dependent instances, as some optimizations may still be possible. It may also be possible to transform the loop to remove or modify the dependence.
In the course of (dis)proving such dependences, a statement S may be decomposed according to which iteration it comes from. For instance, $S[1, 3, 5]$ refers to the iteration where $i1 = 1, i2 = 3$ and $i3 = 5$. Of course, references to abstract iterations, such as $S[d1 + 1, d2, d3]$, are both permitted and common.

- **Iteration vectors** A specific iteration through a normalized loop is referenced through an iteration vector, which encodes the state of each iteration variable. For a loop, an iteration vector is a member of the Cartesian product of the bounds for the loop variables. In the normalized form given previously, this space is defined to be $U1U2...Un$. Specific instances of statements may be parametrized by these iteration vectors, and they are also the domain of the array subscript functions found in the body of the loop. Of particular relevance, these vectors form a lexicographic order which corresponds with the chronological execution order.

- **Dependence Vectors** To classify data dependence, compilers use two important vectors: the distance vector (σ), which tells what is the distance between fn and hn, and the direction vector (ρ), which tells in which direction this distance points to. The distance vector is defined as $\sigma = (\sigma_1, \ldots, \sigma_k)$ where σ_n is $\sigma_n = i_n - j_n$ The direction vector is defined as $\rho = (\rho_1, ..., \rho_k)$ where ρ_n is:

  - $(<)$ if $\sigma_n > 0 \rightarrow [fn < hn]$
  - $(=)$ if $\sigma_n = 0 \rightarrow [fn = hn]$
  - $(>)$ if $\sigma_n < 0 \rightarrow [fn > hn]$

  Example:

```
for(i = 0; i<N;i++)
    for(j=0;j<M;j++)
      for(k=0;k<L;k++)
S1    a[i+1, j, k-1] = a[i, j, k] + 10;
```

  $S1$ has a true dependence on itself. The distance vector is $\sigma = (1, 0, -1)$ and the direction vector is $\rho = (<, =>)$.

- **Classification** A dependence between two operations a and b can be classified according to the following criteria:

  - **Operation type:**
    - if a is a write and b is a read, this is a flow dependence
    - if a is a read and b is a write, this is an anti-dependence

    - if a is a write and b is a write, this is an output dependence
    - if a is a read and b is a read, this is an input dependence
  - **Chronological order:**
    - if $Sa < Sb$, this is a lexically forward dependence
    - if $Sa = Sb$, this is a self-dependence
    - if $Sa > Sb$, this is a lexically backward dependence
  - **Loop dependence:**
    - if all distances (σ) are zero (same place in memory), this is loop independent
    - if at least one distance is non-zero, this is a loop carried dependence

3.3 Loop Transformations

This section catalogs the loop transformations themselves. For each transformation we provide an example, define its benefits and shortcomings.

3.3.1 Loop Fusion

Loop Fusion, also called "Loop Jamming", is a compiler optimization, a loop transformation, which replaces multiple loops with a single one [27][3].
Example in C

```
int i, a[100], b[100];
for (i = 0; i < 100; i++)
    a[i] = 1;
for (i = 0; i < 100; i++)
    b[i] = 2;
```

is equivalent to:

```
int i, a[100], b[100];
for (i = 0; i < 100; i++)
{
    a[i] = 1;
    b[i] = 2;
}
```

Note:

- To fuse two loops they must have the same loop bounds.
- Two loops with same bounds may be fused if there doesn't exist statement $S1$ in the first loop and statement $S2$ in the second loop such that they have a dependence $S2 \longrightarrow^{(<)} S1$, this means that $S1$ depends on $S2$ and direction vector is always "<", in the fused loop. This would be incorrect because before fusion all instances of $S1$ execute before any $S2$ which is not the case after fusion.

- Some optimizations like this don't always improve the run-time performance. This is due to architectures that provide better performance if there are two loops rather than one, for example due to increased data locality within each loop. In those cases, a single loop may be transformed into two, which is called **loop fission**

- We will see later, in our work, that our technique "Prolog Epilog Merging in Nested Loops" can be applied when Fusion can not.

3.3.2 Loop Interchange

Loop interchange is the process of exchanging the order of two iteration variables [3]. For example, in the code fragment:

```
for j= 0 to 10
    for i=0 to 20
        a[i,j] = i + j;
```

loop interchange would result in:

```
for i=0 to 20
    for j=0 to 10
        a[i,j] = i + j;
```

On occasion, such a transformation may lead to opportunities to further optimize, such as vectorization of the array assignments.

The Utility of Loop Interchange: one major purpose of loop interchange is to improve the cache performance for accessing array elements. Cache misses occur if the contiguously accessed array elements within the loop come from a different cache line. Loop interchange can help prevent this. The effectiveness of loop interchange depends on and must be considered in light of the cache model used by the underlying hardware and the array model used by the compiler. In the C programming language, the array elements from the same row are stored consecutively (Ex: a[1,1],a[1,2], a[1,3]...), namely row-major. On the other hand, FORTRAN programs store array elements from the same column together(Ex: a[1,1],a[2,1],a[3,1]...) , called column-major. Thus the order of two iteration variables in the first example is suitable for C program while the second example is better for FORTRAN. Optimizing compilers can detect the improper ordering by programmers and interchange the order to achieve better cache performance.

Caveat: like any other compiler optimization, loop interchange may lead to worse performance because cache performance is only part of the story. Take the following example:

```
for i = 1, 10000
  for j = 1, 1000
    a(i) = a(i) + b(j,i) ×c(i)
```

Loop interchange on this example can improve the cache performance of accessing b(j,i), but it will ruin the reuse of a(i) and c(i) in the inner loop, as it introduces two extra loads (for a(i) and for c(i)) and one extra store (for a(i)) during each iteration. As a result, the overall performance may be degraded after loop interchange.

Safety: it is not always safe to exchange the iteration variables due to dependences between statements for the order in which they must execute. To determine whether a compiler can safely interchange loops, dependence analysis is required.

3.3.3 Loop Tiling

Loop tiling, also known as loop blocking, strip mine and interchange, unroll and jam, or supernode partitioning, is a loop optimization used by compilers to make the execution of certain types of loops more efficient [67][69].
Overview Loop tiling partitions a loop's iteration space into smaller chunks or blocks, so as to help ensure data used in a loop stays in the cache until it is reused. The partitioning of loop iteration space leads to partitioning of large array into smaller blocks, thus fitting accessed array elements into cache size, enhancing cache reuse and eliminating cache size requirements.
Example The following is an example of matrix vector multiplication. There are three arrays, each with 100 elements. The code does not partition the arrays into smaller sizes.

```
int i, j, a[100][100], b[100], c[100];
int n = 100;
for (i = 0; i < n; i++)
{
  for (j = 0; j < n; j++)
  {
    c[i] = c[i] + a[i][j] ×b[j];
  }
}
```

After we apply loop tiling using 2×2 blocks, our code looks like:

```
int i, j, x, y, a[100][100], b[100], c[100];
int n = 100;
for (i = 0; i < n; i += 2)
{
  for (j = 0; j < n; j += 2)
  {
    for (x = i; x < min(i + 2, n); x++)
    {
      for (y = j; y < min(j + 2, n); y++)
      {
        c[x] = c[x] + a[x][y] ×b[y];
      }
    }
  }
}
```

The original loop iteration space is n by n. The accessed chunk of array a[i, j] is also n by n. When n is too large and the cache size of the machine is too small, the accessed array elements in one loop iteration (for example, i = 1, j = 1 to n) may cross cache lines, causing cache misses. Another example using an algorithm

for matrix multiplication:
Original matrix multiplication:

```
for (I = 1; I<= M; I++)
  for (K = 1; K<= M; K++)
    for (J = 1; J<= M; J++)
      Z(J, I) = Z(J, I) + X(K, I) * Y(J, K)
```

After loop tiling $B \times B$:

```
for (K2 = 1; K2<= M; K2 = K2 + B)
  for (J2 = 1; J2<= M; J2 = J2 + B)
    for (I = 1; I<= M; I++)
      for (K1 = K2; K1<= MIN(K2 + B - 1, M); K1++)
        for (J1 = J2; J1<= MIN(J2 + B - 1, M); J1++)
        Z(J1, I) = Z(J1, I) + X(K1, I) * Y(J1, K1)
```

It is not always easy to decide what value of tiling size is optimal for one loop because it demands an accurate estimate of accessed array regions in the loop and the cache size of the target machine. The order of loop nests (loop interchange) also plays an important role in achieving better cache performance.

3.3.4 Retiming

One of the most important techniques our idea is based on is **Retiming**. Although our shifting is more complex, the basic notions of instructions moving forward stay always the same. In the following section we are going to see the idea of retiming in a detailed way.

- **Retiming and its Effect** Retiming [33] is a transformation technique which rebuilds iterations of a loop nest by redistributing instructions. Each instruction I_i is retimed by some integral amount $r(I) \leq 0$ representing the offset between I_i's original iteration and its iteration following retiming. Since retiming moves instructions forward in th execution flow of a program, it alters the distances of any dependences involving said instructions. Thus we may be able to further break the loop-carried dependences of a program by retiming the code by a properly selected retiming factor to extract parallelism. As an example, consider the sample loop in Fig. 3.1.

```
for i= 1 to N
 {
   x = a[i] + b[i];          /* 1 */
   a[i+1] = x;               /* 2 */
   b[i] = b[i] + 1;          /* 3 */
   a[i+3] = a[i+1] -1;       /* 4 */
   a[i+2] = a[i+1] + 1;      /* 5 */
 }
```

Fig. 3.1. Loop Code Before Retiming

Its flow of execution is pictured at the top of Fig. 3.3. If we begin by retiming instruction 1 once, we shift each occurrence of instruction 1 forward by one iteration, as seen by the second line of Fig. 3.3. In doing so, we reorder instructions so that instruction 1 moves from the head of the repeating nest to the tail. This shifting also takes the very first occurrence of instruction 1 out of the repeating loop nest entirely and makes it a code section all its own, to be executed before beginning the repeated iterations. This new section is called the prolog of the schedule. Similarly, shifting removes the last occurrence of instruction 1 from the final iteration, leaving us with an incomplete iteration to execute after our repeating loop terminates. This final section is called the epilog. Fig. 3.2(a) illustrates our revised code at this point, with one iteration of the loop nest removed and divided between prolog and epilog.

```
x = a[1] + b[1];        /* 1 */
for i= 1 to N-1
 {
   a[i+1] = x;          /* 2 */
   b[i] = b[i] + 1;     /* 3 */
   a[i+3] = a[i+1] -1;  /* 4 */
   a[i+2] = a[i+1] + 1; /* 5 */
   x = a[i+1] + b[i+1]; /* 1 */
 }
a[N+1] = x;             /* 2 */
b[N] = b[N] + 1;        /* 3 */
a[N+3] = a[N+1] -1;     /* 4 */
a[N+2] = a[N+1] + 1;    /* 5 */
```

(a)

```
x = a[1] + b[1];        /* 1  iter 1 */
a[2]= x;                /* 2  iter 1*/
x = a[2] + b[2];        /* 1  iter 2 */
for i= 1 to N-2
 {
   b[i] = b[i] + 1;     /* 3  iter i    = 1' */
   a[i+3] = a[i+1] -1;  /* 4  iter i    = 2' */
   a[i+2] = a[i+1] + 1; /* 5  iter i    = 3' */
   a[i+1] = x;          /* 2  iter i+1  = 4' */
   x = a[i+2] + b[i+2]; /* 1  iter i+2  = 5' */
 }
b[N-1] = b[N-1] + 1;    /* 3  iter N-1 */
a[N+2] = a[N] -1;       /* 4  iter N-1 */
a[N+1] = a[N] + 1;      /* 5  iter N-1 */
a[N+1] = x;             /* 2  iter N   */
b[N] = b[N] + 1;        /* 3  iter N */
a[N+3] = a[N+1] -1;     /* 4  iter N */
a[N+2] = a[N+1] + 1;    /* 5  iter N */
```

(b)

Fig. 3.2. (a) After Retiming Instruction Once; (b) After Retiming is completed.

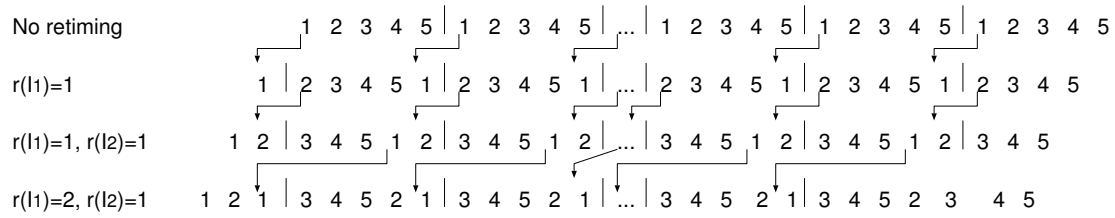


Fig. 3.3. Alteration of Execution Flow Pattern by Retiming.

We may now proceed in a similar fashion to retime instruction 2, resulting in the pattern seen in the third line of Fig. 3.3. Each copy of instruction 2 moves forward an iteration, with the first copy moving to the prolog and the last leaving the epilog. We may also retime instructions within our nest. For example, retime instruction 1 a second time, as in the last line of Fig. 3.3.

The effect is to pick each copy of instruction 1 up out of the middle of its old iteration and append it to the end of the previous iteration as shown. Retiming instruction 1 this second time also leaves us with two incomplete iterations at the end of our execution, which are merged to form the new epilog. (In general, as noted in [11], there are $r(Ii)$ copies of instruction Ii in the prolog and $(maxIjr(Ij)))r(Ii)$ copies in the epilog once retiming is complete.) This final flow pattern from Fig. 3.3 gives us the needed information for constructing a retimed version of our initial example, which is seen in Fig. 3.2(b).

- **Retiming Representation with Directed DFG** To this point, we have discussed the effect of retiming on code. Typically, however, retiming is viewed solely as an operation on directed graphs which represent control flow within code. These graphs, called data-flow graphs or DFGs, model a loop nest by assigning a vertex to each instruction of the loop nest and representing dependences between relations by directed edges between nodes. These edges are weighted by delays which indicate what we have dubbed the dependence distance. For example, the DFG in Fig 3.4(a) models the behavior of the loop nest in Fig. 3.1. Each Read After Write (RAW) with non-zero characteristic value found earlier corresponds to a weighted edge in this graph; we have excluded dependences with value zero from this representation for reasons we will indicate shortly.
  Retiming can now be viewed as pulling delays from a node incoming edges and pushing them onto the node outgoing edges. (Thus retiming will not affect the delay count of an edge from a node to itself, making the representation of zero-characteristic-value dependences in a DFG a useless exercise which only clouds the issue.) For example, let (i, j) represent the directed edge from vertex i to vertex j. Retiming node 1 by 1 pulls a delay in from each of the edges $(4, 1)$, $(5, 1)$ and $(2, 1)$ and deposits a delay onto edge $(1, 2)$. Retiming node 2 by 1 then draws this delay from $(1, 2)$ in and pushes it onto edges $(2, 1)$, $(2, 4)$ and $(2, 5)$. Thus there are enough delays to retime vertex 1 once more, resulting in the retimed DFG of Fig 3.4(b). It is well-known that the effect of retiming by a function r is to alter the delay count of edge $e : u \rightarrow v$ from $d(e)$ to $d(e) + r(u) - r(v)$ (this retimed delay count is denoted $dr(e)$). Since an edge cannot be assigned a negative number of delays, we must have $dr(e) \geq 0$ for all edges e in order for a retiming to be legal.

3.3.5 Software Pipelining

It is well known that loops monopolize most execution time of programs. The well known technique for optimizing loop execution under time and resources constraints is *software pipelining*, a sort of Retiming. It tries to overlap the execution of different loop iterations to increase parallelism. Hence, the problem of loop pipelining is reduced to finding a steady state pattern (a folded loop body) that can execute at the maximum rate allowed by the dependences [13] [1][6]. In the new steady state, instructions from different iterations(folds) are executed, see Fig. 3.5.

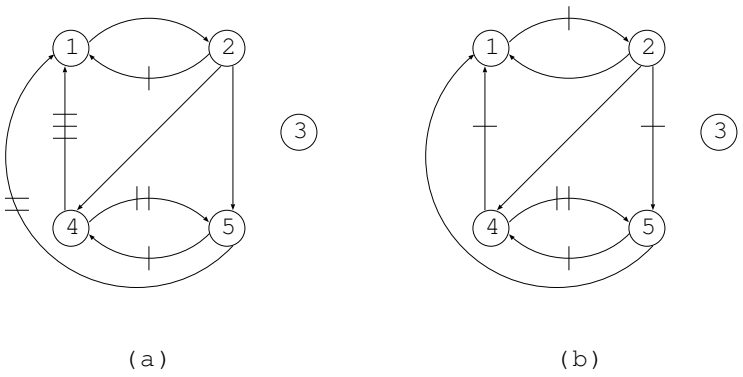


Fig. 3.4. The DFG for Fig: 3.1 (a) Before Retiming; (b) After Retiming

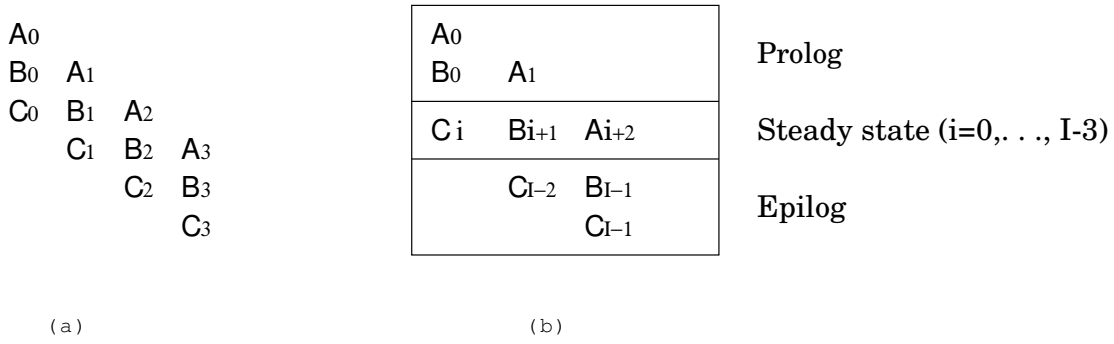


Fig. 3.5. overlapped loop execution

The problem is generally: given a set of resource constraints (functional units and registers), the objective is to find a pipelined loop schedule that minimizes the execution time.

- **Dependence Graph** Loops are usually represented by *data dependence graph* (DDG). Fig. 3.6 shows an example. Vertices represent computational nodes and edges represent intra-loop and loop-carried dependences, e.g. B_i (reads $X[i]$) depends on A_i (produces $X[i]$) and B_{i+1} (reads $Y[i+1]$) depends on B_i (produces $Y[i+1]$). Labels indicate the number of iterations traversed by the dependence.

- **Span** Another important aspect to be considered is *span*, defined as the number of folds required to obtain the steady state. Smaller span results in shorter variable lifetimes, reducing in general the schedule register pressure.

- **Loop Representation** The loop representation here is the same used later in our problem formulation. Loops can be represented by a doubly weighted data dependence graph, $G = (O, E, \lambda, \delta)$, called a loop DDG (LDDG), where O is the set of the operations in the loop, E is the set of dependence edges, λ and δ are two positive integer associated to each edge where δ represents the delay, in clock cycles, required for the operation o_i to produce its results (flow or output-dependence cases); or the delay required to read its operand

```
for (i = 0 ; N-1; i++)
{
  Ai:  X[i] = R[i] + S[i];
  Bi:  Y[i] = Y[i] + 2*X[i];
  Ci:  Z[i] = 4*Y[i+1] + T[i];
}
```

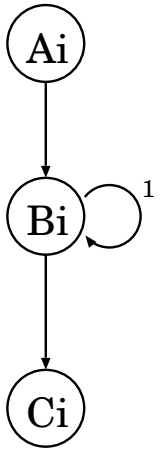


Fig. 3.6. Loop dependence graph

(anti-dependence case) and λ, Iteration Distance, is the number of iterations that separates an instance of the operation o_i from an instance of the operation o_j[13] [1] .

- **Software Pipelining Problem** The software pipelining problem can be described as follows: The aim is to find a loop schedule σ, a mapping function from $O \times N$ to N(positive integer set), $\sigma(op, i)$ denotes the execution cycle where the instance of operation op of the i_{th} iteration is issued if the following constraints are satisfied:

  - Resource constraints: in each cycle, the same resource (or resource stage for pipelined resource) can not used more than one time.
  - Dependence constraints: $\forall e = (op_i, op_j) \in E, \forall k \in N, \sigma(op_i, k) + \delta(e) \leq \sigma(op_j, k + \lambda(e))$.
  - Cyclicity constraint: σ must be expressible in the form of a loop, that is, $\exists\ \alpha,\ \beta \in\ N, \forall\ op\ \in\ O,\ \forall\ i \in\ N$ and $i > 0,\ \sigma(op, i) = \sigma(op, (i \mod \beta)) + \alpha \times (\lceil i/\beta \rceil - 1)$.

  Then we say that σ is a valid loop schedule for the given loop. α/β is called the average initiation interval of σ. The goal of software pipelining is to find a valid schedule with the minimum average initiation interval [66].

- **Initiation Interval Computing** There are many methods for computing the initiation interval [1][66][26][56][53]. For example, as proposed by authors of paper [31], to calculate the initiation interval II we calculate a lower bound on the II of the loop: the minimum initiation interval (MII) $MII = max(ResMII; RecMII)$ where $ResMII$ is determined by the number of available resources and RecMII by the cycles (recurrences) formed by the dependences of the loop. Clearly, $ResMII = \frac{n}{r}$ since each functional unit can execute only 1 instruction/cycle.

A recurrence R is a set of edges that form a cycle. Let us define R as

$$\delta_R = \sum_{(u,v)\in R} \delta(u,v)$$

For any operation u such that $\exists(u,v) \in R$, u_i must be scheduled at least $|R|$ cycles before $u_{i+\delta_R}$. Therefore, R imposes a minimum initiation interval, $RecMII_R$, on the execution of the loop:

$RecMIIR = \frac{|R|}{\delta_R}$

A loop can have several recurrences. Therefore: $RecMII = maxRecMII_R$

3.4 Conclusion

This was a survey on loop basic notions and its well-known transformations. Please keep in mind, data dependence, software pipelining and retiming notions, they will be repeated a lot in the second part of this thesis, 'Imperfectly Nested Multidimensional Shifting (INMS)'.

Chapter 4

Dependence Removal Techniques

4.1 Introduction

Flow (or value-based) dependences are the only "true" dependences of a program. Anti dependences, occuring when a variable is used in one statement and reassigned in a subsequently executed statement, and output dependences, occuring when a variable is assigned in one statement and reassigned in a subsequently executed statement, are due to storage re-use and can be eliminated at the price of more memory usage. Removing anti and output dependences may prove very useful to break data dependence cycles and thereby enable vectorization and/or improve parallelization. However, removing all memory-based or "false" (i.e. anti and output) dependences may have a prohibitive cost. A complete removal of false dependences is usually achieved, if feasible, via conversion of the original loop nest program into single assignment form. This turns out to be unnecessarily costly. Indeed, there are some memory-based dependences whose removal will not improve the parallelization. Rather, we should introduce as much memory overhead as needed to expose all the parallelism of the original program. As much as, but no more than, needed. In our work, breaking interphase dependences, dependences between two instructions of two different loops, is useful in parallelization improving too. So how anti and output dependences can be eliminated? The answer of this question and others are the aim of this chapter.

4.2 False Dependence Removal Techniques

Many papers have been devoted to the problem of eliminating anti and output dependences. Proposed methods include 'array data flow analysis' [21],[36], 'variable expansion' [8], 'variable renaming' [41] and 'node splitting'[41][44]. See the survey papers of Banerjee, Eigenmann, Nicolau and Padua [61] and Bacon, Graham and Sharp [17], as well as the books of Wolfe [68] and Zima [70], for further references. Note that 'array privatization' [28] is yet another technique that can be applied.

4.2.1 Renaming

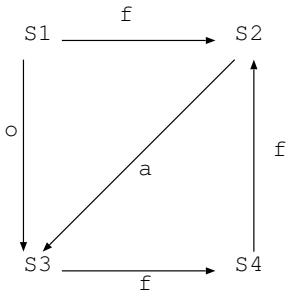
Scalar renaming consists in giving a different name to occurrences of a scalar locally used in a program. This allows the removal of anti and output dependences due to the multiple use of the scalar. This technique can be directly extended to array renaming. Consider the loop nest in Fig 4.1(a). The dependence graph (Fig 4.1(c)) contains a cycle with an anti dependence from statement S2 to statement S3. In the graph, 'o' stands for output dependence, 'a' for anti-dependence and 'f' for flow dependence. Renaming the array "a" in $S3$ (see the code in Fig 4.1(b)) breaks this dependence: the new graph (Fig 4.1(d)) is acyclic. Loop statements can now be parallelized.

```
For i =1 to N
 {
    S1 : a(i) = sin(i)
    S2 : b(i + 1) = a(i ) + c(i)
    S3 : a(i) = cos(i)
    S4 : c(i+ 1) = a(i)
 }
```

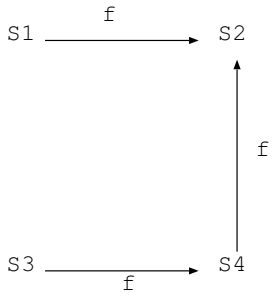
(a) original code

```
For i =1 to N
 {
    S1 : renamed(i) = sin(i)
    S2 : b(i + 1) = renamed(i ) + c(i)
    S3 : a(i) = cos(i)
    S4 : c(i+ 1) = a(i)
 }
```

(b) code after renaming



(c) original dependence graph



(d) graph after renaming

Fig. 4.1. Example of Variable Renaming

4.2.2 Privatization

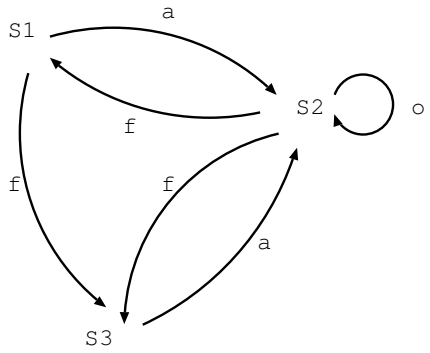
Consider a loop nest where a scalar variable is written at several iterations. This implies an output dependence from and to the statement involved in the multiple writings. Consider the example of Fig. 4.2(a). Since scalar a is written at each iteration, there is a self output loop around statement2 (see the dependence graph in Fig. 4.2(c)). To suppress this dependence, we expand a into a linear array, as shown in Fig. 4.2(b). Again, the new graph (Fig. 4.2(d))is acyclic. This technique can be extended to multi-dimensional loop nests for expanding scalars, or for expanding multi-dimensional arrays in the simple cases where the arrays can be considered as scalars when some loop indexes are fixed.

```
For i = 1 to N
   {
     S1: c(i) = 3 + a
     S2: a = i + 1
     S3: b(i) = c(i) + a
   }
```

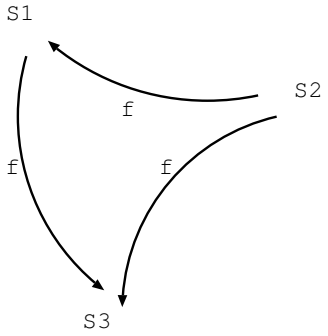
(a) original code

```
temp(0)= a
For i =1 to N
  {
    S1: c(i) = 3 + temp(i-1)
    S2: temp(i) = i + 1
    S3: b(i) = c(i) + temp(i)
  }
If (N>=1) a = temp(N)
```

(b) code after privatization



(c) original dependence graph



(d) graph after privatization

Fig. 4.2. Example of Variable Privatization

4.2.3 Node Splitting

This technique consists in splitting a statement into two statements, in order to break cycles in the dependence graph. Consider the example of Fig. 4.3(a). The dependence graph (Fig. 4.3(c)) contains a cycle involving a flow dependence from $S1$ to $S2$, and an output dependence from $S2$ to $S1$. To break this cycle, rather than writing array a in statement $S1$, we store the evaluation of the right-hand side into a temporary array *temp*. This temporary array is read in $S2$ instead of array a. The transformed code is given in Fig. 4.3(b), and the new dependence graph is represented in Fig. 4.3(d).

The previous example is due to Padua and Wolfe [41]. We generalize [44] the statement transformation as indicated in Fig. 4.4. The value computed at each iteration of statement S is stored into a temporary array whose access function is the same as that of "lhs", the left hand side of S. Obviously, if another statement instance depends upon a value "$lhs(g(i))$" computed by S, then the access to "$lhs(g(i))$" must be replaced by temp $(g(i))$. This implies knowledge of the statement instances which depend upon the value calculated in S (or in $S-'$ after the transformation).

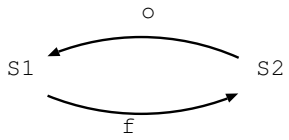
The impact of this transformation on the dependences going to and coming from a statement S is summarized in Fig. 4.5. As shown in [44], if this transformation is applied to all the statements of the original loop nest then the new dependence

```
For i=1 to N
  {
    S1: a(i) = b(i) + c(i)
    S2: a(i+1) = a(i)+ 2*d(i)
  }
```

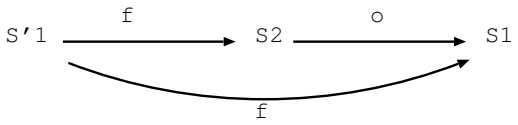
(a) original code

```
For i =1 to N
   {
     S'1: temp(i) = b(i) + c(i)
     S1: a(i) = temp(i)
     S2: a(i+ 1) = temp(i) + 2*d(i)
   }
```

(b) code after node splitting



(c) original graph



(d) graph after node splitting

Fig. 4.3. Example of Variable Splitting

```
For i=1 to N
  {
      ...
      S: lhs(f(i)) = rhs(...)
      ...
      ...          = lhs(g(i))
```

(a) original code

```
For i =1   to N
      ...
      S': temp(f(i)) = rhs(...)
      S: lhs(f(i)) = temp(f(i))
      ...
      ...            = temp(g(i))
```

(b) code after node splitting

Fig. 4.4. Transformation of Statement S

graph contains only flow dependence cycles and output dependence cycles. Furthermore,these cycles correspond to cycles of the initial dependence graph. However, applying the transformation to all statements is not a good approach. First, it can be too costly. Moreover, it is useless to transform some statements.

4.2.4 Conversion to Single-assignment Form

The only full transformation to single assignment form (SAF) has been proposed by Feautrier in [21]. The technique relies on an exact analysis of direct flow dependences (through parametric integer linear programming) that permits the source of each array reference to be found, i.e. the statement and the value of the surrounding loop counters where the desired element of the array has been computed.

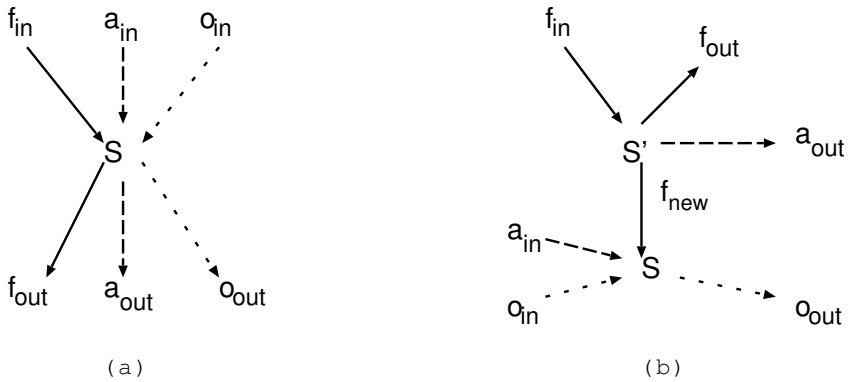


Fig. 4.5. A statement S with in-coming and out-going dependences (a) before and (b) after generalized Padua and Wolfe's transformation

4.3 Conclusion

As we have seen each technique has its advantages and its inconvenients. In our work we apply renaming technique to break false interphase dependences. We can apply privatization or any other technique in future to compare with renaming.

Chapter 5

Knapsack Problem

5.1 Introduction

In the previous chapters we have talked about many different subjects: 'Streaming Applications', 'Loop Dependencies and Transformations' and 'Dependence Removal Techniques'. In this chapter we are going to see the 'Knapsack Problem' because in order to schedule the steady state, with Prolog-epilog Merging scheduling technique, we need to solve this problem first.
Suppose we are planning a hiking trip; and we are, therefore, interested in filling a knapsack with items that are considered necessary for the trip. There are N different item types that are deemed desirable; these could include bottle of water, apple, orange, sandwich, and so forth. Each item type has a given set of two attributes, namely a weight (or volume) and a value that quantifies the level of importance associated with each unit of that type of item. Since the knapsack has a limited weight (or volume) capacity, the problem of interest is to figure out how to load the knapsack with a combination of units of the specified types of items that yields the greatest total value. What we have just described is called the knapsack problem [35].
A large variety of resource allocation problems can be cast in the framework of a knapsack problem. The general idea is to think of the capacity of the knapsack as the available amount of a resource and the item types as activities to which this resource can be allocated.

5.2 Formal Definition

Formally, suppose we are given the following parameters:

- w_k =the weight of each type$-k$ item, for $k = 1, 2, ..., N$,
- r_k = the value associated with each type$-k$ item, for $k = 1, 2, ..., N$,
- c = the weight capacity of the knapsack.

Then, our problem can be formulated as:

$$\begin{cases} \text{variables:} & \forall j \in \{1,\ldots,n\}, X_j \in \{0,1,2,\ldots\} \\ \text{objective:} & \max \sum_{j=1}^{n} r_j X_j \\ \text{constraints:} & \forall j \in \{1,\ldots,n\}, \\ & \qquad \sum_{j=1}^{n} w_j X_j \leq W \end{cases} \tag{5.1}$$

where $X_1, X_2, ..., X_N$ are nonnegative integer-valued decision variables, defined by X_k = the number of type-k items that are loaded into the knapsack.
Notice that since the Xs are integer-valued, what we have is not an ordinary linear program, but rather an integer program. Consequently, the Simplex algorithm cannot be applied to solve this problem.
As a simple numerical example, suppose we have: $N = 3$; $w_1 = 3$, $w_2 = 8$ and $w_3 = 5$; $r_1 = 4$, $r_2 = 6$, and $r_3 = 5$ and finally, $c = 8$. Observe that of the three item types, the first type has the greatest value per unit of weight. That is, of the three ratios, $r_1/w_1 = 4/3$; $r_2/w_2 = 6/8$; $r_3/x_3 = 5/5$ the first ratio is the greatest. Therefore, it seems natural to attempt to load as many type-1 items as possible into the knapsack. Since the capacity of the knapsack is 8, such an attempt will then result in the loading combination $X_1 = 2, X_2 = 0$ and $X_3 = 0$, with a total value of $r_1X_1 + r_2X_2 + r_3X_3 = 4 \times 2 + 6 \times 0 + 5 \times 0 = 8$. Is this loading combination optimal? The fact that this combination leaves a wasted slack of 2 in the knapsack is discomforting. Indeed, it turns out that this combination is not optimal; and that the optimal combination is to let $X_1 = 1$, $X_2 = 0$, and $X_3 = 1$, which achieves a total value of 9.
The Knapsack problem often arises in resource allocation with financial constraints. A similar problem also appears in combinatorics, complexity theory, cryptography and applied mathematics. The decision problem form of the knapsack problem is the question "can a value of at least V be achieved without exceeding the weight W?".
The problem appears in our case too where we have many loops and we can't retime all of them because of cycle distance limit. Choosing which loop to be retimed under cycle distance and buffer size is exactly a knapsack problem.

5.3 Knapsack Problems

The domain of X_j's values defines the kind of the knapsack problem.

5.3.1 0-1-Knapsack Problem

In the following, we have n kinds of items, 1 through n. Each kind of item j has a value r_j and a weight w_j. We usually assume that all values and weights are nonnegative. The maximum weight that we can carry in the bag is W.
The most common formulation of the problem is the **0-1 knapsack problem**, which restricts the number X_j of copies of each kind of item to zero or one, in our work our

problem is also 0-1 knapsack problem. Mathematically the 0-1-knapsack problem can be formulated as:

$$\begin{cases} \text{variables:} & \forall j \in \{1, \ldots, n\}, X_j \in \{0, 1\} \\ \text{objective:} & \max \sum_{j=1}^{n} r_j X_j \\ \text{constraints:} & \forall j \in \{1, \ldots, n\}, \\ & \qquad \sum_{j=1}^{n} w_j X_j \leq W \end{cases} \tag{5.2}$$

5.3.2 Bounded Knapsack Problem

The **bounded knapsack problem** restricts the number X_j of copies of each kind of item to a maximum integer value b_j. Mathematically the bounded knapsack problem can be formulated as:

$$\begin{cases} \text{variables:} & \forall j \in \{1, \ldots, n\}, X_j \in \{0, 1, \ldots, b_j\} \\ \text{objective:} & \max \sum_{j=1}^{n} r_j X_j \\ \text{constraints:} & \forall j \in \{1, \ldots, n\}, \\ & \qquad \sum_{j=1}^{n} w_j X_j \leq W \end{cases} \tag{5.3}$$

5.3.3 Unbounded Knapsack Problem

The **unbounded knapsack problem** places no upper bound on the number of copies of each kind of item [35]. Of particular interest is the special case of the problem with these properties:

- it is a decision problem
- it is a 0-1 problem
- for each kind of item, the weight equals the value: $w_j = r_j$

Notice that in this special case, the problem is equivalent to this: given a set of nonnegative integers, does any subset of it add up to exactly W? Or, if negative weights are allowed and W is chosen to be zero, the problem is: given a set of integers, does any subset add up to exactly 0? This special case is called the subset sum problem. In the field of cryptography the term knapsack problem is often used to refer specifically to the subset sum problem.
If multiple knapsacks are allowed, the problem is better thought of as the **bin packing problem**.

5.4 Computational Complexity

The knapsack problem is interesting from the perspective of computer science because:

- there is a pseudo-polynomial time algorithm using dynamic programming,

- there is a fully polynomial-time approximation scheme, which uses the pseudo-polynomial time algorithm as a subroutine,
- the problem is NP-complete to solve exactly, thus it is expected that no algorithm can be both correct and fast (polynomial-time) on all cases,
- many cases that arise in practice, and "random instances" from some distributions, can nonetheless be solved exactly.

Several algorithms are available to solve knapsack problems, based on dynamic programming approach [5], branch and bound approach [35] or hybridations of both approaches [54][64][45][34]

5.5 Knapsack Problem Solutions

5.5.1 Dynamic Programming Solution

Unbounded Knapsack Problem: If all weights $w_1, ..., w_n$ and W are nonnegative integers, the knapsack problem can be solved in pseudo-polynomial time using dynamic programming. The following describes a dynamic programming solution for the unbounded knapsack problem [35].
To simplify things, assume all weights are strictly positive ($w_j \geq 0$). We wish to maximize total value subject to the constraint that total weight is less than or equal to W. Then for each $Y \leq W$, define $A(Y)$ to be the maximum value that can be attained with total weight less than or equal to Y. $A(W)$ then is the solution to the problem.
Observe that A(Y) has the following properties:
$A(0) = 0$ (the sum of zero items, i.e., the summation of the empty set)
$A(Y) = max\{r_j \ + \ A(Y - w_j) \mid w_j \leq Y\}$
where r_j is the value of the jth kind of item.
Here the maximum of the empty set is taken to be zero. Tabulating the results from $A(0)$ up through $A(W)$ gives the solution. Since the calculation of each $A(Y)$ involves examining n items, and there are W values of $A(Y)$ to calculate, the running time of the dynamic programming solution is $O(nW)$. Dividing $w_1..., w_n, W$ by their greatest common divisor is an obvious way to improve the running time.
The $O(nW)$ complexity does not contradict the fact that the knapsack problem is NP-complete, since W, unlike n, is not polynomial in the length of the input to the problem. The length of the input to the problem is proportional to the number, $logW$, of bits in W, not to W itself.
0-1 Knapsack Problem A similar dynamic programming solution for the 0-1 knapsack problem also runs in pseudo-polynomial time. As above, assume $w_1, ..., w_n, W$ are strictly positive integers. Define $A(j, Y)$ to be the maximum value that can be attained with weight less than or equal to Y using items up to j.
We can define A(j,Y) recursively as follows:
$A(0, Y) = 0$
$A(j, 0) = 0$

$A(j, Y) = A(j-1,\ Y)$ if $(w_j > Y)$
$A(j, Y) = max\{A(j-1, Y), r_j + A(j-1\ ,\ Y - w_j)\}$ if$(w_j \leq Y)$.
The solution can then be found by calculating $A(n, W)$. To do this efficiently we can use a table to store previous computations. This solution will therefore run in $O(nW)$ time and $O(nW)$ space.

5.5.2 Greedy Approximation Algorithm:

George Dantzig proposed (1957) a greedy approximation algorithm to solve the unbounded knapsack problem. His version sorts the items in decreasing order of value per unit of weight, r_j/w_j. It then proceeds to insert them into the sack, starting with as many copies as possible of the first kind of item until there is no longer space in the sack for more. Provided that there is an unlimited supply of each kind of item, if A is the maximum value of items that fit into the sack, then the greedy algorithm is guaranteed to achieve at least a value of $A/2$. However, for the bounded problem, where the supply of each kind of item is limited, the algorithm may be very much further from optimal[35].

5.6 Conclusion

Our problem is a 0-1 knapsack problem and we are lucky that there is a solution for this kind of problem as we have presented above.

Part II

Imperfectly Nested Multidimensional Shifting (INMS)

Chapter 6

Overview of The Problem

6.1 Introduction

As we have said in chapter 1, StreamIt is a tool or a language to implement streaming applications. It helps programmers by its C-like language and its phased scheduling module that schedules applications implicitly. Unfortunately the model StreamIt used to build applications is a restricted SDF model, the application graph is built from predefined features, which can not model all SDF streaming applications. Also phased scheduling is not a fine-grained schedule, it is a coarse-grained schedule [30][29].
Unlike StreamIt, we want to develop a scheduling approach which takes in consideration both the code and buffer sizes and respects machine resources and time constraints. The schedule will be a fine-grained one and the applications are any one obeying the SDF model, no restrictions.
In this chapter we cite different specifications and characteristics of applications, actors and schedule and present a global view of the problem and the Imperfectly Nested Multidimensional Shifting (INMS) framework.

6.2 Specifications

We are looking for a schedule for applications that have these characteristics:

- Streaming application computational model: the only condition is to be an SDF application (without the restriction of SteamIt).

- Actor: the basic execution unit, we consider actors consisting only of a basic block of few sequential instructions, no loops and no branch statements. This design point is necessary to build a fine-grained schedule. Coarse-grained actors can be seen as a sub-SDF applications.

- Schedule structure: To schedule SDF applications we schedule the repeated motif, the steady state. Its schedule consists of actor occurrences (firings) represented as strings where actor names are preceded by a number specifying

how many times this actor executes successively. The number with the actor occurrences may form a loop, we call it **phase**. Each phase may be enclosed at any level in a loop nest. Two phases that are enclosed in by the same loop have the same level.

- We are looking for a schedule under these constraints:
  - Code size: the scheduler should guarantee that the generated code satisfies the bound specified by the system designer. It should save code size as much as possible, especially because most SDF applications are embedded ones.
  - Buffer size: because of producer-consumer relations between actors, buffer size is a sensitive constraint. Scheduling too many productions before a consumption may exceed the buffer size.
  - Machine resources: the scheduler tries to execute as many actor occurrences in parallel as possible. To achieve this task it manages machine resources, operation units, registers...The target architecture may be ,for example, a general-purpose or embedded RISC or VLIW processor.
  - Execution time: in a real-time system, to maximize the throughput latency and throughput can be explicitly stated as constraints as well.

6.3 The Global View

The SDF application schedule should respect all constraints cited above and these constraints are not independent: each one may affect the others negatively, minimizing the code size increases the buffer size automatically and vice versa and machine resources uses, implicitly actors firings parallelization, may affect both buffer size and code size. So finding a fine-grained schedule for an SDF streaming application consisting of fine-grained actors, a schedule which respects all constraints and takes advantages of architecture offered parallelism, is to find a compromise between all these constraints. To achieve this aim we propose to divide the problem into two sub problems because it is so difficult if not impossible to treat all constraints at the same time: first, to find a coarse-grained schedule that takes care of code size and buffer size and then to find the final schedule that respects machine resources and takes advantages of architecture offered parallelism without breaking other constraints. We have called the first process *"Pre-scheduling Algorithm"* and the second one *"Imperfectly Nested Multidimensional Shifting"*.
The diagram in fig 6.1 shows the different scheduling process steps and different inputs/outputs:

- **SDF Streaming Application:** it is represented by an SDF graph and the code written by the programmer.
- **Pre-scheduling Algorithm:** it computes the minimal steady state schedule period, there is a minimal steady state for any SDF applications [29] and orders actor firings composing it according to code size and buffer size bounds.

- **Pre-scheduled Steady State:** the output of the pre-scheduling algorithm, the pre-scheduled steady state is exactly a loop nest where actor occurrences replace instructions. We consider loop nests of two levels only. Generalization to any number of levels is one of our future works.

- **Imperfectly Nested Multidimensional Shifting(INMS):** this is the general framework we propose for fine-grain scheduling of SDF. INMS tries to pipeline steady state phases (inner loops), by finding the repeated kernel pattern, and executing actor firings in parallel as much as possible while respecting all constraints.

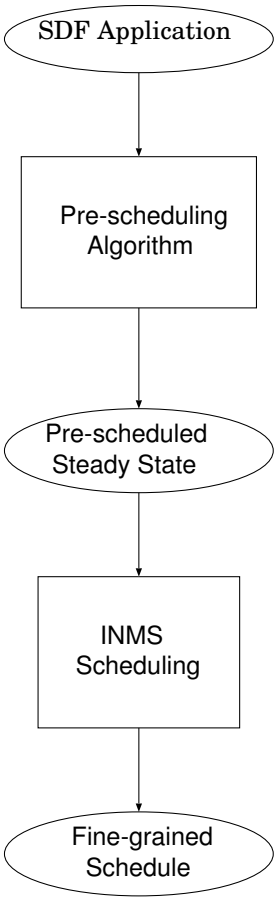


Fig. 6.1. Scheduling Process Diagram

6.4 Pre-scheduling Algorithm

The first step to find a fine-grained schedule for the steady state is the pre-scheduling process. It is to build a schedule that respects code size and buffer size constraints. We propose here an algorithm that can perform this task. Its idea is very simple, it takes a schedule that respects code size constraint only as input, a SAS (Single

Appearance Schedule) where every actor is listed in the schedule only once. Therefore the schedule size is minimal but the buffer size is not optimal and it may not respect the buffer size. Then it decreases buffer consumption till finding a schedule respecting the buffer size constraint as well. It decreases buffer consumption by splitting the succession of occurrences of the most greedy actor. One split operation decreases buffer size considerably without affecting code size by a great change but it is possible that repeating this operation many times ends up increasing code size considerably. For this reason the algorithm checks code size each time it splits an actor. We need to answer the question of which actor should be split. For this reason the algorithm orders actor occurrences according to their consumption of buffer, '$push$×actor_occurrence_number' where $push$ is the number of tokens produced by an actor firing (see section 3.3) and chooses each time the most greedy actor, defined as the succession of actor occurrences consuming the highest amount of buffer space. Then it decreases its repetition factor to decrease buffer consumption. Our algorithm is greedy, it takes into account both the intrinsic push amount and the repetition factor of the most greedy actor. When we split an actor (the succession of occurrences) the first copy keeps its place in the schedule and its factor becomes the old _factor divided by 2 if old_factor is pair and old _factor divided by 2 plus 1 if impair, the second copy is postponed to the end of the schedule and its factor is old _factor divided by 2. All actor firings that depend on the second copy will take their places after it in the schedule, it is exactly what the function **modif_sched** does. By repeating this procedure and verifying each time buffer and code sizes we will achieve to build a valid schedule satisfying all the resource constraints if one exists.
For example $2A5B$ is a steady state where $push_A = 5$ and $pop_B = 2$, A the producer and B the consumer. The SAS schedule of this steady state is $2A5B$. Now suppose that the buffer size is less or equals 8 tokens. Because A executes twice successively so it produces 10 tokens then the consumer starts its execution to consume them. The greedy actor firings sequence is $2A$, 2 the factor and A the actor firing and $push_A = 5$. To decrease the buffer consumption the algorithm splits $2A$ sequence to two copies and put the second one at the end of the schedule followed by B firings that depends on it. The Schedule becomes $A2BA3B$. See the algorithm below

input: an SDF Graph, it can be represented by a matrix where columns represent nodes and rows represent arcs (see section 2.4.3).

output: the output is the steady state schedule, actor firings sequence. It can be represented by a string or ordered array.

support functions:

```
/*comput_steady_state( ): computes the steady state of the input SDF graph,
 the output is an actor firings sequence */
 Vec[ ] comput_steady_state(SDF_graph);

/* SAS( ): computes a Single Appearance Schedule, if it exists it generates
```

```
 a firing sequence from the steady state and SDF graph */
vec[ ] SAS(vec[ ] steady\_state , int SDF\_graph[ ][ ]);

/*greedy_node_sort( ): order actors from 'greedy' to the less one according
to nb_actor_occurrences*push*/
char[nb_actor] greedy_node_sort(vec[ ] steady_state , int SDF_graph[ ][ ]);

/*modif_sched( ): it modifies the schedule by making the actor appear many
times in the sequence*/
modif_sched(steady_state , SDF_graph[ ][ ] , nb_exec[ ] );

/*check_buf_size( ) and check_code_size( ): they compare schedule code size
 and buffer size with the available buffer or memory*/
check_buf_size(new_sched, SDF_graph[ ][ ], buf_size);
check_buf_size(new_sched, SDF_graph[ ][ ],code_size);
```

Algorithm

```
  comput_steady_state(SDF_graph[ ][ ]);
/*if SAS doesn't exist, pick any actor firing sequence satisfying
the code size constraint, or fail if no one is found*/
  new_sched = SAS(steady_state, SDF_graph[ ][ ]);
  old_sched = null;
  valid_buf = false;
  valid_code = true;
  greedy_node_sort(new_sched, SDF_graph[ ][ ]);
  nb_actor_firings = length(new_sched);
  i=0;
  while(i <= nb_actor_firings - 1)&&(!valid_code or !valid_buf)
     {
        if (nb_exec[i]==1) // i is the index of the greedy actor
            i=i+1;
        else
         {
             t = nb_exec[i] ;
             nb_exec[i] = nb_exec[i]/2;
             old_sched = new_sched;
             new_sched = modif_sched(new_sched, SDF_graph[ ][ ], nb_exec[i], t);
             valid_buf = check_buf_size(new_sched, SDF_graph[ ][ ], buf_size);
             valid_code = check_buf_size(new_sched, SDF_graph[ ][ ], code_size);
             if (!valid_code)
               {
                  new_sched = old_sched;
                  i = i + 1;
               }
             else if (!valid_buf)
              {
                 greedy_node_sort(new_sched, SDF_graph[ ][ ]);
                 nb_actor_firings = length(new_sched);
```

```
                i=0;
            }
        }
    }
if (valid_code && valid_buf) return new_sched;
```

Algorithm Termination: the algorithm terminates in all possible cases:

- the algorithm may find a schedule with ($valid\_code = true$) and ($valid\_buf = true$) and the algorithm terminates.

- else either the algorithm continues decreasing buffer size till all actor firing factors equals one then the algorithm stops its executions because the i counter will continues its incrementation till it becomes greater than '($nb\_actor\_firings-1$)' in this case.

- or it is the case where code size is not valid (($valid\_code = false$), so either we find another greedy actor firing that verifies the code size and buffer size or the i counter continues its incrementation till it becomes greater than '($nb\_actor\_firings - 1$)' and in both cases the algorithm termination condition is verified because the algorithm stops its executions if ($i > nb\_actor\_firings - 1$).

6.5 Imperfectly Nested Multidimensional Shifting

Imperfectly nested Multidimensional Shifting (INMS) is the heart of the process. It tries to find a fine-grained schedule for the pre-scheduled steady state, the output of pre-scheduling algorithm, which is exactly a loop nest where actor occurrences replace instructions. So the task of **INMS** is to schedule this special loop nest; the number of phases or loops by level may be one or more, the loop nest may be a perfect one or not and many actor occurrences may appear in the same phase (see Sections 6.3, 6.4).
INMS is a multidimensional shifting technique: it shifts actor occurrences across the phase to break dependences between actor firings of the same phase (intraphase dependences) to extract parallelism. We call this process "Phase Parallelization". Then it tries to move out prologs and epilogs generated by the first process or combine them with other prologs epilogs to build some kernel iterations, we call this process 'Phase Prolog-epilog Moving'. INMS takes care of interphase dependences as well by checking interphase dependences each time 'Phase Prolog-epilog Moving' is applied and shifting actor occurrences when necessary. We call this INMS process "Prolog-epilog Moving Effect on Other Phases".

6.5.1 Phase Parallelization

There are a lot of work on loop parallelization. The most important one for our case is software pipelining. It looks like our problem. It seems that it can solve it but unfortunately not. Software pipelining schedules one loop only and not a steady state, imperfectly nested loops with inter-phase dependences. The first task of INMS is to software-pipeline as much phases as possible under machine resource constraints. The output of this process is a loop nest with many pipelined phases. Unfortunately the code size constraint may be broken down because software pipelining generates prologs and epilogs inside the global loop and this may increase the code size considerably.

SDF to DDG transformation It is not possible to apply software pipelining on SDF directly because software pipelining is proposed for DDG graph (Data Dependence Graph) which is completely different from SDF graph. There is a great distinction between them: a dataflow arc in the SDF between two actors translates into multiple dependence arcs connecting different occurrences of these actor in the DDG (see Fig 6.2). Then, in order to apply software pipelining algorithm we need first to transform the SDF graph to a DDG graph, we build an actor occurrences graph for each phase. This will not increase the code size in reality, authors of paper [58] find that, for most streaming applications, dataflow dependences are $1-1$ dependences. This means that the number of tokens the producer produces equals the number of tokens the consumer consumes.

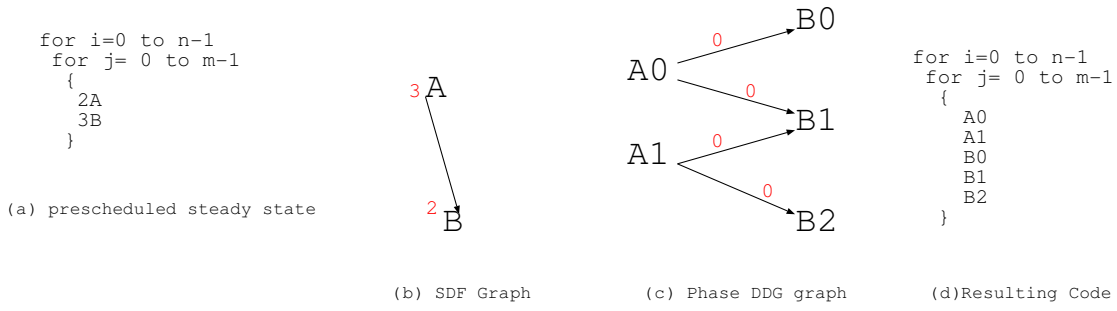


Fig. 6.2. SDF to DDG Transformation

6.5.2 Actor Firing Index

An actor occurrence is defined by its global loop iteration i, its intern loop lp, which iteration of this loop j and the actor occurrence number in this iteration, r. So an occurrence of the actor A is the quadruplet $A(i, lp, j, r)$. In the clock system (hour: min: sec), we can always convert hours and minutes to seconds to have a single value, a linearized index. We can do the same thing with actor occurrences. The

linearized index of an actor occurrence $A(i, lp, j, r)$ is:

$$k = i \times nb\_A(Glp) + \sum_{l=1}^{lp-1} nb\_A(l) \times (l\_upbound + 1) + nb\_A(lp) \times j + r \quad (6.1)$$

where:

- $nb\_A(Glp)$ is the number of A occurrences in the global loop, one iteration of the global loop precisely
- $nb\_A(l)$ is the number of A occurrences in the loop l and $l\_upbound$ is the upper bound of loop l iterator.
- $nb\_A(lp)$ is the number of A occurrences in the loop lp

We can index or order firings (occurrences) of an actor A according to their indexes. Fig. 6.3 shows an example of actor firings ordering.

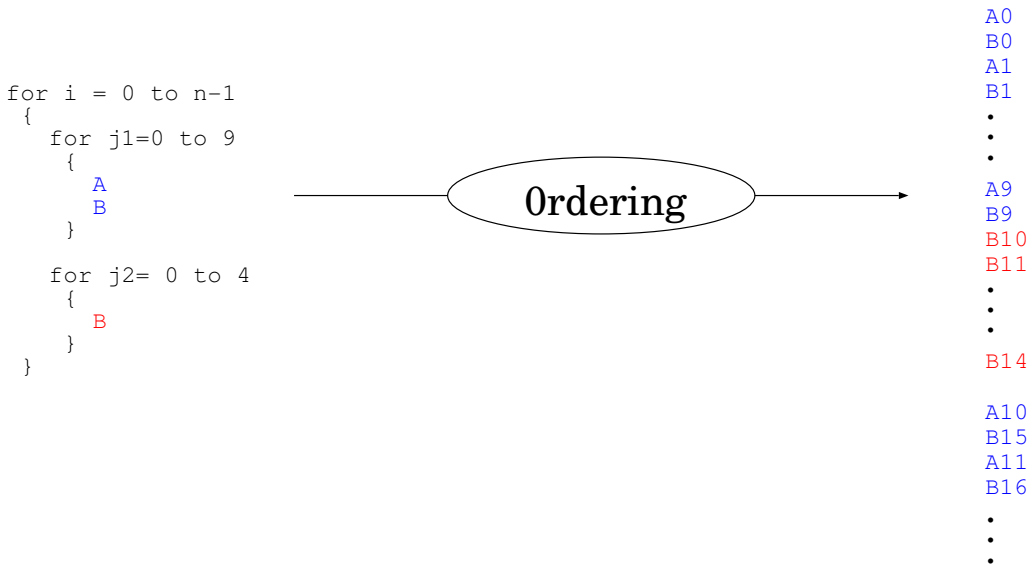


Fig. 6.3. Actor firings ordering example

6.5.3 Phase Prolog-epilog Moving

The example in Fig. 6.4 shows a pre-scheduled steady state where its phase is software pipelined. As we see in this example, generated prolog and epilog increase the code size and this could break down code size constraint. To solve the problem we need to make the prolog and the epilog disappear from the global loop but how to do that? We propose two different solutions: **"Phase Prolog-epilog Merging"** and **"Phase Epilog Fill-in by other Phases"**.

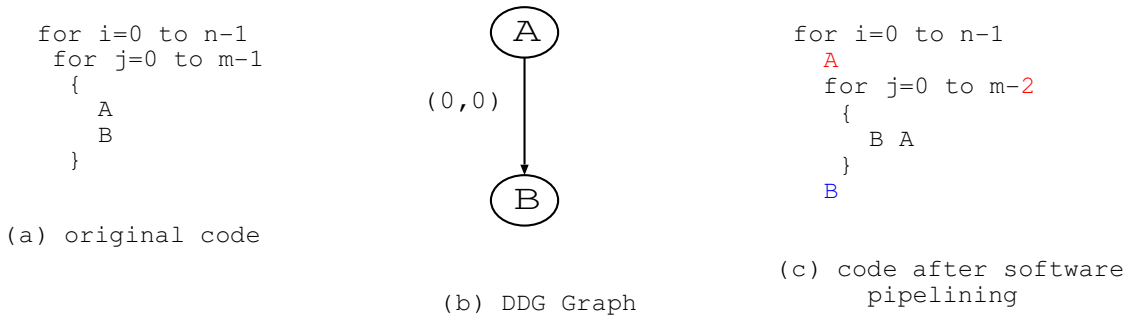


Fig. 6.4. Phase Pipelining Example

6.5.3.1 Phase Prolog-epilog Merging

To explain this idea let's see the example in Fig. 6.5. Fig. 6.5(a) shows three iterations of a global loop enclosing a pipelined phase. Merging the epilog of iteration $(i = 0)$ with the prolog of iteration $(i = 1)$ forms one kernel iteration, see Fig. 6.5 (b). The first global iteration finishes by B and the second one starts by A, so if we just move the prolog of iteration $(i = 0)$ out the global loop and merge the prolog of iteration i with the epilog of iteration $(i - 1)$ then add these new kernel iterations to the kernel of iteration $(i - 1)$ the prolog and epilog generated by phase pipelining process will disappear (see Fig. 6.5(c)).
So "Phase Prolog-epilog Merging" omits phase prolog and epilog by moving out the phase prolog of the first iteration of the global loop and then merging prolog of iteration i with epilog of iteration $(i - 1)$.

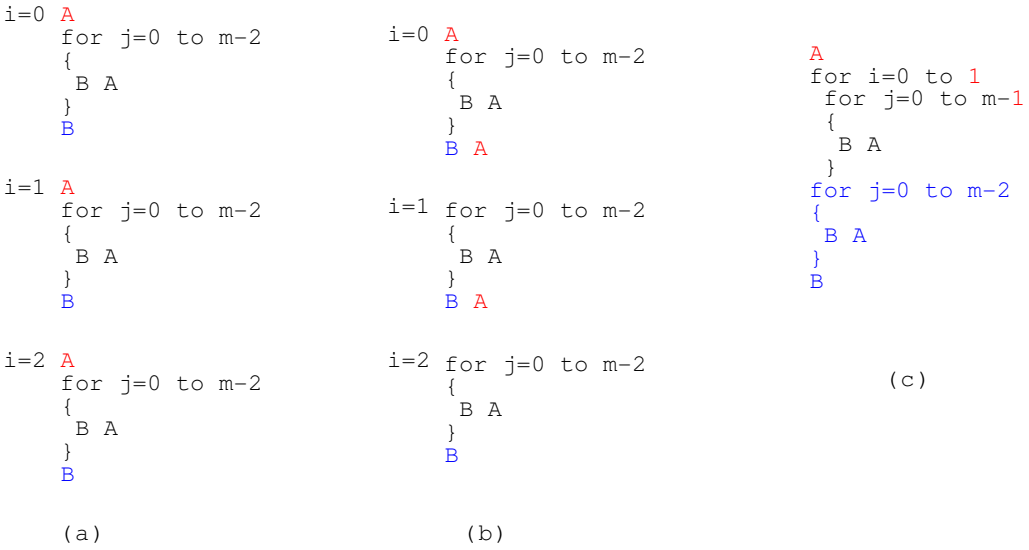


Fig. 6.5. Phase Prolog-epilog Merging Example

Remark To distinguish actors that are going to be moved out the global loop, each actor appears in one phase only. If an actor appears in two phases or more

then we need to give to each actor copy a name different than the others otherwise it will not be possible, in code generation phase, to refer to actors in the global loop prolog and epilog.

6.5.3.2 Phase Epilog Fill-in by Other Phases

A phase prolog or epilog are an incomplete kernel. Adding to the prolog or the epilog lacked occurrences make them disappear from the global loop code. "Phase Epilog Fill-in by Other Phases" does not merge the epilog with the prolog of the the same phase but it looks for lacked actor occurrences of this epilog in the other phases. Let's see the example in Fig. 6.6. The first phase is pipelined and the second not, we suppose that there is a cycle between A and C of phase 2, C depends on A with dependence distance equals 0 and A depends on C of the previous iteration, dependence distance equals 1 see Fig. 6.6(a). In Fig. 6.6(b) three iterations of the global loop are written one under the other. For $i = 0$, we remark that the epilog of the first phase,B, lacks one occurrence of actor A to form the missed kernel iteration of its phase and we see that actor A appears in second phase too. Moving one occurrence of A from phase 2 to the epilog of phase 1 will form this missed iteration. This shifting makes the latest iteration of phase two incomplete, it lacks one A occurrence to be complete. The second iteration of the global loop , $i = 1$, starts by the prolog of the first phase, by A, so we can complete the latest iteration of the second phase of the first global loop iteration, $i = 0$, by A prolog of the first phase of the second loop iteration, $i = 1$ as we have done in Fig. 6.6(c). Repeating this process till the latest iteration of the global loop produces code in Fig. 6.6(d). The latest iteration of the global loop is incomplete, it lacks one A, it forms an epilog for the global loop where the prolog is A.

This second solution is better, filling in a loop incomplete iteration by actor occurrences of another loop consumes less memory and variables life span is shorter comparing it with variables in the first solution. The disadvantage of this idea is that it is not always applicable, it is not always possible to compose a phase prolog from other phases, what if needed actors do not exist in other phases?

Remark Examples in Fig. 6.7 and Fig. 6.8 show how actor occurrence are ordered according to the shifting type. We remark that "Phase Epilog Fill-in by Other Phases" shifting respects order(index) of actor occurrences in the original code but "Phase Prolog-epilog Merging" does not, because Shifted iterations are filled in only by occurrences of the same phase then firings of the next iteration of the global loop may execute before firings of the previous iteration.

6.5.4 Phase Prolog-epilog Moving Effect on Other Phases

Before talking about "Phase Prolog-epilog Moving Effect on Other Phases" we need to understand first inter-phase dependences and what happens when we shift an actor occurrence that depends on an actor occurrence of another phase.
Dataflow dependences of streaming applications have a special characteristic: they

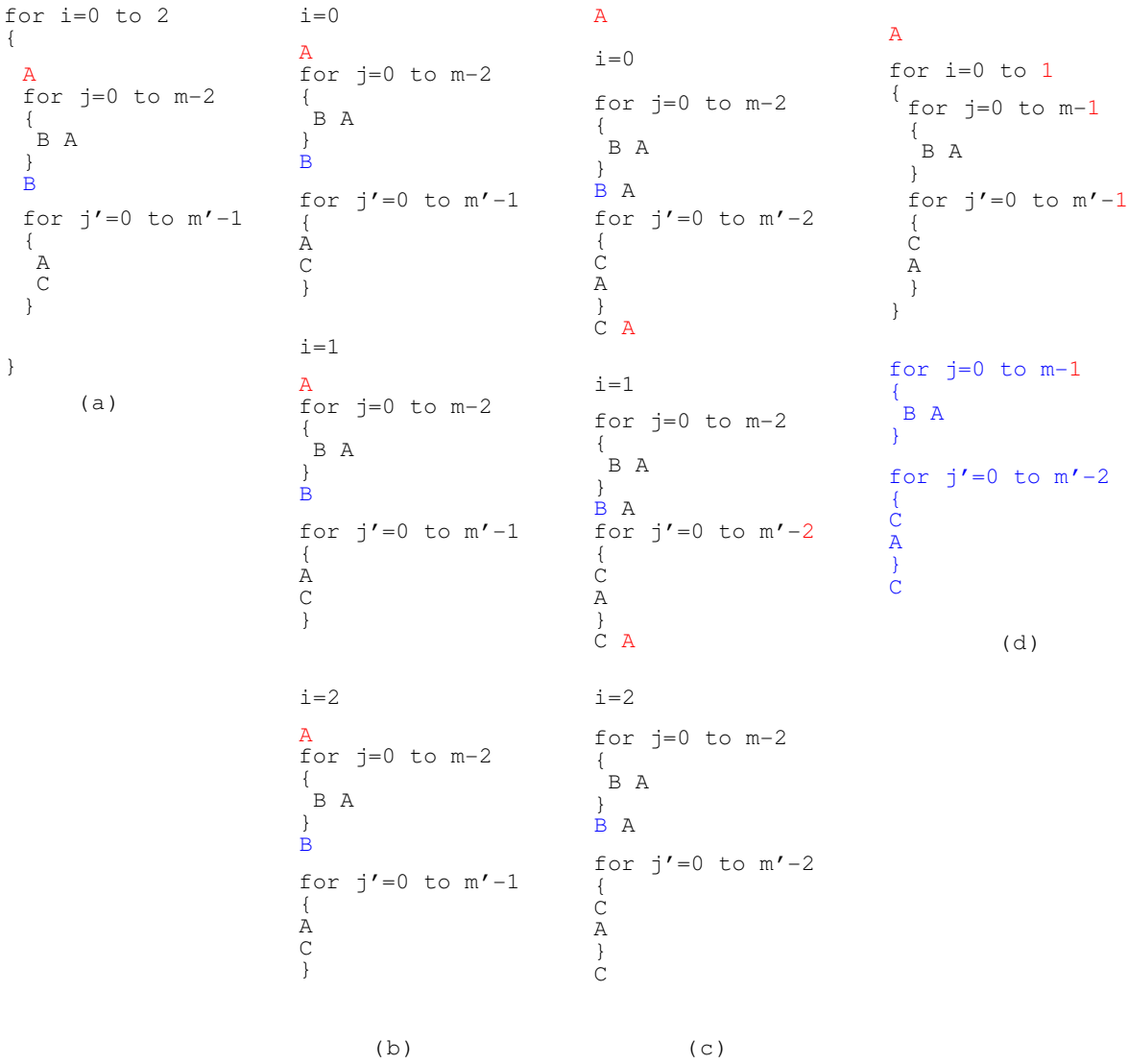


Fig. 6.6. Phase Epilog Filing in by Other Phases Example

can change from one actor firing to another. For instance, if we have this sequence '$2A3B$' as a steady state and $push_A = 3$ and $pop_B = 2$ and B depends on A. The first occurrence of B consumes 2 tokens produced by the first occurrence of A then the first occurrence of B depends on the first occurrence of A. The second occurrence of B depends on the first and the second occurrences of A and the third occurrence of B depends on the second occurrence of A. So what is the dependence rule we will consider to build the pattern? Then dependence between two actor occurrences is not like dependence between two instructions and for '$2A3B$' example we can only say that $3B$ depends on $2A$, 3 executions of B depends on 2 executions of A. So shifting one B occurrence implies shifting all A occurrences, to avoid any dependence stealing.

We will see how INMS selects actor occurrences that are in dependence (inter-phase

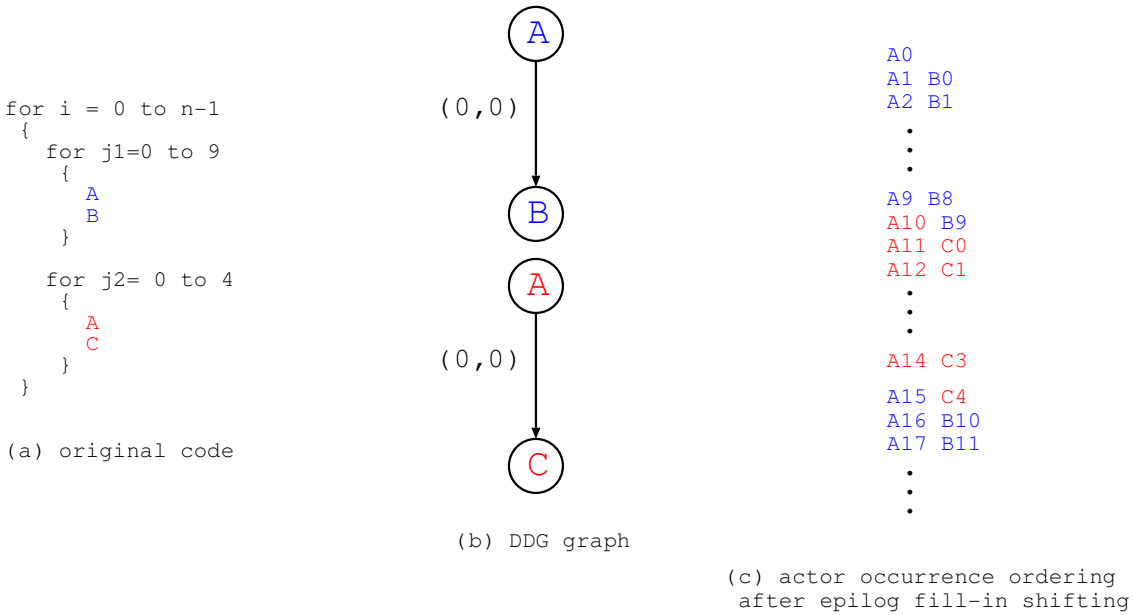


Fig. 6.7. Phase Epilog Fill-in by Other Phases example

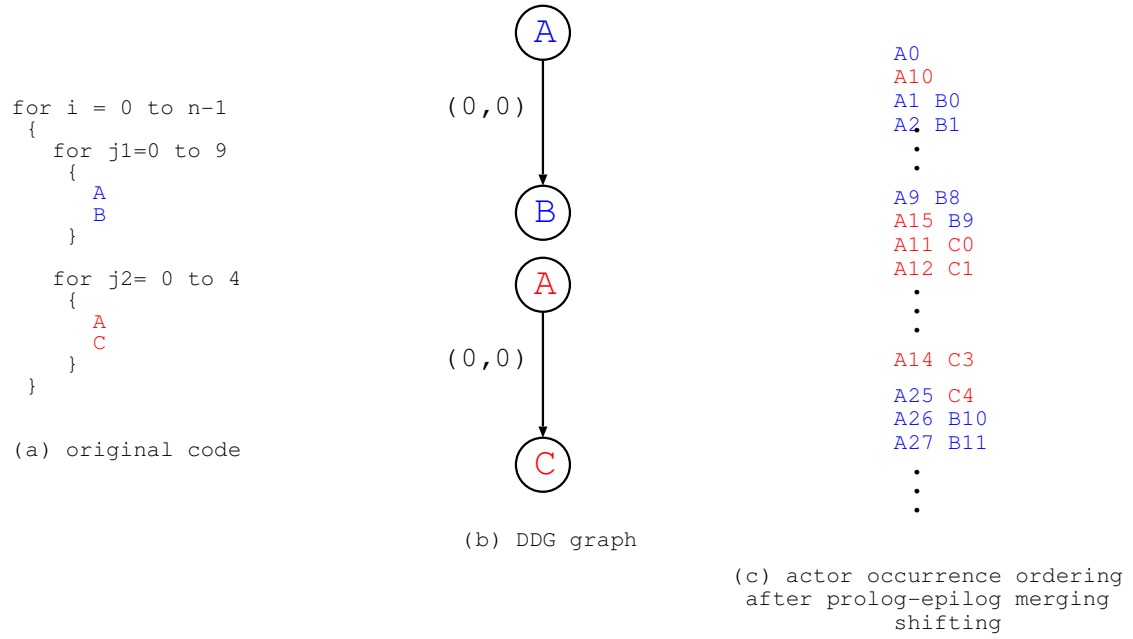


Fig. 6.8. Phase Prolog-epilog Merging Example

dependences) with shifted actors, and how it moves them to the global loop prolog in section 9.2.3.

6.6 Multi-dimensional Shifting Formalization

A is an actor and $A(i, lp, j, r)$ is an A occurrence. Its dimensions i, lp, j and r are dependent like the clock system (hour: min: sec). Shifting the A actor occurrence

by a *shifting distance* c may affect only the dimension r or the inner loop iteration j or the loop index lp or even the global loop iteration i. If the *shifting distance*, c, is less than r so it is simply a translation applied on the dimension r. If not so it is a shifting from one loop iteration to another or completely from a loop to another, exactly like clock system.
INMS tries to find the right translation distance c for each actor occurrence, taking in consideration all kinds of resources and time constraints: code size, buffer size, data dependences, resources (register, operations units) and time (real time application).
Suppose INMS has found the right c for each actor occurrence $A(i, lp, j, r)$ so what is $A(i', lp', j', r')$, the actor occurrence coordinates after shifting? Finding $A(i', lp', j', r')$ is useful for INMS code generation process. To Formalize the shifting we have to find the relation between $A(i', lp', j', r')$ and $A(i, lp, j, r)$. How to shift actor occurrences determines how to compute i', j', lp' and r' using i, j, lp and r. As we have said before, to avoid having prologs and epilogs inside the global loop two multi-dimensional shifting are possible: Filling in the shifted iterations from the same phase only, called "Phase Prolog-epilog Merging" and filling in them from the other phases as well called "Phase Epilog Fill-in by Other Phases". We will see how to compute $A(i', lp', j', r')$ in both kinds of shifting.

6.6.1 Phase Prolog-epilog Merging

In this kind of shifting, the last iteration of a loop is filled in by the first iteration of the same loop of the next global loop iteration. The shifting distance c of an actor A of a loop lp is less than the number of A actor occurrences in this loop lp. Shifting is a transformation of $A(i, lp, j, r)$ to $A(i', lp', j', r')$. Given an actor occurrence $A(i, lp, j, r)$ and a shifting distance c , $A(i', lp', j', r')$ coordinates are computed as follow:

- **computing the coordinate "j'":**

$$j' = (j - \alpha) \mod (upper\_bound(lp) + 1) \tag{6.2}$$

  $\alpha = \lfloor \lfloor \frac{c}{r+1} \rfloor \times (\frac{r+1}{c+1}) \times 2 \rfloor$ and $upper\_bound(lp)$ is the upper bound of the loop lp iterator

  - If $c > r$ then
    - $\alpha = 1$ because $1/2 \leq \lfloor \frac{c}{r+1} \rfloor \times (\frac{r+1}{c+1}) < 1$
    - $j' = (j - 1) \mod (upper\_bound(lp) + 1)$
    - $j = 0$ is a special case because $j - 1$ is negative. For this reason we use the operator modulo in the formula to move this A occurrence to the latest iteration of its loop (the inner loop), of the prior iteration of the global loop.

- If $c \leq r$ then $\alpha = 0$ and $j' = j$. Therefore shifting this A occurrence will affect only the r coordinate.

- **computing the coordinate "i'":**

$$i' = i - \beta \tag{6.3}$$

and $\beta = \lfloor \lfloor \frac{j'}{j+1} \rfloor \times (\frac{j+1}{j'+1}) \times 2 \rfloor$. from (6.3)

- if $(j' \leq j)$ then $i' = i - 0 = i$.
- else $i' = i - 1$, the A occurrence is moved from the global loop iteration i to its prior.

- **computing the coordinate "r'":**

$$r' = (r - c) \mod nb_A(Lp) \tag{6.4}$$

where $nb_A(lp)$ is the number of A actor occurrences in the loop lp. If $r > c$ then $r' = r - c$ else A is moved to the loop lp iteration $j' = j - 1$ and $r' = r - c + nb_A(Lp)$

- **computing the coordinate "lp":**

$$lp' = lp \tag{6.5}$$

lp coordinate does not change, it is a "Phase Prolog-epilog Merging" shifting.

6.6.2 Phase Epilog Fill-in by Other Phases

This shifting technique respects the occurrence order (index). (A, k) and (A, k') two occurrences of the actor A, if $k < k'$ then (A, k) is before (A, k') in the schedule. At the end of shifting, the latest loop iteration will be filled in by the following loops containing the A occurrences. Loop containing occurrences of an actor A are ordered from top to bottom, in an array for example. Shifting operation moves A occurrences from Lp to Lp iff $lp > lp'$. It is possible that shifting a loop by 'c' occurrences moves occurrences of many loops, it depends on the c value and A actor occurrence number by loop.

Because it is possible that $c > nb_A(lp)$, $nb_A(lp)$ is the number of A occurrences in the loop or phase lp, then two shiftings are possible: from one loop iteration to another or from a loop to another. Each case is solved separately. Let's see both cases:

- **Shifting from a loop to another**

  - **Computing the coordinate "lp'"**

$$lp' = (lp - 1) \mod nbr(lp) = \gamma \tag{6.6}$$

Where $nbr(lp)$ is the number of loops containing the A occurrences. We use mod operator to include the case where moving an A occurrence from one global iteration to the previous one.

- **Computing the coordinate "j'"**

$$j' = upperbound(lp') + 1 - M = \alpha \tag{6.7}$$

shifting from one loop to another means that c is superior than $j \times nb_A(lp) + r$, $nb_A(lp)$ is the number of A occurrences in lp, so A should be firstly shifted in its loop by $j \times nb_A(lp) + r$ and then shifted in the first loop containing A by $c - j \times nb_A(lp) + r$. with $M = \lfloor \frac{c-(j \times nb_A(lp)+r)}{nb_A(lp')} \rfloor$

- **Computing the coordinate "r'"**

$$r' = (c - (j \times nb_A(lp) + r)) \mod nb_A(lp') = \beta \tag{6.8}$$

We divide by $nb_A(lp')$ and then the remaining is r', $nb_A(lp')$ is the number of A occurrences in lp'

- **Shifting from one loop iteration to another:** means shifting across the same loop but c may be big enough to move A by many iterations.

  - **Computing the coordinate "j'"**

$$j' = j - \lfloor \frac{c}{nb_A(lp)} \rfloor - \lfloor \lfloor \frac{\sigma}{r+1} \rfloor \times (\frac{r+1}{\sigma+1}) \times 2 \rfloor = \alpha' \tag{6.9}$$

with $\sigma = c \mod nb_A(lp)$ shifting across the same loop with 'c' superior than one iteration occurrences means to shift A by c divided by A occurrences number in one iteration and if the remaining is superior than r as well so we will shift one iteration more.

  - **Computing the coordinate "r'"**

$$r' = (r - c) \mod nb_A(lp) = \beta' \tag{6.10}$$

The number of iterations c contains doesn't affect r' at all, because it depends on the remaining of division and r

  - **Computing the coordinate "lp'"**

$$lp' = lp = \gamma' \tag{6.11}$$

because we talk about shifting across the same loop (phase)

To get the general formulas for j', r',lp' we have two factors y and y', the first one equals 1 for the first case and 0 in the second case and the second factor y' is the opposite of it, it equals 0 for the first case and 1 in the second case. Only one of y and y' zero or one at the same time. With the help of y and y' we have:

$$y = \lfloor \lfloor \frac{z}{c+1} \rfloor \times (\frac{c+1}{z+1}) \times 2 \rfloor \tag{6.12}$$

$$y' = \lfloor \lfloor \frac{c}{z+1} \rfloor \times (\frac{z+1}{c+1}) \times 2 \rfloor \tag{6.13}$$

with $z = j \times nb_A(lp) + r$

$$j' = \alpha \times y + \alpha' \times y' \tag{6.14}$$

$$r' = \beta \times y + \beta' \times y' \tag{6.15}$$

$$lp' = \gamma \times y + \gamma' \times y' \tag{6.16}$$

The computation of i' depends only on lp', if it is inferior than lp so it is in the same global loop, $i' = i$ else it is in the global loop iteration $i' = i - 1$.

$$i' = i - \lfloor \lfloor \frac{lp'}{lp+1} \rfloor \times (\frac{lp+1}{lp'+1}) \times 2 \rfloor \tag{6.17}$$

6.7 INMS Implementation

We have presented INMS but we have not said how to realize it. We will see in chapters 7 and 8 two different ways of INMS Implementation

- **Pattern Table Shifting:** a heuristic applicable in case of "Phase Epilog Filing in by Other Phases". It aims to present a polynomial execution time solution. It transforms the SDF graph to Data dependence graph, DDG, to apply later software pipelining on each phase. The result of this process is kernel tables, one table by phase representing the execution instance of each actor occurrence within the pattern. Shifting idea is applied here by moving all actor occurrences in tables to the same column, the same actor occurrences column for all phase tables.

- **Prolog-epilog Merging:** we find it the best implementation of INMS. It is applicable in case of "Phase Prolog-epilog Merging".

.

Chapter 7

Pattern Table Shifting

7.1 introduction

Solving the problem, INMS implementation, without dividing it into sub problems is not possible because we can't always pipeline all phases. Therefore it is necessary to find phases to be pipelined then apply prolog-epilog merging. We can look at the problem as three different sub problems: phases parallelization, prolog-epilog merging and code generation problems. Because we can shift firings in two different ways, therefore we can look at the problem in two different ways according to the shifting technique, "Phase Epilog Fill-in by Other Phases" or "Phase Prolog-epilog Merging". In this chapter we will see the first kind of shifting, "Phase Epilog Filling in by Other Phases" and its implementation "Pattern Table Shifting". The solution we present here is very limited and it is not useful in the general case. We present it here because it is the first idea we have thought about and it results in shorter variable lifetimes.

7.2 Heuristic

Software pipelining may be represented by a kernel (pattern) table that gives the execution instance of all kernel actor occurrences (instructions) where all actor occurrences that have the same row number execute in parallel. The column number is the iteration number of the actor occurrence in the original loop code, before software pipelining, see the example in Fig. 7.1. One thing we have thought about is to use this pattern table in our shifting, a heuristic that may solve the problem in a polynomial time. The idea is to apply software pipelining to each phase of the pre-scheduled loop nest, the output of the pre-scheduling algorithm, and then shift the resulting kernel tables to have as a result all occurrences of the same actor, in different tables, in the same column. Fig. 7.2 shows INMS implemented using 'Patterns Tables Shifting' heuristic.

As we see in Fig. 7.2, the first thing to do is SDF to DDG Transformation (see section 6.5.1) because software pipelining is proposed for DDG graph (Data Dependence Graph), which is different from SDF graph.

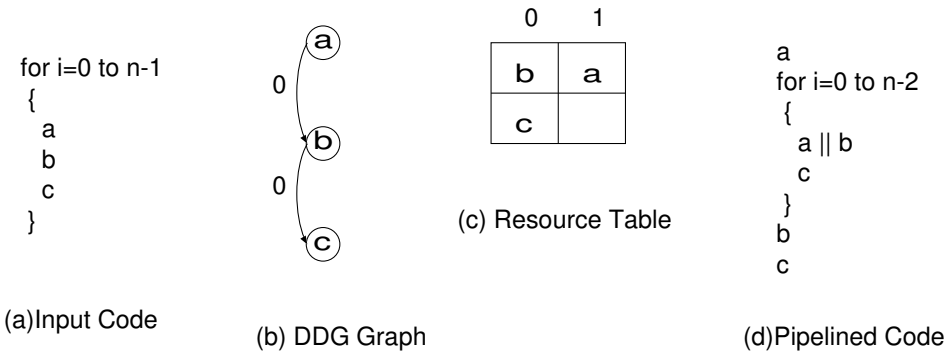


Fig. 7.1. Resource Table Example

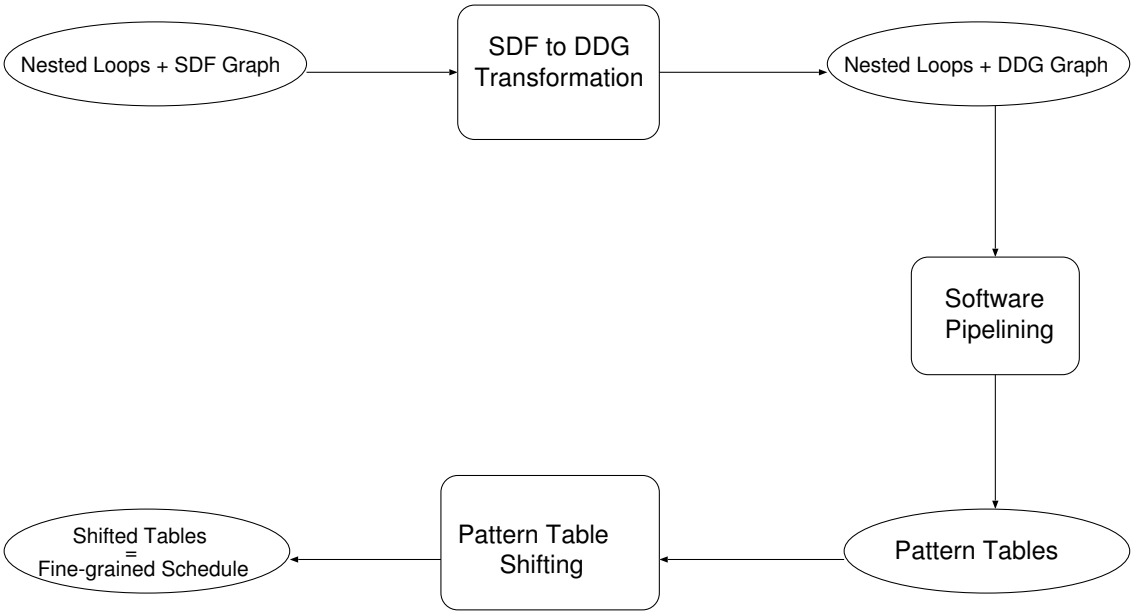


Fig. 7.2. INMS Implementation with 'Pattern Table Shifting'

7.3 Pattern Table Shifting

The 'Patterns Tables Shifting' is a way of implementing "Phase Epilog Filling in by Other Phases" shifting technique.To do that, the "Pattern Table Shifting" tries to move all the actor occurrences that appear in different pattern tables to the same column. Thus the number of occurrences executed before patterns, for this actor, is the same for all phases. 'Pattern table Shifting' is applied only on columns. Actor row change is not permitted, it increases the kernel span .
In the example depicted by Fig. 7.3, after applying software pipelining to the first phase, the result is a pattern table showing when each actor occurrence starts its execution. B of column 0 and A of column 1, both of them in the same row signifies that the A occurrence executes in parallel with the B occurrence of the prior iteration. Hence, the prolog equals A. In the first table, A is in the column number 1 and in the second table is in the column number 2. Shifting A of the first table to

the column number 2 makes both A_0 and A_1 execute before the pattern. Also B in the first table is in the column number 0 and in the second table is in the column number 1. Shifting B of the first table to the column number 1 makes B_0 of both phases execute before their kernels, they are in the prolog. So the prolog is $2AB$ for both phases, the epilog of the first phase is B and the epilog of the second phase is $B2C$. The fusion of the first phase epilog with the second phase prolog, of the same global loop iteration, completes the first phase by two kernel iterations,$2(AB)$, and $n-3$ becomes $n-1$. Merging the second phase epilog, of the global loop iteration i, with the first phase prolog, of the global loop iteration $i-1$, adds two kernel iterations to the second phase, $2(CBA)$, and and $m-3$ becomes $m-1$. In this way all prologs and epilogs generated by software pipelining disappear from the global loop.

This example shows the easiest case, no cycles and actors composing a prolog may be found in other phases. In most general cases an SDF graph may contain cycles and an actor may appear in a phase and not in others, 'Pattern Table Shifting' can do nothing in this case.

7.3.1 Cycle

Cycles in SDF graphs may prevent shifting of pattern tables, because cycle nodes depend on each other, then shifting one node may imply shifting all cycle nodes infinitely. In Fig. 7.4, the example is a case where shifting is not possible because of the cycle formed by dependences. In the steady state $m_1(ACB)m_2(ACD)$, shifting the actor C of the first table to the column number 3 implies shifting A , because C depends on A and dependence distance is 0, and shifting A implies shifting B because of dependence distance value,0, which implies shifting C and the shifting continues till no actor remains in the kernel.

7.4 Simple Case Algorithm

We will present here an algorithm for the simple case, no cycle.
A node in the pattern table may be in dependence relation with other nodes of the same pattern table or of other patterns. A question arising here: if an actor is shifted, are there any effects on the other actors that are in dependence relation with this actor, the nodes the actor depends on them and nodes that depend on the actor? Within a phase:

- if A depends on B, shifting A implies we should shift B especially if A_i depends on B_i because B_i should execute before A_i.
- if B depends on A, we don't need to shift B because A still execute before B

This discussion is about a dependence between two actors, what about all the dependence graph? How to shift all tables respecting all dependence relations? We propose an algorithm that is able to perform this task. It generates shifted tables

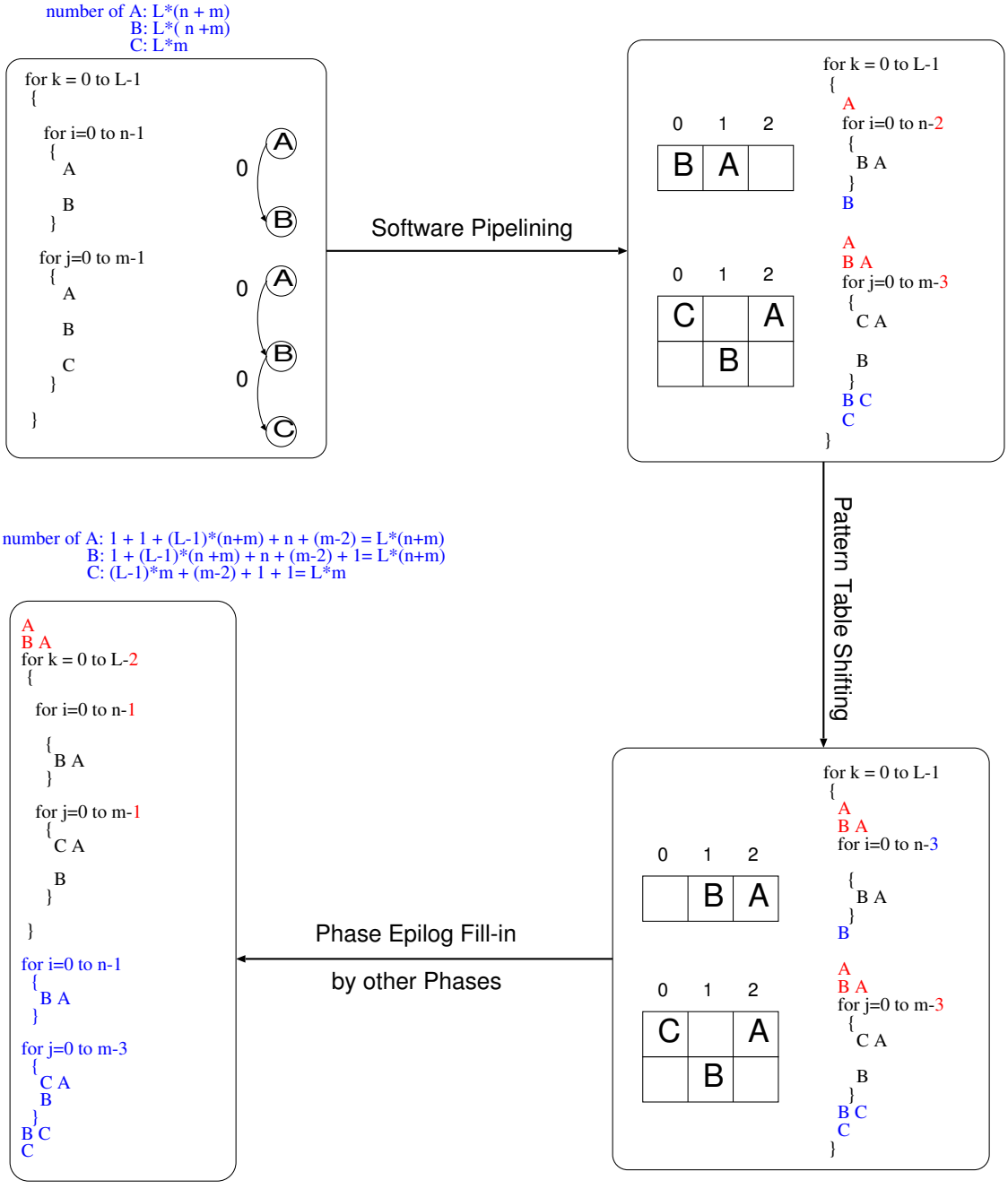


Fig. 7.3. Pattern Table Shifting Example

from software pipelining pattern tables but it is applicable for acyclic SDF graph only.

7.4.1 Algorithm

We will present the algorithm in an informal way.
input: pattern tables and an acyclic specific SDF graph, interphase dependences removed from the original SDF graph because interphase dependences are not treated

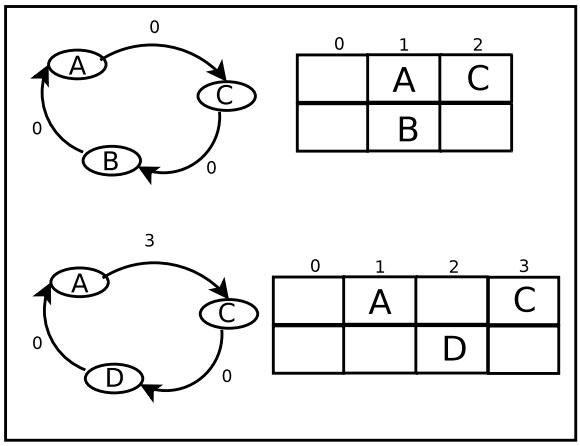


Fig. 7.4. Cycle Example

here and we don't need to do so, in another way, this acyclic SDF graph is the union of all phases DDG graph
output: shifted pattern tables

1. construct a set L of leaves, nodes that no node depends on them.
2. for every leaf A of the set L, do:
   (a) find the maximum of its column numbers CL, if A appears in several phases then it has a specific column number for pattern table
   (b) if for all tables containing A, the box with $column\_nbr = CL$ and $row\_nbr = row\_nbr(A)$ is empty, then move A, in all pattern tables, to this column CL.
   (c) else for all tables containing A, find the empty box with $column\_nbr = CL + d$ and $row\_nbr = row\_nbr(A)$ and move A, in all pattern tables, to this column $CL + d$.
   (d) remove A from L and from the graph
3. if the graph is not empty then
   - for every nodes B that depends on one or several leaves:
     - $max\_leaves\_column\_nbr$ equals the maximum column numbers of all leaves A that depend on B
     - if $((column\_nbr(B) + dep\_dist(B, A)) \leq max\_leaves\_column\_nbr)$ then
       * if the box $(row\_nbr(B)\ ,\ column\_nbr(B))$ is empty, $column\_nbr(B) = (max\_leaves\_column\_nbr\ -\ dep\_dist(B, A)) + 1$, then move B to this box to let it execute before A.
       * else look for the nearest empty box and move B to it, its $column\_nbr(B)$ is $(max\_leaves\_column\_nbr\ -\ dep\_dist(b, A))\ +\ d$

- go to 1

4. omit empty columns
5. end

7.4.2 Running Example

We are going to see a complete execution of the the algorithm here.

- Fig. 7.5(b) shows an example of software pipelining applied on pre-scheduled steady state phases. The DDG graph of each phase is in Fig. 7.5(a), after SDF to DDG transformation of course. The graph in Fig. 7.5(c) represents the global graph, the union of all phase DDGs.

- The algorithm takes this acyclic graph as input and constructs a set of leaves L. For the first iteration of the algorithm, L contains only one leaf, D. In the graph of Fig. 7.6(a), this leaf is in red and its edge is in blue. Because D appears one time only, in the fourth phase, so no pattern tables shifting is needed for it, see Fig. 7.6(d). Then the algorithm removes D from L and from the global graph and looks for nodes that the leave D depends on them, the algorithm finds one node only E, it is in blue in Fig. 7.6(c). E column number is greater than D one, dependence relation is verified so no E shifting is necessary.

- The graph is not empty so the algorithm starts its second iteration and creates $L = \{E\}$. E appears in two different pattern tables with two different column numbers, its column number in the third phase is 0 and in the fourth phase is 1 (see Fig. 7.7(b)). First, the algorithm tries to shift E of the third phase to column number 1 but unfortunately this column box is not empty, it contains the actor C. It is impossible to shift it to the box with column number 2 as well, this box is 'occupied' in pattern table of phase 4 by B. So the only nearest box is the one with column number equal three, so E column number becomes 3 for both phases, the third and the fourth, see Fig. 7.7(d). The algorithm continues this iteration by removing E from the graph and from the set L, which becomes empty, and finding nodes that the leaf E depends on them. These nodes are B and C as Fig. 7.7(a) shows, they are in blue. The node B appears two times, the first time in the first phase (column number equals 1) and the second time in the fourth phase (column number equals 2). The nearest empty boxes for both phases, with the same column number of course, are those with column number equals 4 (see Fig. 7.7(f)). The C column number becomes 5 as depicted in Fig. 7.7(g).

- The graph is still not empty. By repeating the same procedure, the set L will contain C. C occurrences are already in the same column in all phases where C appears (see Fig. 7.8(b) and Fig. 7.8(c)). Nodes that C depends on them are B and A. C $max\_column\_number = 5$ and B $column\_number = 4$

so *max_column_number* < *B column_number* then *B column_number* becomes 6, these boxes are empty (see Fig. 7.8(e). The same thing happens with *A* and *A column_number* becomes 7 (see Fig. 7.8(f).

- The graph is still not empty and pattern tables are well shifted, every node in the right place, then the algorithm continues its execution by removing nodes from the graph, *B* and *A*, till it becomes empty. At the end, the algorithm omits empty column generated, $1, 2, 4$, see Fig. 7.9(b) and Fig. 7.9(c)
- The code generated by this transformation is showed in Fig. 7.10. Fig. 7.10(a) shows the first global loop iteration, $i = 0$ and Fig. 7.10(b) shows the second one. Fig. 7.10(c) explains how epilog filling happens, each epilog is filled by prolog actors of phases that are below this epilog in the code, of the same global loop iteration, or by prolog actors of phases that are above it in the code but of the next global loop iteration. The result code is in Fig. 7.10(d)

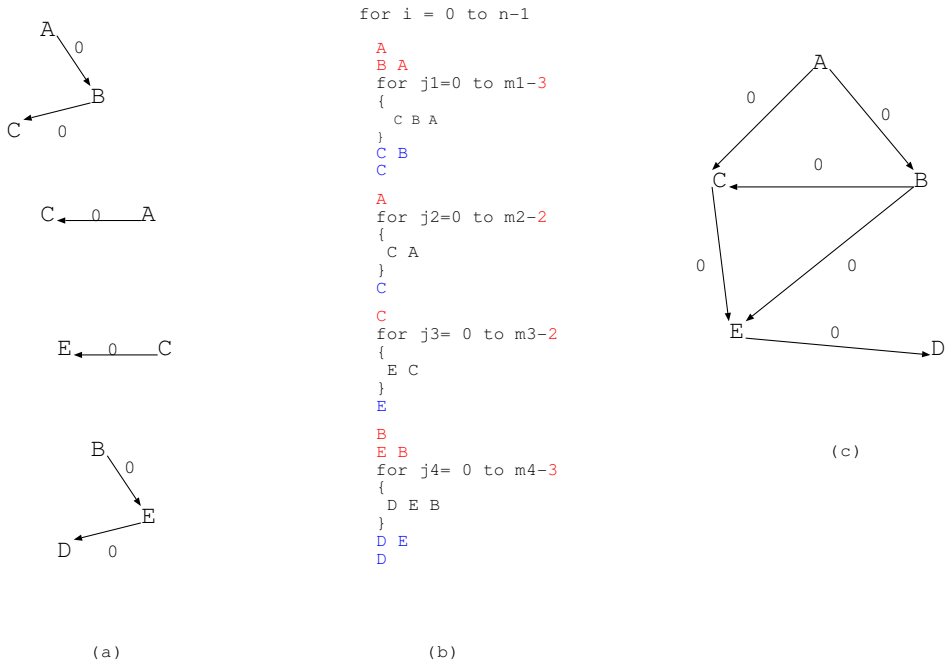


Fig. 7.5. Pre-scheduled Steady State with Phase DDGs

7.4.3 Termination

The algorithm removes each time a leaf from the acyclic graph. Because the graph is acyclic, so we are sure that the algorithm arrives at an execution state where the graph is empty, so it terminates.

7.4.4 Correctness

- The algorithm is always able to provide shifted tables, shifting actor is always possible because we can always find the maximum column number, the

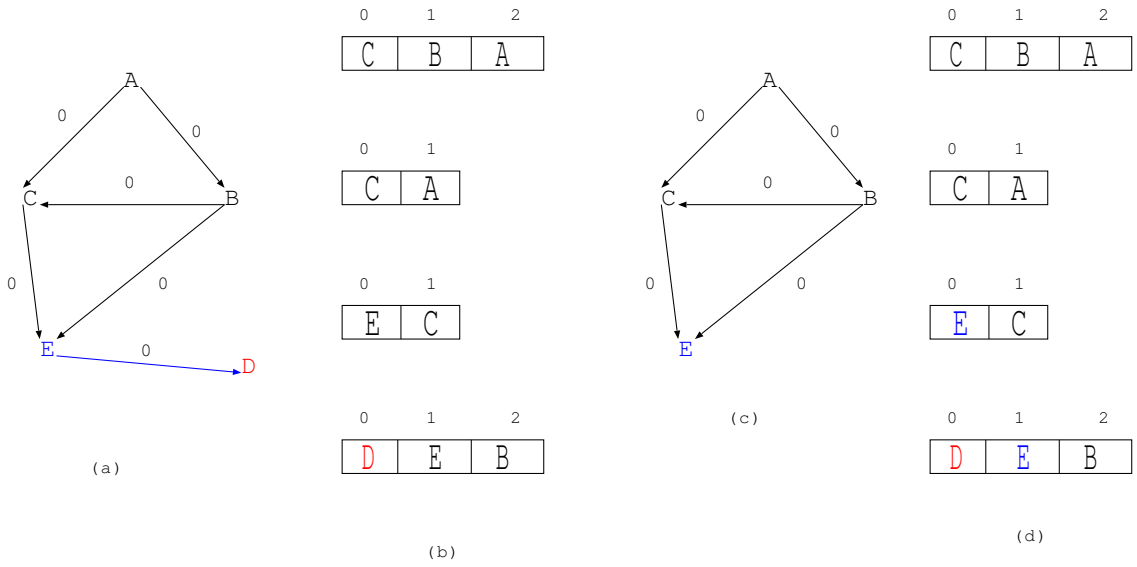


Fig. 7.6. First Iteration of the Algorithm Execution

maximum of numbers always exists and an empty case always exist as well $CL + d$.

- Are these shifted tables a solution? They are a solution if dependence relations are respected. Then we have to check if the algorithm respects dependence relations. The algorithm has two actions: shifting leaves and shifting the nodes that leaves depend on them.

  - if a node doesn't depend on any other node shifting it will not alert dependence relations, so the algorithm is correct;

  - if one or several leaves A depends on B, shifting A may break down this dependence relations but the algorithm corrects that by the second action, by treating all nodes that leaves depend on them and shifting them if it is necessary to meet the dependences relation;

    * if several leaves depend on a node B, the algorithm keeps dependence between B and these leaves by checking dependence between B and the most shifted leaf A, the leaf with the maximum column number. If the dependence is still respected with this leaf, so it is still respected with others because they are less shifted than it. Else the algorithm moves B to the right box to let it execute before the most shifted leaf A. So the algorithm correctness condition is verified. To summarize, because the algorithm always affect to a leaf a column number less than its ascendant nodes, so the dependence relation is always verified.

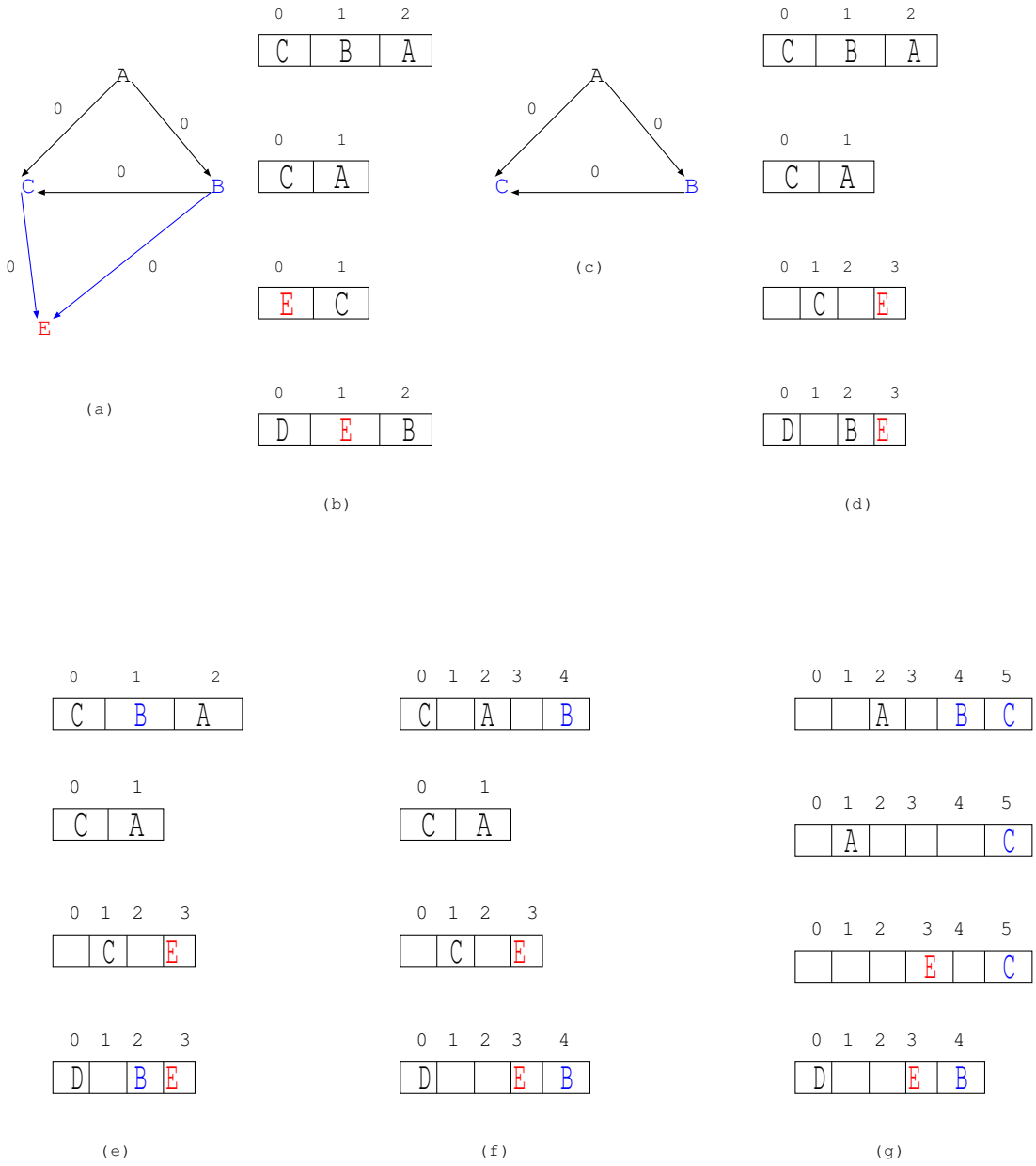


Fig. 7.7. Second Iteration of the Algorithm Execution

7.4.5 Remarks.

Shifting the pattern tables doesn't always mean that we have found the fine-grained schedule. Fig. 7.11 depicts an opposite example. In this example, shifting pattern tables will let the epilog of the first phase B be filled in by A, an actor in the prolog of the second phase. Then the prolog and epilog of the first phase may disappear from the global loop but what about the second loop? To fill in its epilog, two occurrences of actor C are needed but unfortunately the first phase contains no C occurrence. So the code in fig. 7.11(3) is better than the one in fig. 7.11(1),in term of performance, because phases are pipelined but the code in fig. 7.11(1) is better,

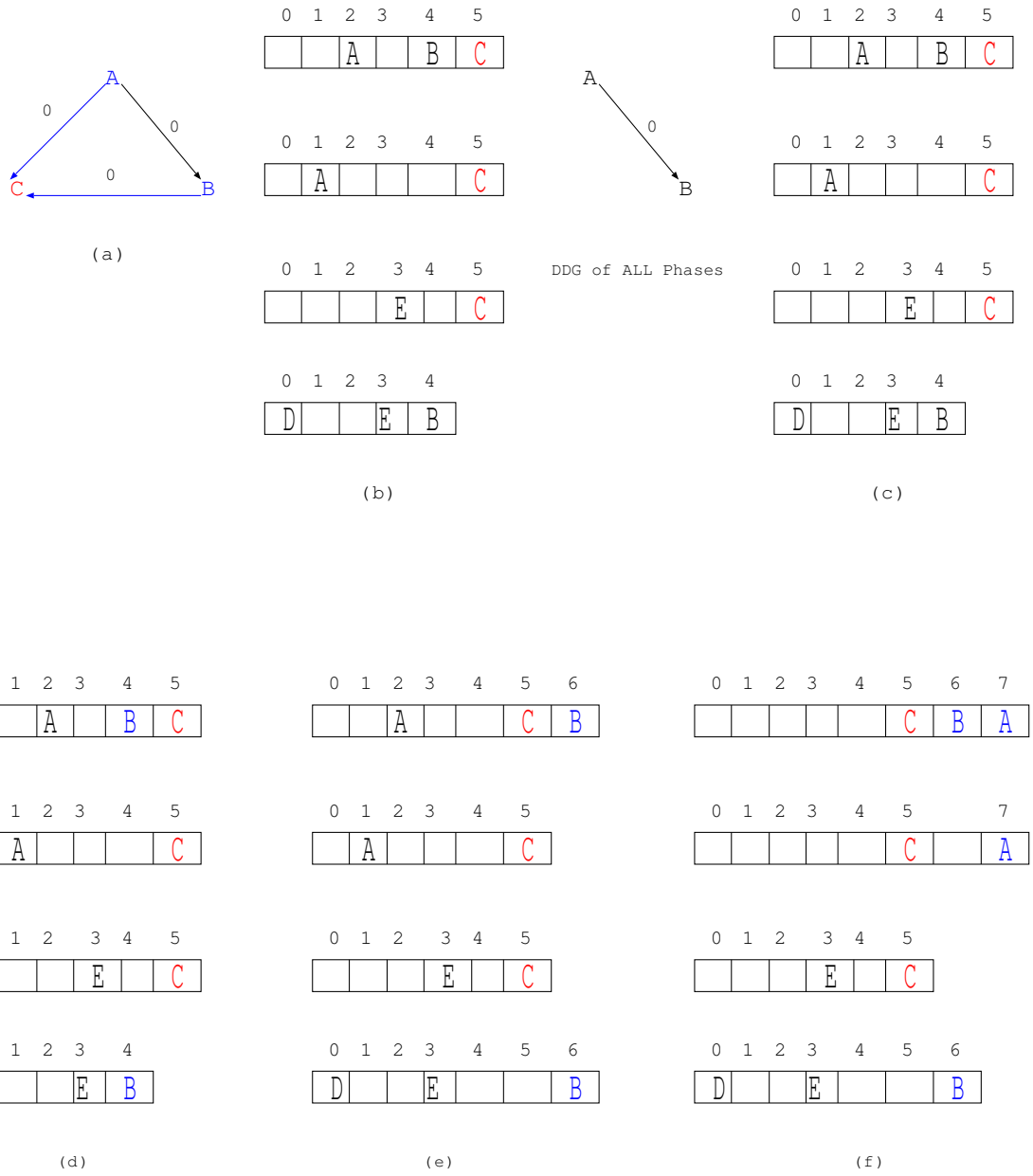


Fig. 7.8. Third Iteration of the Algorithm Execution

in term of code size.

7.5 Conclusion

The heuristic implements **INMS** approach according to the shifting technique we called: "Epilog Fill-in with other Phase Firings" . Unfortunately, it is applicable only for simple cases: no cycle. In addition to this limit, the shifting technique itself,'Epilog Fill-in with other Phase Firings', is limited because it is not always possible to find all actor occurrences composing a phase prolog in other phases.

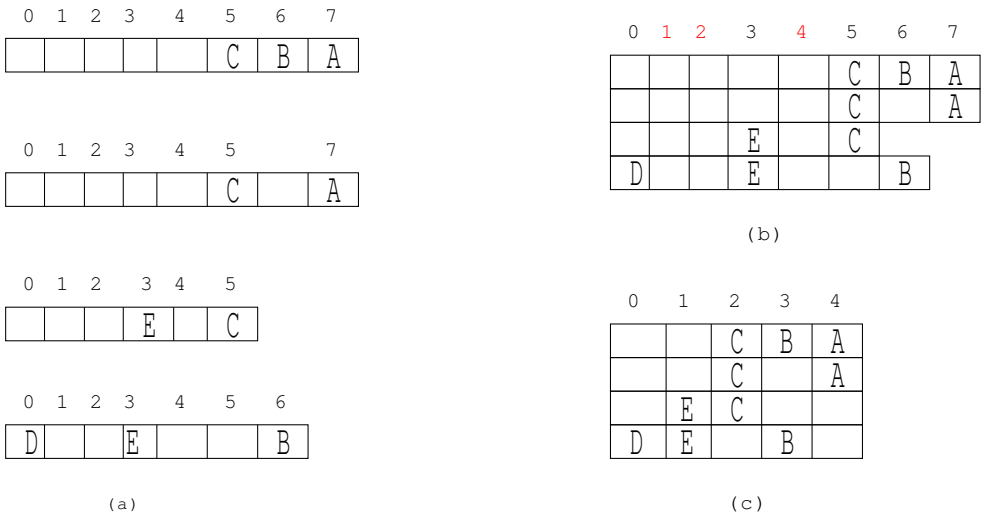


Fig. 7.9. End of Algorithm Execution

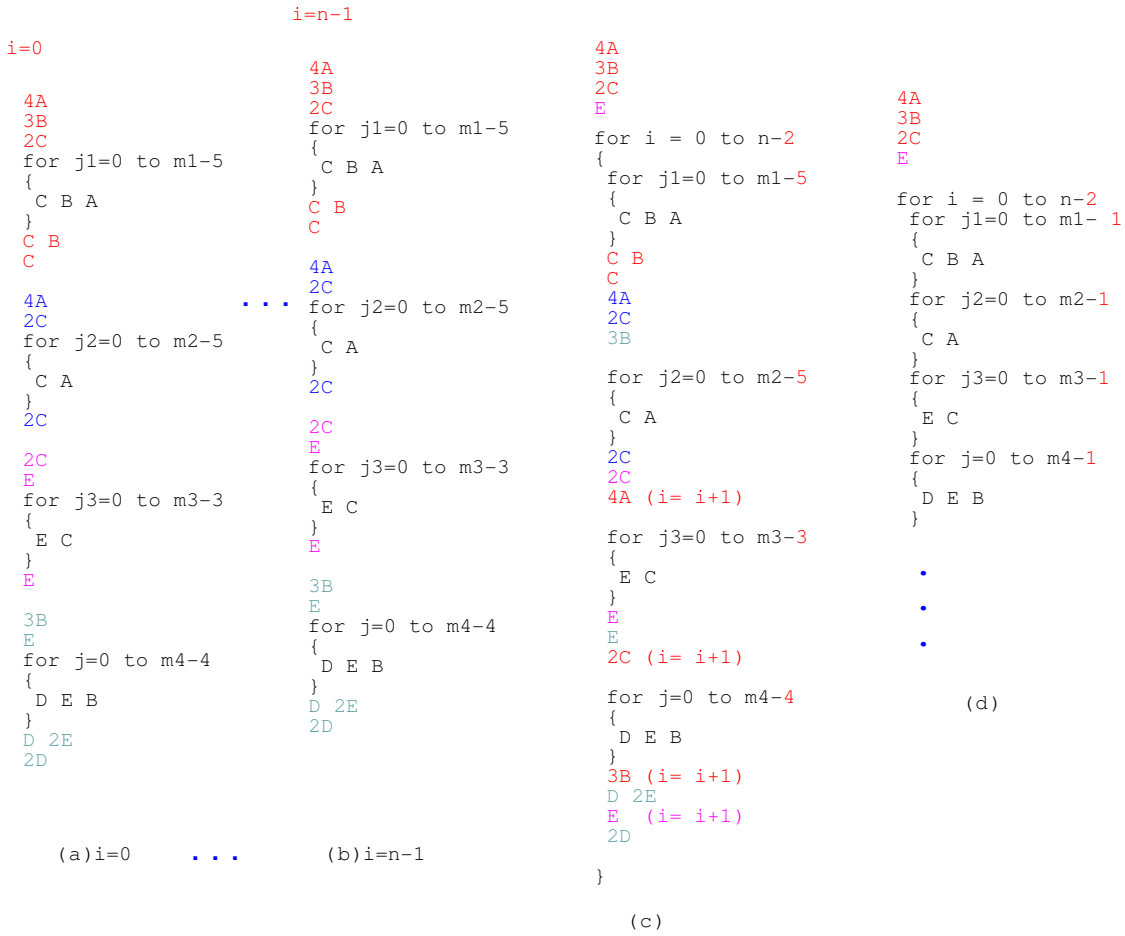


Fig. 7.10. Generated Code

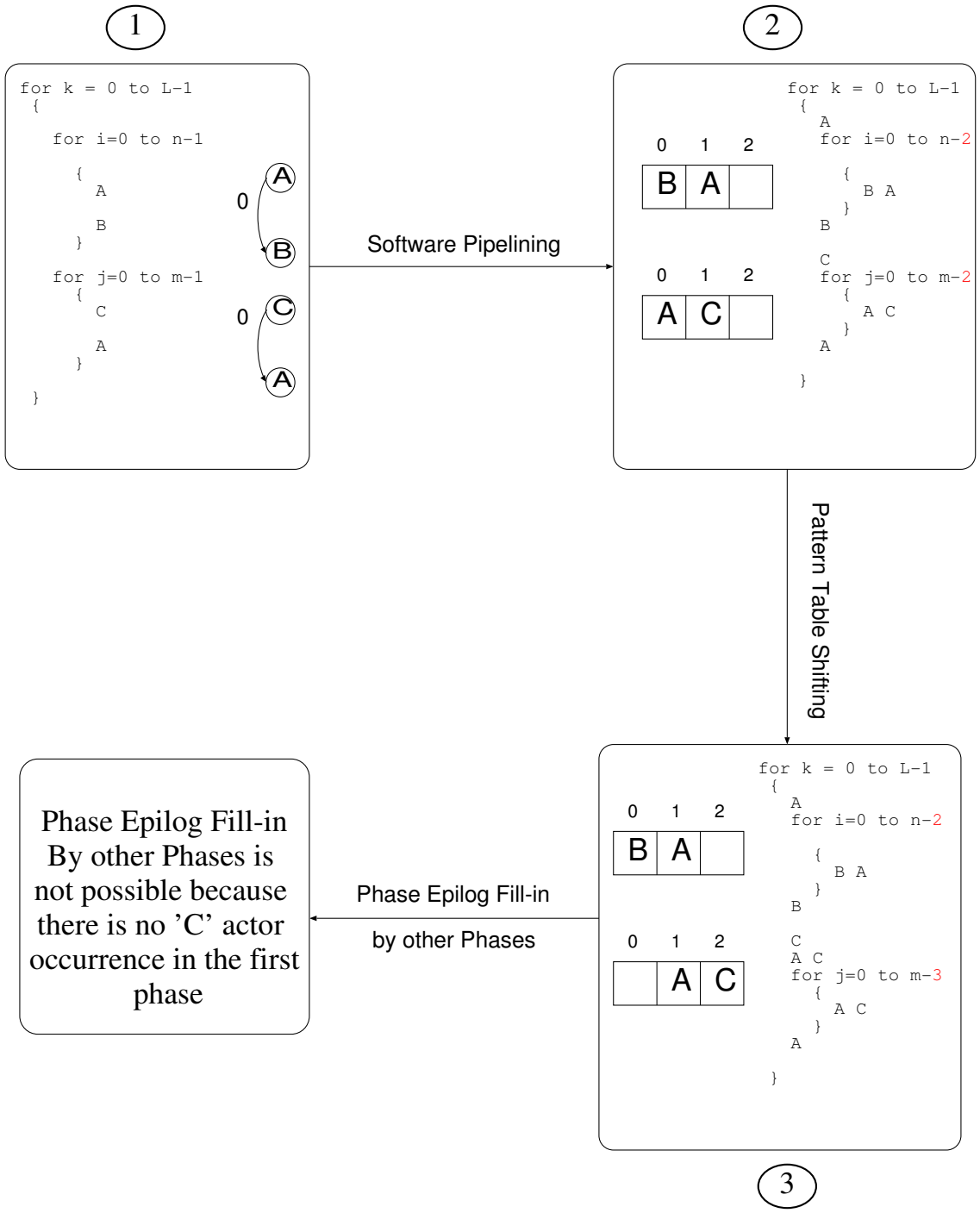


Fig. 7.11. Opposite Example

Because of all these disadvantages it is no more interesting to think about 'Epilog Filling in with other Phase Firings' shifting technique.

Chapter 8

Prolog Epilog Merging

8.1 Introduction

As we have seen in chapter 7 INMS solution proposed, 'Pattern Tables Shifting', is not efficient enough to schedule a streaming application or any other loop nest. It is applicable only for some simple cases. In this chapter we propose another solution that can skip 'Pattern Tables Shifting' limits, the 'Prolog Epilog Merging'. The conflict between pipelining and code size can often be a side-effect of separating the optimization of individual inner loops. 'Prolog-epilog Merging' shows how to pipeline phases (inner loops) without any overhead on the size of the global outer loop. We are going to talk about 'Prolog-epilog merging' as a technique for loop nest in general and not especially for SDF applications. So when we say nested loops or loop nest it means both SDF pre-scheduled steady state and loop nest. Inner loop and phase have the same meaning as well. We will see in the following sections the 'Prolog Epilog Merging' shifting technique solution and its improvement by 'Renaming'.

8.2 Problem Statement

We propose to modulo-schedule (software pipeline) [50] as many phases as possible, while merging the prolog of each outer iteration of a phase with the epilog of its previous outer iteration. Such prolog-epilog merging is enabled by an explicit inner loops retiming and an implicit outer loop retiming (or shifting) [33; 16], at the cost of a few additional constraints on modulo scheduling. It is then possible to reintegrate the merged code block within the pipelined kernel, restoring the loop to its original number of iterations. This operation is not always possible, and depends on the outer loop's dependence cycles. Indeed, after software-pipelining, prolog-epilog merging may affect phases that are in dependence with statements shifted by the software pipeline (along an inner loop). This makes our problem more difficult than in the perfectly nested case [53].

8.2.1 Running Example

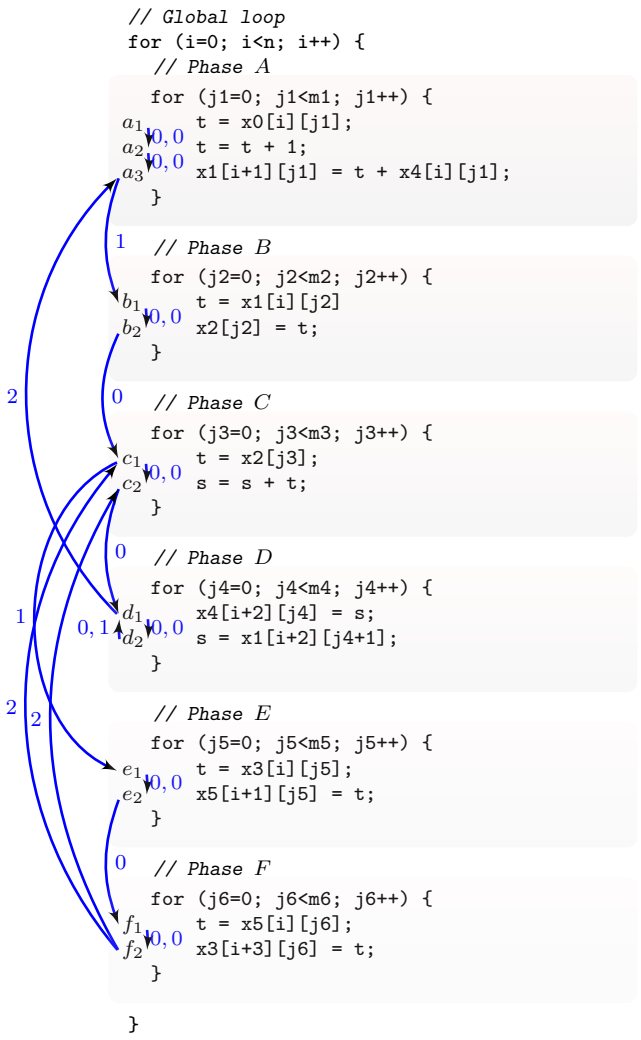


Fig. 8.1. Running example

Our running example is given in Figure 8.1. Statements and phases are labeled. Both intra-phase and inter-phase dependence vectors are shown. We consider that scalars are renamed automatically if they create an output dependence

The classical approach to the optimization of such an example is (1) to look for high-level loop fusions that may improve the locality in inner loops, often resulting in array contraction and scalar promotion opportunities [2; 63], and (2) to pipeline the phases (inner loops) whose trip count is high enough. We assume the first loop fusion step has been applied, and that further fusion is hampered by complex dependence patterns or mismatching loop trip counts. The result of the second step is sketched using statement labels in Figure 8.2; notice the *modified termination condition* in pipelined phases. As expected, this dramatically improves ILP, at the expense of code size increase. In addition, some ILP is lost in the prolog and epilog of each phase, and this results in accumulated overhead across the execution of the

global loop.

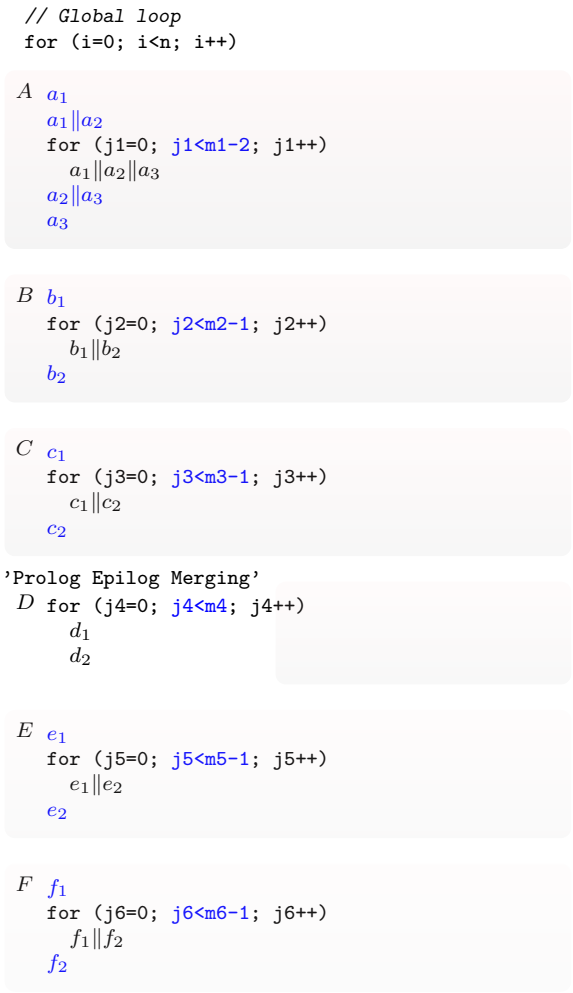


Fig. 8.2. Software pipelining all phases independently

The alternative is to shift the prolog of each pipelined phase, advancing it by one iteration of the global loop, then to merge it with the corresponding epilog of the previous iteration of the global loop. This is not always possible, and we will show in the following sections how to formalize the selection of phases subject to pipelining as a linear optimization problem.

Back to our running example, a possible solution is to pipeline and apply prolog-epilog merging to phases A, B, E and F. The code after advancing the prologs of pipelined phases is outlined in Figure 8.3; notice the outermost prolog — resulting from advancing the first global iteration of the phase prologs — and epilog — the last global iteration of phase epilogs. Yet this code is incorrect, for two reasons.

- The inter-phase dependence from statement e_2 to statement f_1 is violated, since f_1 in the prolog of phase F has been anticipated before one full iteration of phase E; some instances of this violation are depicted by a bold arc on Figure 8.3. To fix this violation, one may shift the whole phase E, advancing

it by one iteration of the global loop. This is possible since the only inter-phase dependence targeting phase E (statement e_1) has a non-null distance.

- A similar problem exists with the inter-phase dependence from statement b_2 to statement c_1; some instances of this violation are depicted by a bold dashed arc on Figure 8.3. Yet we will see that this violation cannot be fixed by shifting, due to the accumulation of shifting constraints on the cycle of inter-phase dependences involving A, B, C and D. We choose not to pipeline C in the following; we will later demonstrate the optimality of this choice after formalizing the global optimization problem.

The final code after pipelining all phases but C,[1] prolog-epilog merging, shifting E, and reintegrating the merged prologs and epilogs into the kernels is outlined in Figure 8.4; notice the *modified termination condition* on the global loop, and the *restored termination condition* on the pipelined phases (due to prolog-epilog merging).

The body of the global loop recovered its original size, and prolog/epilog overhead has disappeared. This major improvement was done at the minor expense of the loss of ILP on phase D, and some extra code outside the global loop, due to the global shifting of phase E.[2]

8.2.2 Inter-Phase Dependences

In the following, shifting is understood as *advancing* the execution of a statement by one or more iterations. For example, shifting b_1 implies that the first iteration of b_1 (or more) will end up in a prolog of phase B; this prolog will have to be merged with the epilog of this phase for the previous iteration of the outer loop. Since the dependence from a_3 to b_1 is carried by the outer loop, its associated distance (0) does not tell anything about the precise iterations of b_1 within phase B that are in dependence. Shifting b_1 along the inner loop — by any positive amount — is thus equivalent to shifting the whole phase B by 1 iteration of the outer loop. This observation is key to converting our prolog-epilog merging problem into a classical retiming one.

8.3 Characterization of Pipelinable Phases

From the global dependence graph G with multidimensional dependence vectors, the *phase dependence graph* G_p is defined as follows:

- nodes of G_p are the phases;
- an arc links a phase A to a phase B if and only if there is a path in G from a statement a of A to statement b of B; to avoid spurious transitively covered arcs, we also require this path to contain a single inter-phase arc;

[1] Attempting to pipeline C does not bring any ILP.

[2] The first iteration of the global loop executes E only, while the last iteration executes every phase but E.

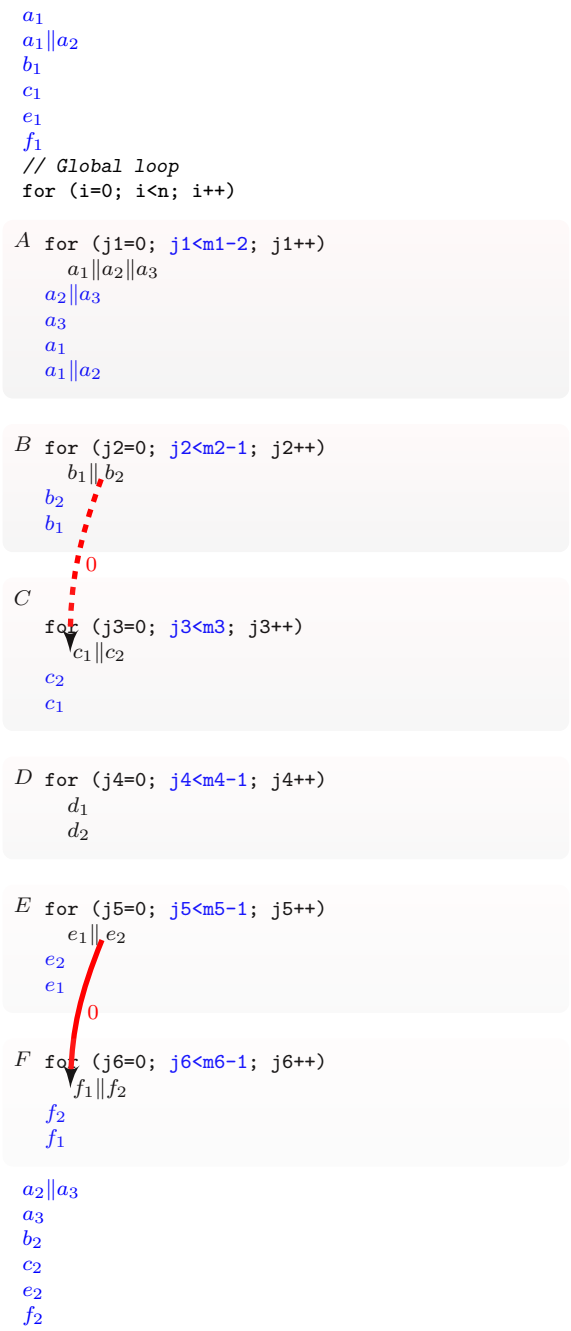


Fig. 8.3. Advancing prologs of pipelined phases (incorrect code)

- the distance associated with an arc of G_p is the sum of the distances, for the dimension of the global loop, along the corresponding path from a to b in G.

Arcs in G_p will be called *phase dependences*. They correspond to one inter-phase dependence and zero or more transitively-covered intra-phase dependence.

Notice the distance associated with a phase dependence takes into account non-zero distances along the outer dimension of intra-phase dependences.

Figure 8.5 shows the phase dependence graph for the running example.

```
// Prolog for shifted iteration of E
e1
for (j5=0; j5<m5; j5++)
  e1||e2

// Prologs of A, B, and F
a1
a1||a2
b1
f1

// Global loop
for (i=0; i<n-2; i++)
{
A  for (j1=0; j1<m1; j1++)
     a1||a2||a3

B  for (j2=0; j2<m2; j2++)
     b1||b2

C  for (j3=0; j3<m3; j3++)
     c1
     c2

D  for (j4=0; j4<m4; j4++)
     d1
     d2

E  for (j5=0; j5<m5; j5++)
     e1||e2

F  for (j6=0; j6<m6; j6++)
     f1||f2
}
// Epilog for shifted interation of E
e2
// Shifted iterations of A , B, C, D and F
for (j1=0; j1<m1; j1++)
  a1||a2||a3
for (j2=0; j2<m2; j2++)
  b1||b2
for (j3=0; j3<m3; j3++)
  c1
  c2
for (j4=0; j4<m4; j4++)
  d1
  d2
for (j6=0; j6<m6; j6++)
  f1||f2

// Epilogs of A, B, and F
a2||a3
a3
b2
f2
```

Fig. 8.4. Software pipelining with prolog-epilog merging

8.3.1 Causality Condition

Every time a phase is software pipelined, we just showed that merging its prolog and epilog is equivalent — when considering G_p — to shifting the whole phase by

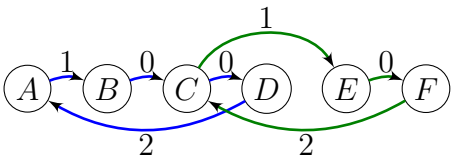


Fig. 8.5. Phase dependence graph

1, the interphase dependence distance will be decreased by one . To guarantee that all phases can be pipelined and their prolog and epilog merged, it is thus sufficient that every forward arc in G_p has distance $d > 0$, and any backward arc has distance $d > 1$.

This is of course too restrictive, and in general we are back to a traditional retiming problem [33]. Pipelining all phases is possible if and only if, for any cycle C,

$$\sum_{p \in C} d_p - nb\_backward\_edges(C) \geq nb\_phases(C). \tag{8.1}$$

We can state a more general result.

Let us define

$$k_C \stackrel{\text{def}}{=} \sum_{p \in C} d_p - nb\_backward\_edges(C). \tag{8.2}$$

Théorème 1 *For every cycle, the number of phases that can be safely pipelined is greater than or equal to k_C.*

This is only a lower bound, as we did not capture in G_p whether pipelining a phase did result in an intra-phase shifting of the specific statements involved in some inter-phase dependence.

Proof. Let us prove this result. For any instruction a, let $a(i, p, j)$ denote an instance of instruction a, given an iteration i of the global loop, a phase p and an iteration j of p. Let $t_{a(i,p,j)}$ denote the logical execution instance of $a(i, p, j)$ and $a(i, p)$ denote the set of instances of a at global loop iteration i.

$$b(i', p') \text{ depends on } a(i, p) \text{ with dependence distance } d \implies \forall j, j', t_{a(i,p,j)} < t_{b(i',p',j')} \text{ and } i \leq i'. \tag{8.3}$$

This means that if there is an interphase dependence between a of phase p and b of phase p' then all instances of a execute before b, $i \leq i'$.

Indeed, a phase dependence in G_p between p and p' corresponds to dependences between two sets of statement instances $a(i, p)$ and $b(i', p')$.

Software pipelining p' may imply shifting occurrences of instruction a. We call c_j the associated shifting distance along p', c_i the shifting distance along the global loop, and we consider two cases.

Forward edge. If p' depends on p with distance d and p' follows p in the loop nest, c_i must be chosen such that $d \geq 0$.

Backward edge. If p' depends on p with distance d and p' precedes p in the loop nest, c_i must be chosen such that $d > 0$.

We may compute c_i, taking into account the global loop shifts over outgoing arcs, the distance d, and whether p' is pipelined or not. The global loop shifts and d are the classical retiming variables and parameters. What happens to p' can be modeled easily, as we previously observed in Section 8.2.2 that shifting along an inner loop by any amount c_j can be compensated by shifting along the global loop by 1.

Therefore software pipelining p' will increase the total pressure over a cycle by at most 1. This constraint can be modeled by decrementing the distance d when p' is pipelined. We are back to a classical retiming problem, from which we deduce that p' can be pipelined if decrementing d does not induce any cycle with negative or null distance in G_p.

A simple recurrence on the number of pipelined phases concludes the proof.

8.3.2 Necessary and Sufficient Condition

In the absence of any information about the statements involved as sink and source of phase dependences, one may only assume that pipelining a phase will incur a shifting constraint along the global loop. In this case, the sufficient condition becomes a necessary one, and the previous proof can be extended to show that the number of phases that can be pipelined while merging prologs and epilogs is exactly k_C, as defined by (8.2).

Conversely, when considering the full dependence graph G, it is possible to constrain the pipelining of individual phases so that to forbid any inner loop shifting on some specific statements (targets of inter-phase dependences). This will allow to further pipeline some phases without impacting retimability of the global loop. We will come back to this extension when describing the complete algorithm.

8.4 Global Optimization Problem

Based on Theorem 1, we can formalize the software pipelining of multiple inner loops with prolog-epilog merging as a global optimization problem.

8.4.1 Multidimensional Knapsack Problem

First of all, the causality preservation condition in Theorem 1 needs to be extended to cover the whole phase dependence graph G_p. Indeed, software-pipelining k_C phases for each cycle C may create a retiming conflict, as a phase may belong to several cycles and can be chosen to be software-pipelined for one cycle and not for another.

The subject is not to software pipeline exactly k_C phases for each cycle C but to minimize the global outer loop execution time. Since for every cycle,k_C phases can be safely pipelined, we have to maximize an objective function under some constraints. The objective function associated with the (static) cycle count for the loop nest is the sum over all phases p of

$$profit_p = seqtime_p - m_p II_p,$$

where $seqtime_p$ is the number of cycles to execute phase p and II_p is the initiation interval for the pipelined version of phase p and m_p the number of iteration of phase p. Let $w_{Cp} \in \{0,1\}$ denote whether phase p belongs to cycle C. The optimization problem is the following:

$$\begin{cases} \text{variables:} & \forall p \in \{1,\ldots,nb\_phases\}, X_p \in \{0,1\} \\ \text{objective:} & \max \sum_{p=1}^{nb\_phases} profit_p X_p \\ \text{constraints:} & \forall C \in \{1,\ldots,nb\_cycles\}, \\ & \qquad \sum_{p=1}^{nb\_phases} w_{Cp} X_p \leq k_C \end{cases} \tag{8.4}$$

This is a multidimensional Knapsack problem, a well known NP-complete problem; unlike the one-dimensional case, there is no known pseudo-polynomial algorithm [43] but some heuristics give good results [48].

8.4.2 Algorithm

1. If for every cycle
$$k_C \geq nb\_phases$$
then software-pipeline each phase independently.

2. Otherwise:
   - solve the multidimensional knapsack problem to identify which are the k_C phases to pipeline;
   - retime the global outer loop, considering phase dependences in G_p, reducing their distance by one every-time the sink phase has been pipelined and contains intra-phase shifted statements at the sink of an inter-phase dependence; this step is guaranteed to terminate according to Theorem 1.

3. As an optional extension, pipeline all remaining phases with the additional constraint that statements at the sink of an inter-phase dependence may not be shifted; this may be easily modeled in any modulo scheduling algorithm by placing such statements initially in column 0 [50]. This step is guaranteed not incur global retiming constraints, because anticipating statements that are not the sink of a dependence carried by the global loop does not violate a schedule of the global loop.

4. Generate code, gathering all prologs and epilogs from pipelined phases, and iterating on them according to the retiming of the global outer loop.

8.5 Back to the Running Example

Figure 8.2 showed how to software pipeline all phases independently. This allows to compute the initiation interval II_p for every phase p. The profit of pipelining a phase is the difference in (static) execution cycles, between executing the original inner loop body and the pipelined version. Values given here are not measured nor real, just to explain the algrithm. Figure 8.6 shows the profit for all phases in the running example, assuming the trip counts (number of iterations) of all phases are identical and equal to $m = m_1 = \cdots = m_6$.

Phase	A	B	C	D	E	F
Profit	$2m$	m	m	0	m	m

Fig. 8.6. Profit table

The graph G_p was given in Figure 8.5. It consists of two cycles, $(ABCD)$ and (CEF). These cycles share phase C, which makes the optimization problem even more interesting as a naive approach may select C to be pipelined for one cycle but not for the other. Figure 8.7 shows k_C, the maximum number of phases that can be pipelined for each cycle.

Cycle	$ABCD$	CEF
k_C	2	2

Fig. 8.7. Cycle retiming constraints

Overall, we have to solve the following optimization problem:

$$\begin{cases} X_j \in 0, 1 \\ \max(2X_1 + X_2 + X_3 + X_5 + X_6) \\ X_1 + X_2 + X_3 + X_4 \leq 2 \\ X_3 + X_5 + X_6 \leq 2 \end{cases}$$

A greedy approximation of the solution orders phases from the most profitable phase to the less profitable one, and selects as many phases as possible for software pipelining, while respecting the k_C constraint for every cycle C. The result for the running example is to pipeline A, C, and E, with a total profit of $4m$.

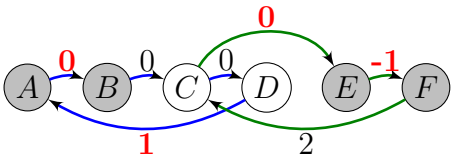


Fig. 8.8. Modified phase dependence graph after pipelining A, B, E and F

The multidimensional knapsack solution is better: phases A, B, E, F are pipelined, with a total profit of $5m$. Figure 8.8 shows the modified phase dependence graph, where pipelined phases are shaded, and the decremented distances of

incoming arcs appear in a bold face — following the retiming model of the proof of Theorem 1. Notice that phase C is more profitable than B, but pipelining B instead gives us a chance to choose another phase for the other cycle and increases the total profit. We suppose that the execution time of every instruction is 1. In the exmaple there are thirteen insturctions so the execution time of the sequential code is $13 \times m$ and after software pipeling of A, B, E and F phases we have a profit of $5 \times m$ wich corresponds to a speedup of $13/(13-8) = \mathbf{1.625}$.

As this example shows, it may be overall more effective to pipeline less profitable phases but maximize the profit on every cycle. This observation is very natural when the phases have a different trip count, but our running example shows that this may also occur when cycles in the phase dependence graph are not disjoint.

The resulting code, with prolog-epilog merging and generation of the global loop's prolog and epilog was shown in Figure 8.4.

This corresponds to a speedup of $13/(13-8) = \mathbf{1.625}$.

8.6 Dependence Removal

So far, we did not consider the applicability of data dependence removal techniques, like privatization and renaming [20; 37]. It is reasonable to assume scalar variables have been renamed through conversion to SSA form [14], as is the case in modern optimizing compilers; this guarantees the absence of output (write-after-write) and anti (write-after-read) dependences on *scalar* variables. The case of arrays requires significant static analysis and code generation effort (copy-in and copy-out), and a memory overhead [20; 37]. We are not worried by the overheads of copy-in and copy-out, assuming it is amortized over many iterations of the global loop. In addition, we will not consider privatization as the dependences we try to remove involve *distinct source and sink statements*, where array renaming applies. Nevertheless, since prolog-epilog merging is partly motivated by code size improvements, the memory costs of array renaming must be severely controlled. Our work is driven by embedded applications, and we assume a constant bound M on the total memory available for array renaming. Since our technique guarantees the size of the global loop (kernel) does not increase, it is easy to compute such bound, given the original code size and memory footprint of the global loop, for a given local memory configuration (local memory size and hierarchy).

Removing a dependence may suppress a cycle, hence yield more pipelinable phases, but it also consumes more memory. Overall, the solution is a compromise between the pipelining profit (indirecly linked with the number of pipelinable phases) and the need to keep the amount of extra memory below M. We can model this tradeoff as an extension to the previous linear optimization problem. When renaming a left-hand side (LHS) occurrence of an array, incoming anti-dependences and both incoming and outgoing output dependences to that statement are removed.

We model the decision of renaming an array a in *all* LHS occurrences of instructions of a phase p through a variable $R_{a,p} \in \{0, 1\}$. Such variables are multiplied to "big" constants, controlling which constraints should be nullified — depending on

which cycle is broken through array renaming. To capture the correlation between the decision to rename an array and the removal of an inter-phase dependence, it is important that the inter-phase dependence graph G_p is a *multi-graph*: each distinct inter-phase arc in G must yield a distinct arc in G_p. The complete optimization problem is stated in Fig. 8.9.

$$\begin{cases} \text{Variables:} & \forall p \in \{1,\ldots,nb\_phases\}, X_p \in \{0,1\} \\ & \forall a \in \{1,\ldots,nb\_arrays\}, \forall p \in \{1,\ldots,nb\_phases\}, R_{a,p} \in \{0,1\} \\ \text{Objective:} & \max \sum_{p=1}^{nb\_phases} profit_p \times X_p \\ \text{Constraints:} & \sum_{a=1}^{nb\_arrays} \sum_{p=1}^{nb\_phases} sizeof_a \times R_{a,p} \leq M \\ & \forall C \in \{1,\ldots,nb\_cycles\}, \sum_{p=1}^{nb\_phases} belongs_{p,C} \times X_p \leq k_C + nb\_phases \times \sum_{a=1}^{nb\_arrays} \sum_{p \in C \wedge assigned_{a,p}} R_{a,p} \end{cases} \tag{8.5}$$

Fig. 8.9. Optimizing the pipelining profit with array renaming

nb_arrays denotes the number of arrays, and $sizeof_a$ denotes the size of array a, i.e., the memory overhead of renaming one LHS occurrence, $assigned_{a,p}$ states that array a is assigned in p. The "big" constant is *nb_phases*: it is multiplied by the sum of all variables associated with renaming of some array a in some phase p belonging to a given cycle C. This constant is big enough that the constraint on a cycle C will be nullified *if and only if* one or more renaming occurs along the cycle.

8.6.1 Prolog-Epilog Merging with Renaming Algorithm

We may now outline the main steps of the algorithm, assuming a loop nest with multiple phases enclosed by a single global loop. In this section, we focus on solving our optimization problem without considering the impact on downstream loop nest generation methods.

1. If $k_C \geq nb\_phases$ for every cycle then software-pipeline each phase independently.
2. Otherwise:
   - solve the integer linear optimization problem to identify which are the k_C phases to pipeline;
   - retime the global outer loop, considering phase dependences in G_p, reducing their distance by one every-time the sink phase has been pipelined and contains intra-phase shifted statements at the sink of an inter-phase dependence; this step is guaranteed to terminate according to Theorem 1.
3. Pipeline all remaining phases with the additional constraint that any statement at the *sink* of an inter-phase dependence may not be shifted; in a modulo scheduling algorithm, this constraint can be modeled by forcing such statements to be assigned to column 0 [50]. This step is guaranteed not impact global retiming constraints.

4. Generate the kernel, prolog and epilog of the retimed *global* loop.

5. Generate code for the kernel, prolog and epilog of every pipelined *phase*.

6. Gather all prologs, hoist them *before* the global loop, *after* the prolog of the retimed global loop, and execute them in the same order as phases in the global loop.

7. Gather all epilogs, hoist them *after* the global loop, *before* the prolog of the retimed global loop, and execute them in the same order as phases in the global loop.

8.6.2 Code Generation

The previous algorithms yield multidimensional shifts resulting from phase pipelining and global loop retiming. However, unlike code generation for single-dimensional pipelining [52], we are not dealing with multiple repetitive patterns in the phase kernels and can rely on the classical code generation methods [50].

Code generation for the retimed global loop only involves classical loop peeling and induction variable substitutions of multidimensional loop shifting [15].

Code generation for pipelined phases after prolog-epilog merging is almost identical to the inner loop pipelining case, except for the following steps.

1. As the loop kernel is now collapsed with the merged prolog and epilog, the trip count of a pipelined phase is *not* decreased by the pipeline depth from the original trip count.

2. When the loop counter occurs in an expression of some shifted statement, one needs to generate an extra induction variable and schedule an extra integer addition in the kernel. In our case, if the statement is shifted by k iterations, the extra induction variable needs to *wrap-around* [24] before proceeding with the last k iterations of the phase. This requires an additional comparison and a conditional move (or a simple mask in case of power-of-two trip counts). These instructions are off the critical path and are not expected to have a high overhead, except on small loop bodies. We will come back to the evaluation of this overhead in the experimental section.

8.7 Related Work and Challenges

We do not aim at extending software pipelining to nested loops, unlike Muthukumar and Doshi [40], Rong et al. [53; 52] and most previous work on multidimensional pipelining (see e.g. Ramanujam et al. [49]). We simply leverage the enclosing loop nest to amortize the startup/flush overhead associated with software pipelining, and to control the code expansion in inner loops.

Compared to plain shifting of statement iterations, our technique involves a more complex combination of affine scheduling [22] and iteration domain splitting (a.k.a. index-set splitting) [23]. This raises many issues, some of which are discussed below.

8.7.1 Managing Register Pressure

There is an unfortunate side-effect of retiming a prolog (resp. epilog) along the global loop: any live variable entering (resp. leaving) the pipelined kernel will interfere with *every variable in other phases*. The effect on register pressure can be disastrous [60]. There are multiple ways to tackle this problem.

- The increased pressure is comparable to aggressive scheduling of unrolled or fused loops [38; 9]. This should be encouraging given the practical importance of loop fusion among loop optimizations for memory locality and ILP enhancement.
- Spills resulting from inter-phase liveness can always be spilled outside phases. This may turn out to be cheaper than executing the low-ILP prolog/epilog of a deeply pipelined inner loop. It is even more likely to be shorter, especially on architectures with instruction set support for register spill/refill: register stack engine on the Itanium [39], register windows on Sparc, or multi-push/multi-pop operations on CISC instruction sets.

8.7.2 Managing Code Size

Our method results in code growth outside the global loop only. This is nicer to memory-constrained architectures, but it may still increase cache pollution (or code-copying on local memories). Furthermore, code growth is amplified by the global loop retiming induced by prolog-epilog merging. For innermost loops, *prolog and epilog collapsing* is an alternative strategy consisting in guarding the phases with rotating predicate registers [18; 19]. This does not reduce pipeline startup/flush delays however. In our case, pipeline depth has negligible influence on startup time since prologs/epilogs are hoisted outside the global loop.

Muthukumar and Doshi extended the technique to multidimensional software pipelining [40]. They do not target code size reduction, but increased throughput on perfectly nested kernels with low-trip-count innermost loops. Iterations corresponding to prologs and epilogs are shifted over the entire execution of the innermost loop kernel, effectively overlapping iterations of an epilog with those of the next prolog. Compared to prolog-epilog merging, collapsing is difficult to generalize to imperfectly nested loops and incurs harder legality constraints. It is also limited to ISAs with rotating predicate registers. Experimental results on a prototype implementation inside Intel's production compiler are encouraging (despite register pressure challenges similar to ours); this motivates revisiting Muthukumar and Doshi's technique [40] in the context of prolog-epilog merging.

8.7.3 Multidimensional Scheduling

There are clear opportunities for integrating our technique with other forms of multi-level pipelining, or combined pipelining and unroll-and-jam [9; 53]. E.g., considering phase C of the running example, it is possible to improve ILP by shifting c_1 by one iteration of the *global* loop.

High-level loop optimizations are also promising application of prolog-epilog merging. The polyhedral model is an expressive way to define and search for complex sequences of loop transformations [25]. Yet such complex transformations often induce code size expansion. One source of code duplication comes from multidimensional shifts [62]. It seems possible to integrate our technique in the code-generation phass of a polyhedral compilation tool [25].

8.8 Conclusion

Prolog-epilog merging may appear as the most natural extension to inner loop pipelining and the right way to schedule an SDF application steady state. Indeed, it avoids the code size and startup time overhead of nested prologs and epilogs: these advantages over loop unrolling are exactly the motivations that drove to the design of software pipelining algorithms [32]. We formalized the concept of prolog-epilog merging, combining inner loop pipelining with multidimensional retiming. We combined our technique with array renaming to improve the pipelinability of inner loops. This results in a global scheduling and memory expansion tradeoff, modeled as a tractable, integer linear optimization problem.

Chapter 9

Code generation of Prolog-Epilog Merging

9.1 Introduction

As we have seen in the previous chapter, chapter 8, Prolog-Epilog Merging technique has proved efficient in pipelinable phases detection and prologs epilogs merging. Now phases to be pipelined are selected but how to generate their codes? How to generate code for prolog-epilog merging? Generating the code for this technique means to write each actor firing in the right place in the code and recompute its new iterators i' and j', moving an instruction or actor occurrence from one iteration to another implies a change in iteration and then modifications of iterators values.
This shifting looks like clock system. For example if time is $10:15:59$, one second after time will be $11:16:00$, so a shifting in time by one second has modified both minutes and seconds loops, the second coordinate 59 becomes 00 and the minute coordinate 15 becomes16. Prolog-epilog merging shifting technique does the same thing. The actor firing $A(1,0)$ $i=1$ and $j=0$, in Fig. 9.1, will execute in parallel with $B(0,m-1)$ after prolog epilog merging. In the generated code its coordinates are, $i'=0$ and $j'=m-1$, the shifting has transformed point $A(1,0)$ to $A(0,m-1)$, so a shifting by one of an actor may have effects on both loops, the inner and the global ones, 0 becomes $m-1$ and 1 becomes 0.
In the following sections we are going to talk about our code generation technique and its implementation.

9.2 Prolog-Epilog Merging Implementation Idea

INMS framework is composed of three processes:

- The first process:
  - if it is an SDF application then it takes the pre-scheduled steady state as input else if it is any loop nest so this nest itself will be the input

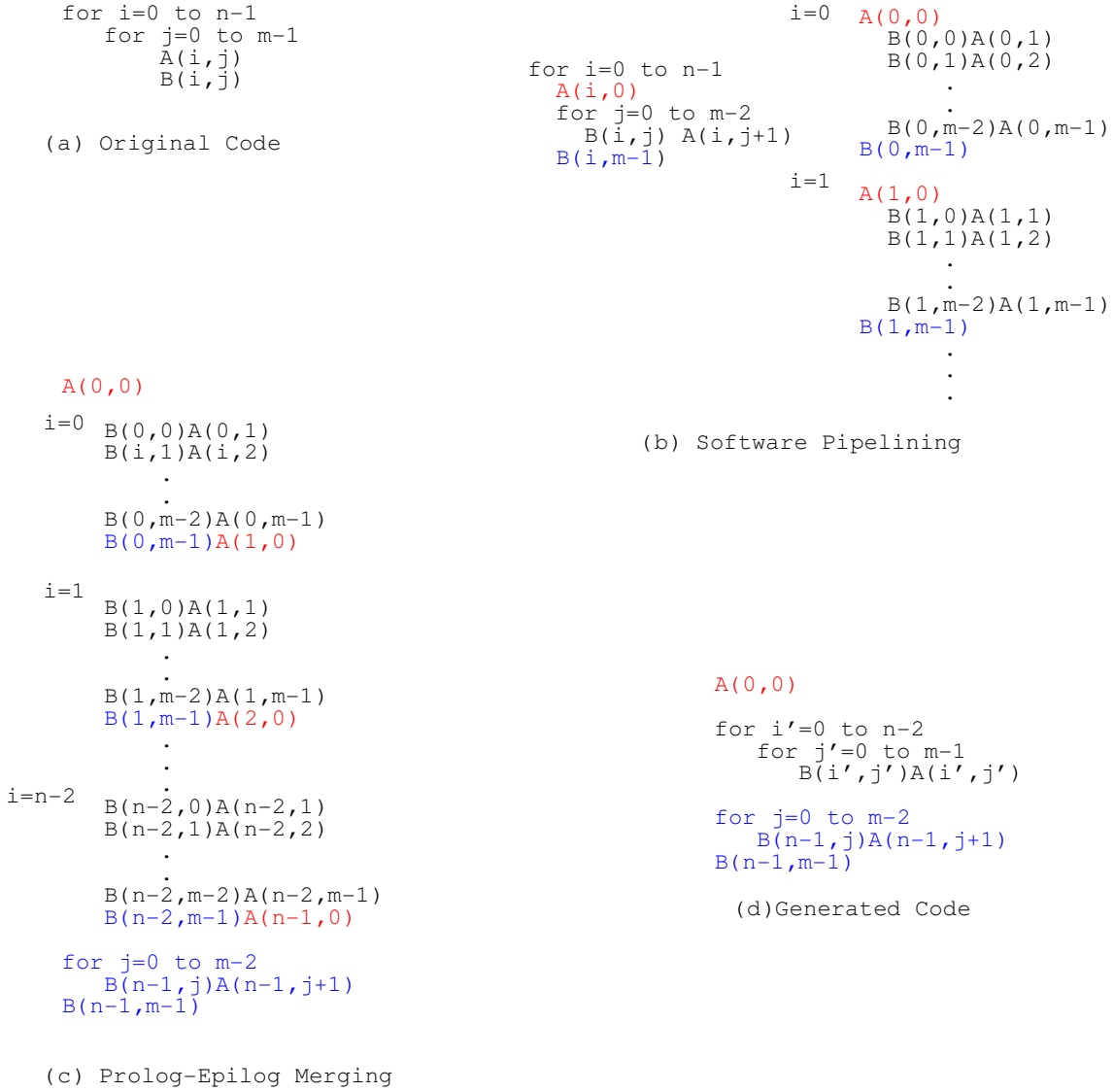


Fig. 9.1. Code Generation Example

– selects phases to be pipelined as output, by solving the multi-knapsack system. To select phases to be pipelined the process should be able to detect phases first. A phase is a loop and the loop structure is defined by the language itself, the beginning of each loop is a language key word. We will see in section 9.2.2.1 how to detect loops in general and loop nest in particular.

- The second process is software-pipelining the selected phases. In the output code, selected phases are pipelined and many prologs and epilogs are generated (inside the global loop) see the example in Fig. 9.1(b).

- The third and the last one, code generation process, takes the code, output

of second process, as input and merges the prolog of a phase lp, of the global loop iteration i, with the lp epilog of the previous global loop iteration $i-1$ to form the final code, a loop nest without any prologs nor epilogs inside it (see Fig. 9.1(c) and Fig. 9.1(d)).

In the previous chapter 8 we have presented the first process. The second one, software pipelining of selected phases, may be realized by implementing any software pipelining technique like Swing Modulo Scheduling (SMS) [26] or using any tool or compiler where software pipelining is implemented. The third process, 'Code Generation' is what we are going to see here. It has as input a pre-scheduled steady state (loop nest) with pipelined phases and has as output the same code with prolog epilog merged for each pipelined phase. It does this task by merging prolog and epilog of each pipelined phase and then checking the effect of this merging on the other phases. We call the first task **'Phase Prolog-Epilog Merging'** and the second one **'Effect of Phase Prolog-Epilog Merging on Loop Nest'**.

Remark: Because we focus on two kinds of applications at the same time, pre-scheduled steady state (special loop nest) and loop nest, although they are almost the same thing, so every time there is a different in treatment we specify it.

9.2.1 Phase Prolog-Epilog Merging

Focusing on each phase separately simplifies the problem and lets us understand clearly what happens when we merge a phase prolog and epilog. Consider that we have a loop nest with one phase inside only, see Fig 9.1 (a). Merging the prolog and the epilog of this phase means to move the prolog of the first iteration of the inner loop outside the global loop and to execute it first; see Fig 9.1(b). Then merging every prolog of iteration i with the epilog of iteration $i-1$, which forms some phase kernel iterations, and adding them to the phase kernel as in Fig 9.1(d). The upper bound of phase iterator will be increased by this number of iterations. The latest global loop iteration is incomplete. It is exactly the phase kernel code followed by the epilog, see Fig 9.1(d). But how to automate this process?

For each phase,to merge its prolog and epilog we:

1. take the phase prolog from the global loop and put it above, just before it

2. go to the end of the global loop and write again the kernel code, this copy of kernel is not enclosed by the global loop.

3. take the phase epilog from the global loop and put it below, just after the copy of the kernel.

4. compute bounds. Shifted prolog decreases the global loop upper bound and prolog-epilog merging increases the upper bound of the phase (inner loop) kernel, by the number of iterations formed by its prolog and epilog.

5. compute actor occurrence coordinates in this new iterators space. After this shifting and merging actor occurrence coordinates (i, lp, j, r) becomes (i', lp, j', r), lp is the phase containing this actor occurrence and r is its order in the loop if the loop contains more than one occurrence (see section 6.5.2); the r dimension doesn't change, it always equals one after SDF to DDG transformation and lp does not change as well because the shifting technique is "Prolog-epilog Merging". If an actor occurrence appears in the global loop iteration i' and phase iteration j' this doesn't mean that its coordinates are (i', j'). We see in Fig 9.1(c) that $B(0, lp, m-1)$, $i' = 0, j' = m-1$ executes in parallel with $A(1, lp, 0)$, $i = 1, j = 0$, this means they are in the same iteration although they do not have the same coordinates. We will see how to compute Actor coordinates (instruction indexes) in this new iterators space in section 9.2.2.4.

The Example in Fig 9.2 shows the prolog-epilog merging automation process. The idea is so simple and so efficient. It works with any technique that generates prologs and epilogs, not only software pipelining, because the idea has no relation with software pipelining technique, it tries only to merge the prolog and the epilog. The code in Fig 9.2(e) is better than the code in Fig 9.2(a), in terms of efficiency and code size, because the code size of the global loop in Fig 9.2(e) is less than the code size of the global loop in Fig 9.2(a) and in streaming applcations, the number of iteration of the global loop (n) is largely greater than the number of iteration of the inner loops (m). Therefore, more the global loop code size is smaller more the efficiency (execution time) is better.

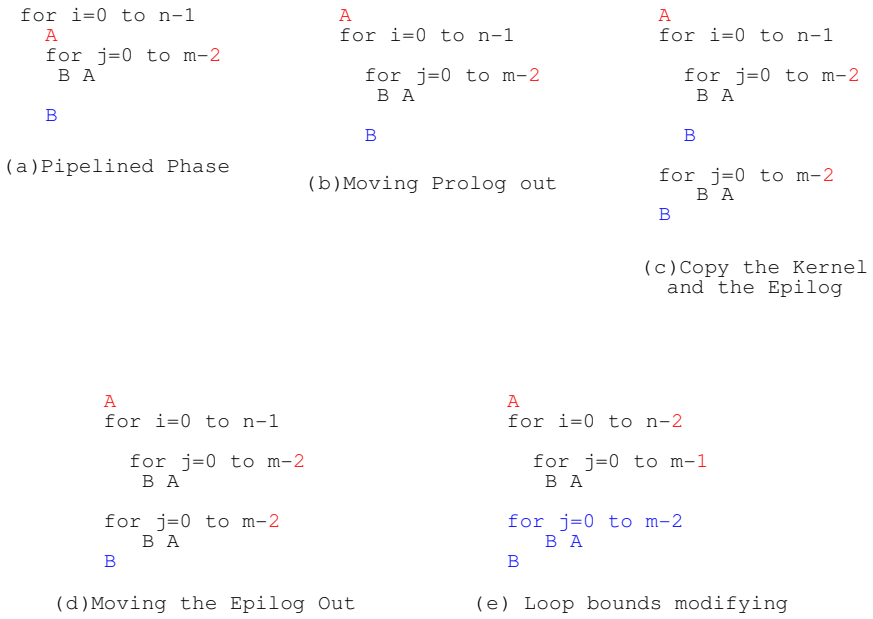


Fig. 9.2. Prolog Epilog Merging Automatizing

9.2.2 Phase Prolog-Epilog Merging implementation

The implementation of 'Phase Prolog-Epilog Merging' consists of:

- loop nest detection
- prolog epilog moving and kernel duplicating
- computing iterator bounds
- computing coordinates of actor occurrences

9.2.2.1 Loop Nest Detection

Because the INMS input may be a pre-scheduled steady state or any other loop nest, INMS is an SDF application scheduling technique and a loop nest optimization technique as well, two cases are possible:

- if it is an SDF application schedule, a pre-scheduled steady state, loop nest detection is not needed, because the SDF application schedule is represented by one steady state only.
- The code may be composed of many loop nests and each nest may contain many phases. Because INMS focuses on loop nests, each one separately, so we need first to detect them.
  The idea is very simple: the beginning of each nest is a loop key word, for example for a C language code it is 'for' or 'while'. We look at the code as a text and we try to find the first 'for' word, if we find it so this is the beginning of loop nest then the counter will be incremented if '{' is read and decremented if '}' is. If the counter value is zero so it is the end of the loops nest. The program terminates if counter equals zero. Because for every "{" we have "}" so the program terminates in all cases. The loop nest detection depends on the language itself, the language in which the code is written, because the loop key word and structure are defined by the language (in our implementation we have treated 'for' loop nest written in 'C').

9.2.2.2 Prolog Epilog Moving and Kernel Duplicating

The idea is: for every phase, we first detect its kernel, prolog and epilog and then move the prolog and epilog out of the global loop and copy the kernel code below the global loop as mentioned in section 9.2.1.

- kernel detection: we detect a phase kernel as we detect any phase (loop), see section 9.2.2.1.
- prolog epilog detection: because actor occurrences (instructions) in prolog and epilog appear in kernel code as well so for every kernel we:

  1. look for actor occurrences (instructions) that appear in the kernel and just above it. These actor occurrences form the prolog. Because we know the loop (phase) upper bound in both the original code and the pipelined phase, and the dependence graphs (DDGs) before and after

software pipeling so we can can easily find the exact number of each actor occurrence we are looking for, actor occurrences just before the kernel (the prolog) and actor occurrence just after it (the epilog)

2. look for actor occurrences (instructions) that appear in the kernel and just below it. These actor occurrences form the epilog.

3. move the prolog out to be executed before the global loop and write the kernel copy below, to be executed just after the global loop, then we move the epilog out the global loop as well but below it, the kernel copy plus the phase epilog form the true epilog.

Two important questions can be asked here:

- how to know if two actors are identical (two instructions) ?
  - an actor is generally a function so it has a name, two actors are identical if they have the same name
  - an instruction is:
    - INSTRUCTION: var = EXPRESSION
    - EXPRESSION: var | constant | op EXPRESSION | EXPRESSION op EXPRESSION | (EXPRESSION)

    '=' is the affectation operator and 'op' represents all arithmetic and logic operators. Two instructions are identical if their expressions are and two expression are identical if they consist of the same operators and the same operands, except iterators or indexes which may differ from one expression to another because of shifting. For example $a[i][j] = 5$, $a[i+1][0] = 5$ and $a[0][j-1] = 5$ are all identical but $a = 5$ and $a[i][j] = 5$ are not.
- 'prolog and epilog actors (instructions) appear in kernel as well' is true but how many occurrences of each actor do the prolog and epilog consist of, (instruction)? We are not going to move all actor occurrences (instructions) that appear in the kernel and up it out the global loop, the same thing for epilog, we need to move out only the right number of actor occurrences that form the prolog and epilog. INMS have already selected pipelinable phase and software pipelined them so we have some data, input, that tell us for every actor how many occurrences have been moved out of the inner loop (phase). By using these data we can move out of the global exactly the right number of occurrences, occurrences composing the prolog. The maximum number of an actor occurrences in prolog represents the number of iterations that should be added to the kernel upper bound. Because we have the kernel and the prolog we can easily built the epilog: prolog plus epilog equals iterations added to the kernel.

9.2.2.3 Computing Iterator Bounds

- For every pipelined phase, after applying Prolog-epilog merging, the kernel iterator upper bound is increased by the number of iterations formed by the merged prolog and epilog.
- For the global loop, the upper bound iterator value is modified too. Its new value equals the previous value minus one because in reality we shift the prolog actors and no more, the prolog contains less actor occurrences than the phase itself.

9.2.2.4 Computing of Actor Occurrence Coordinates

Untill now we have not talked about shifting effects on actor occurrence coordinates or instructions. Actor occurrence coordinates are in function of iterators i and j and shifting an actor occurrence means to move it from one phase iteration to another or even from one global loop iteration to anther. Before prolog-epilog merging, the actor occurrence coordinates are (i, j), we don't talk about lp and r because they do not change, so what are its coordinates after merging? It depends on where the actor occurrence will be, in prolog, kernel or epilog. We will see the epilog case in 9.2.3.3.

- for actor occurrences that are in prolog, we replace the phase iterator j by zero for the first occurrence and we increment it each time we find another actor occurrence. If the actor occurrence number is $m - 1$ (the inner loop upper bound) then we increment i and we make $j = 0$ again and we repeat the process untill all actor occurrences in prolog become assigned.
- for each kernel actor: if the actor appears in the prolog as well then this means that this actor was shifted by d occurrences, d being the number of the actor occurrences in prolog.
  - if $j \geq d$ then the actor occurrence coordinates are $j' = j - d$ and $i' = i$, shifting across inner loop (phase) iterations only.
  - else $j' = (m-1) - j$ and $i' = i - 1$, shifting across both inner loop (phase) iterations and global loop ones.

.

9.2.3 Effect of Phase Prolog-Epilog Merging on Loop Nest

Code generation suffers from a serious problem, the inter-phase dependence. We have defined this dependence as: the actor A of phase lp depends on the actor B of phase lp' means m occurrences of A depends on m' occurrences of B, m and m' are the number of iterations of phases lp and lp' respectively, one occurrence by iteration. We define interphase dependence relation in this way because dependence between two actors may change from one occurrence to another and we are not

going to define dependences between every two actor occurrences. The problem is that, because of this definition, if A depends on B and the dependence distance is 0 then, shifting one A occurrence implies shifting m' occurrences of B. Fig 9.3 shows an example. In Fig 9.3(a) we have a code consisting of two phases and C in the second phase depends on A of the first phase, the dependence distance is equal to 0. Because it is an interphase dependences, we can only say that m' occurrences of C depends on m occurrences of A. To be exact, we need to specify for every $C(i, j')$ all $A(i, j)$ occurences that $C(i, j')$ depends on them. However, if we do so, we will have a code size and DDG graph explosion. Therefore, we prefer only to say m' occurrences of C depends on m occurrences of A. In Fig 9.3(b) we see that only the second phase was pipelined, we suppose that the first phase can not be pipelined because of dependences between A and B. Now, applying prolog-epilog merging on the second phase makes C be moved out of the global loop and change the upper bound value of the second phase, $m' - 2$ becomes $m' - 1$ and of the global loop, $n - 1$ becomes $n - 2$. The latest incomplete iteration of the global loop will form its epilog, see Fig 9.3(c). C is moved out of the global loop (it is in the prolog code), C depends on A (an interphase dependence) and dependence distance is equal to 0 so m occurrences of A should be moved out the global loop and take their place just before C, see Fig 9.3(d). If the loop nest contains many phases the problem becomes more complex, so how to generate the code in this case?

9.2.3.1 Idea

Prolog-epilog Merging has been applied on all pipelined phases. To know if we need to shift occurrences of other actors or not, because of inter-phase dependence, the idea is:
for every actor A in prolog:

1. find k and r such that the number of A occurrences in prolog ($nb\_A\_occurrences\_in\_prolog$) is equal to $k \times m + r$, m is the number of iterations of phase lp, k is the result of the integer division of $nb\_A\_occurrences\_in\_prolog$ by m and r ($r = nb\_A\_occurrences\_in\_prolog \mod m$) is the rest of this division.

2. For every actor B that A depends on it we have:

   - d_{AB}: the vector distance of the interphase dependence of A on B, it has one dimension i because an interphase dependence is between two actors of two different phases.
   - $d_{AB} \neq 0$ means that the A occurrences in phase lp, from global loop iterations 0 to $d_{AB} - 1$, don't depend on B.
   - $nb\_B\_occurrences\_in\_prolog = k' \times m' + r'$: is the number of B occurrences already in prolog.
   - if $(k = (k' + d_{BA}))$
     - the first $d_{BA} \times m$ occurrences of A don't depend on B and $k' \times m$ occurrences of A depend on $k' \times m'$ and we have already this number of B occurrences in prolog.

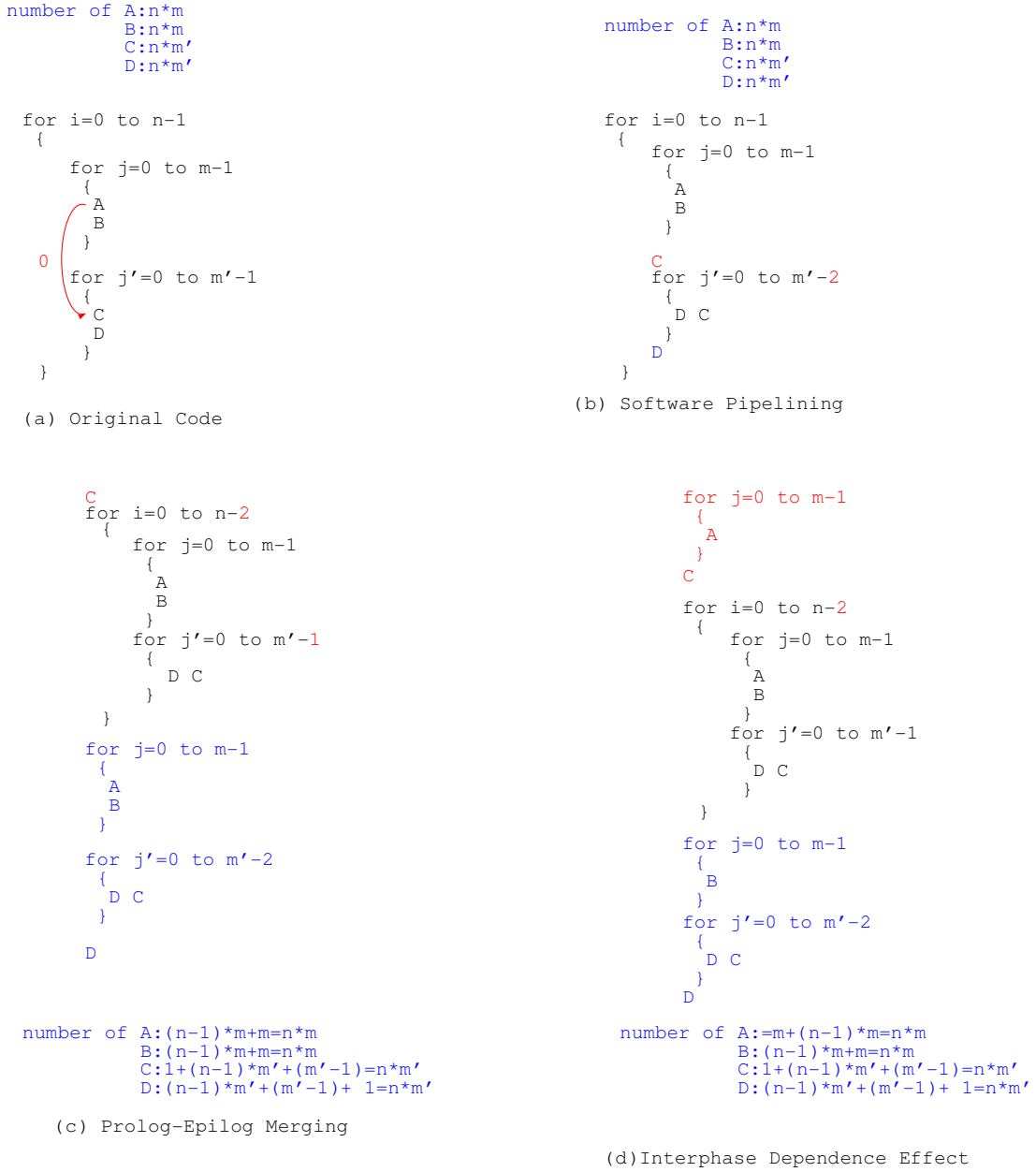


Fig. 9.3. Effect of Phase Prolog-epilog Merging on Loop Nest

  - if $(r > 0)$
    - For the remaining r occurrences of A, $r < m$ because it is the rest of the integer division of $nb\_A\_occurrences\_in\_prolog$ by k, m' occurrences of B should be moved to the prolog.
    - $k' = k' + 1$
  - $d_{BA} = 0$
  - see the example in Fig 9.5
- if $(k > (d_{BA} + k'))$ and $(r = 0)$:
  - from the first clause, $k > (d_{BA} + k')$, we can conclude that k can be

written: $k = k' + d_{BA} + c$ and from the second clause, $r = 0$, we can say that there are exactly $k \times m$ occurrences in the prolog.
  - the first $d_{BA} \times m$ occurrences of A don't depend on B and the $k' \times m$ occurrences of A depend on $k' \times m'$ occurrences of B which are already in prolog. The remaining $c \times m$ occurrences of A depend on $c \times m'$ occurrences of B, $c = k - (k' + d_{BA})$. So $(k - (k' + d_{BA})) \times m$ occurrences of B should be moved to the prolog.
  - $k' = k' + c = k' + (k - (k' + d_{BA}) = k - d_{BA}$
  - $d_{BA} = 0$
  - see the example in Fig 9.6
- if $(k > (k' + d_{BA}))$, $k = k' + d_{BA} + c$, and $r > 0$:
  - for $k \times m$ occurrences of A, $(k - (k' + d_{BA})) \times m$ occurrences of B should be moved to the prolog.
  - For the remaining r occurrences of A, $r < m$ because it is the rest of the integer division of *nb_A_occurrences_in_prolog* by k, m' occurrences of B should be moved to the prolog. So $((k + 1) - (k' + d_{BA})) \times m'$ B occurrences of actor B should be moved to the prolog
  - $k' = k' + c = k + 1 - d_{BA}$
  - $d_{BA} = 0$
- if $(k < (k' + d_{BA}))$ then no B shifting is necessary but d_{BA} may change
  - if $(k > k')$ then $d_{BA} = d_{BA} - (k - k')$
  - else if $(k = k')$ and $(r > 0)$ $d_{BA} = d_{BA} - 1$
  - else d_{BA} doesn't change
  - see the example in Fig 9.7
- in prolog, moved B occurrences take place just before A occurrences.

9.2.3.2 Algorithm

- The algorithm simply constructs a set of actors, actors that appear in prolog then it takes one actor,A, each time from this set, removes it and finds actors,B, that A depends on them, $depends[B][A] \neq -1$. For every B it compares k with the sum of dependence distance depends[B][A] and k', k' is the result of the integer division of the number of B occurrences in prolog by m' as explained before. If $(k > (depends[B][A] + K'))$ then the algorithm moves a number of B occurrences to the prolog according to the r value (r being the remainder of the integer devision of *nb_A_occurrences_in_prolog* by m) and adds B to *actors_in_prolog_set* if B does not belong to it. It repeats this process untill the set becomes empty.

declarations

```
A, B: actor, actors may be represented by numbers

depends[ ][ ]: is the dependence distance matrix, /* if A depends on
B then (depends[B][A] >= 0) else depends[B][A]=-1 */

m: is the number of iterations of phase lp, the A phase
k: k = nb_A_occurrences_in_prolog div m
r: r = nb_A_occurrences_in_prolog mod m
m': is the number of iterations of phase lp', the B  phase
k': nb_B_occurrences_in_prolog div m'
max: integer
global_lp_brn_sup: is the upper bound of the global loop iterator

n: is the number of global loop iteration in the original code,
 before any prolog-epilog merging

actors_in_prolog_set: this set contains all the actors that are in prolog
```

support functions

```
/* choose(): chooses or selects one actor from actors_in_prolog_set.
No ordering process is needed, we can choose any actor we want.*/
 actor choose(actors_in_prolog_set);

/*movetoprolog(): puts q occurrences of A in prolog just after the A
 occurrences, in the prolog, if there are some, else it writes them
before actors that depends on A and after actors that A depends on them.*/
 movetoprolog(A,q);

/*belongto(): checks if an actor belongs to the actor set*/
 boolean belongto(actors_in_prolog_set, B);

/*fill_in(): fills in actors_in_prolog_set by actors in prolog*/
fill_in(actors_in_prolog_set);

/*max_shifted():looks at actors in prolog and find the one that is more
 shifted than the other, it is the actor that have the greater k value
and returns k if r=0 and k+1 if r !=0.*/
 integer max_shifted(actors_in_prolog_set);
```

Algorithm

```
fill_in(actors_in_prolog_set);
while (actors_in_prolog_set != NULL)
 {
```

```
   A = choose(actors_in_prolog_set);
   actors_in_prolog_set = actors_in_prolog_set - A;
   for (B = 0; B < nb_actors; B++)
    {
      if ( (k > (k'+ depends[B][A])) && (depends[B][A]!= -1))
       {
        if(r == 0)
          {
            movetoprolog(B, ((k - (k'+ depends[B][A])) * m'));
            k'=  k  - depends[B][A];
          }
        else
          {
           movetoprolog(B, (( (k+1) - (k'+ depends[B][A]) ) * m'));
           k' =  k + 1  - depends[B][A];
          }
        depends[B][A] = 0;
        if (belongto( actors_in_prolog_set, B) == false)
          actors_in_prolog_set =  actors_in_prolog_set + B;
       }
      else if ( (k  == (k'+ depends[B][A])) && (depends[B][A]!= -1))
           {
             if (r>0)
               {
                movetoprolog(B, (( (k+1) - (k'+ depends[B][A]) )* m'));
                k' = k' +1;
                if (belongto( actors_in_prolog_set, B) == false)
                  actors_in_prolog_set =  actors_in_prolog_set + B;
               }
             depends[B][A] = 0;
           }
      else
          if (k > k') depends[B][A] = depends[B][A] - (k -k');
          else if ((k = k') && (r > 0))  depends[B][A] = depends[B][A] - 1
    }
 }
fill_in(actors_in_prolog_set);
max = max_shifted(actors_in_prolog_set);
global_lp_brn_sup = n- max;
```

We fill in again *actors_in_prolog_set*, at the end of the algorithm, to compute the value of max, *max* is used to computed the new value of the global loop upper bound.

- **Execution Example:** in Fig 9.4, after phase prolog-epilog merging was applied, in 9.4(c), Fig. 9.4(d) shows the execution of 'Effect on other Phases Algorithm'. m' in the running example is m, B is A and A is C. At the beginning, $actors\_in\_prolog\_set = \{C\}$. C is removed from the set. The

algorithm finds that C depends on A and $depend[A][C] = 0$. Because $k = 0$, $k' = 0$, $depend[A][C] = 0$ and $r = 1$ so $((k+1)-(k'+depends[B][A]))\times m' = m'$ so m' occurrences of A should be moved to the prolog. It is exactly what happens Fig. 9.4(d) shows that A has been added to the set and it will be removed after. Because it has no ascendant, the set becomes empty and the algorithm stops it execution.

- **Termination:** The algorithm terminates and stop its execution if *actors_in_prolog_set* is empty. To reach this execution state, the algorithm should removes actors from the set without adding new ones.

  - This is the case if the inter-phase dependence graph contains no cycle. If there is no cycle, roots in graphs do not depend on any actors and because of dependence paths between roots and other actors the algorithm arrives to the execution point where one root or more are in the actors_in_prolog_set set. Every time we remove an actor A, we add its ascendants to the *actors_in_prolog_set* if they don't belong to it, removing these roots from *actors_in_prolog_set* lets it be empty at the end so the algorithm terminates.

  - If the inter-phase dependences graph contains cycles, prolog-epilog merging assures that we are not going to shift phases infinitely. At most k_c phases can be shifted (section 8.3.1). At most k_C phases can be shifted assures that we will arrive to a shifting that has no effect on other phases, this means $depends[B][A]) > k$. So all cycles shifting will arrive to this execution point, with an actor B where $depends[B][A]) > k$. A will be removed form the set and no actor B will be added to so at the end *actors_in_prolog_set* will be empty and our algorithm terminates.

- **Correctness:** in our case the algorithm is correct means there is no case where an actor should be moved to the prolog and the algorithm doesn't move it or a case where an actors should not be moved to the prolog and the algorithm moves it.

  - Suppose there is a case where an actor B should be moved to the prolog and the algorithm doesn't move it. An actor B should be moved to the prolog means that there are some occurrences of A in prolog that depend on some B occurrences that are in the kernel, which means first that if there are already some occurrences of B in the prolog then they are not enough, let's call this number $k' \times m' + r'$. m occurrences of A depend on m' occurrences of B and $k' \times m' + r'$ occurrences of B are not enough so $k' < K$. If the dependence distance $depends[B][A] > k$ then no shifting is needed. It is not the case, moving some B actor is necessary as hypothesis, so $depends[B][A] < k$ and $k' < K$. Two cases are possible: $depends[B][A] + k' > k$ and $depends[B][A] + k' \leq k$. If $depends[B][A] + k' > k$, no shifting is needed. The actor should be

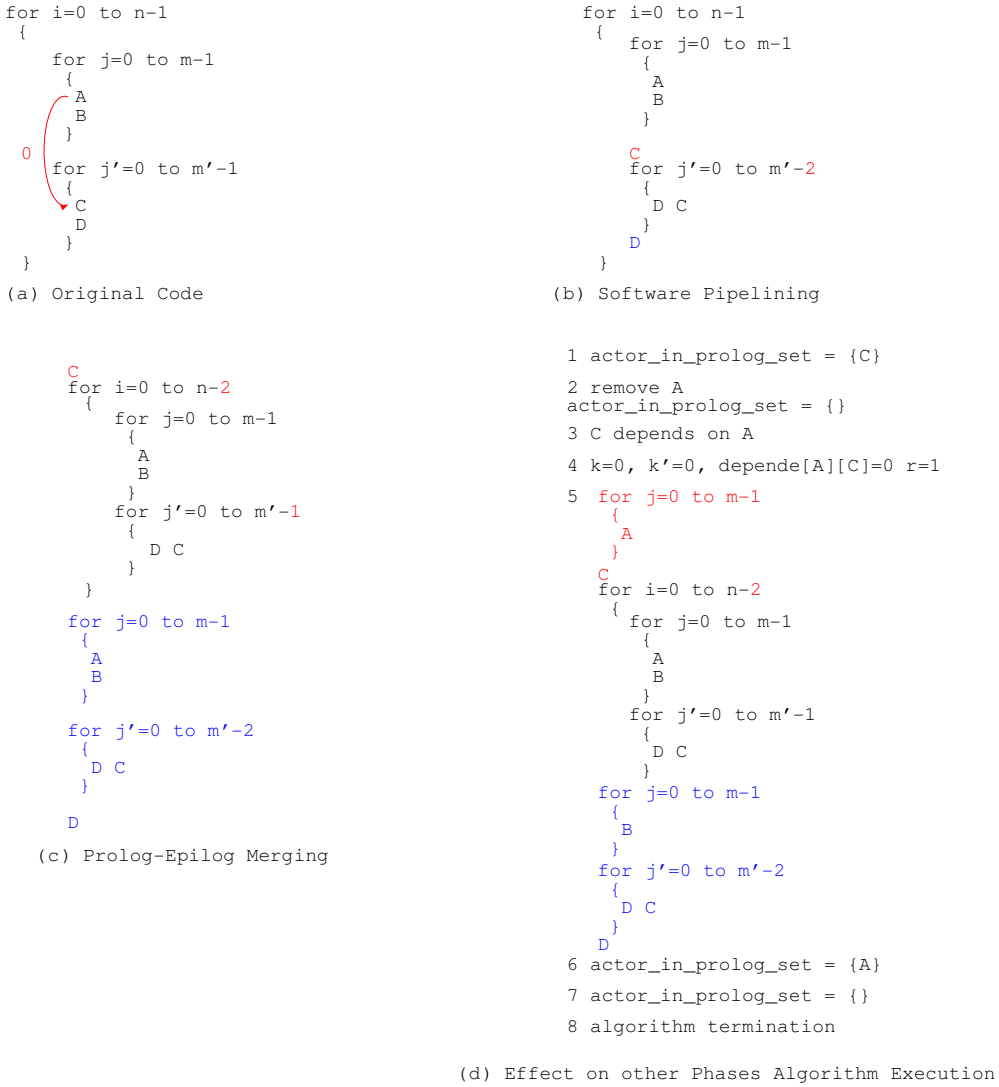


Fig. 9.4. "Effect on other Phases Algorithm" Execution Example

moved to the prolog as hypothesis so $depends[B][A] + k' \leq k$. If it is the case, our algorithm will move $(k - (k' + depends[B][A])) \times m'$ occurrences

of the actor B to the prolog. So the algorithm is correct.

- Suppose the second case, an actor should not be moved to the prolog and the algorithm has moved it. If it is moved to the prolog, then the test if $((k > (k' + depends[B][A])) \;\&\&\; (depends[B][A]! = -1))$ is true or $((k == (k' + depends[B][A])) \;\&\&\; (depends[B][A]! = -1))$ and $r > 0$, as they are the only cases where the algorithm can move actors.
  - P: If $(k > (k' + depends[B][A]))\&\&(depends[B][A]! = -1)$ equals true, then k is superior than $k' + depends[B][A]$, $k \times m$ occurrences in prolog needs $(k - depends[B][A])m'$ to be in prolog.
  - Q: If $((k == (k' + depends[B][A])) \;\&\&\; (depends[B][A]! = -1))$ and $r > 0$, then m' occurrences of B should be moved to the prolog.
  - If the actor B should not be moved then P and Q are false. Then either $(k < k' - depends[B][A])$ or $((k = k' + depends[B][A])\&\&(r = 0))$ and in these cases the algorithm doesn't move actor B so the algorithm is correct.

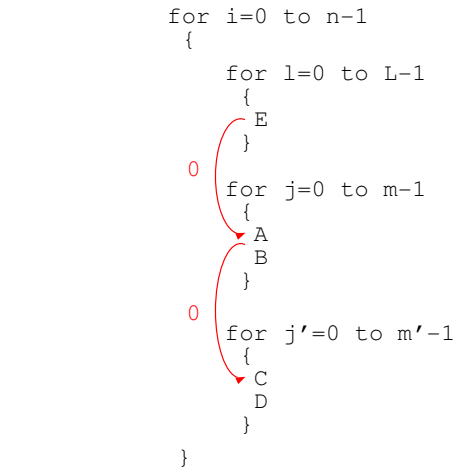
  In both case the algorithm results are correct so the correctness condition is verified.

9.2.3.3 Epilog Construction

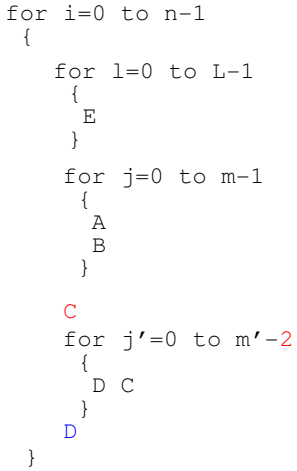
To construct the epilog we check, for each actor A in the global loop, if it is in prolog too or not, we check the k and r values:

- if $(k == 0)$ and $(r == 0)$ so there is no A occurrence in prolog. Then A will appear in all epilog iterations, from n-max to n-1.
- if $(k == 0)$ and $(r! == 0)$ so there is r occurrences of A in prolog. Then A will appear in its phase from global loop iteration n-max to n-2 and r occurrences of A in the latest gloop iteration, $i = n - 1$.
- if $(k == max)$ and $(r == 0)$ so A appears max times in prolog and $n - max$ times in the global loop, which means all its occurrences are in prolog and global loop, $n \times m$ occurrences, so there is no A occurrence in epilog.
- if $(0 < k \leq max)$ and $(r! = 0)$ then A will appear in its phase from global loop iteration $n - k$ to $n - 2$ and r occurences of A in the latest gloop iteration, $i = n - 1$.
- if $(0 < k < max)$ if $(r == 0)$ then A will appear in its phase from global loop iteration $n - k$ to $n - 1$

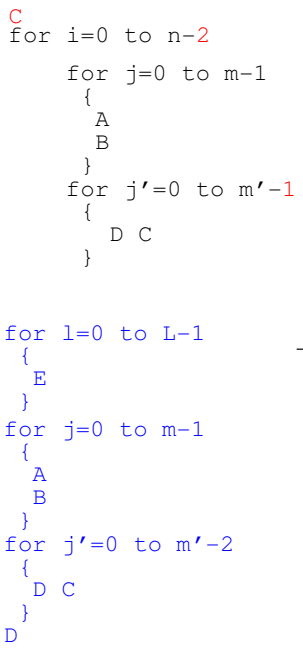
. In this way we can construct the epilog, by respecting execution order condition, the first phase then the second and so on, as in the original code.



(a) Original Code



(b) Software Pipelining of the Third Phase



(c) Prolog-Epilog Merging

- nb_occurrence_in_prolog(C) = 1
- k= 0, r = 1

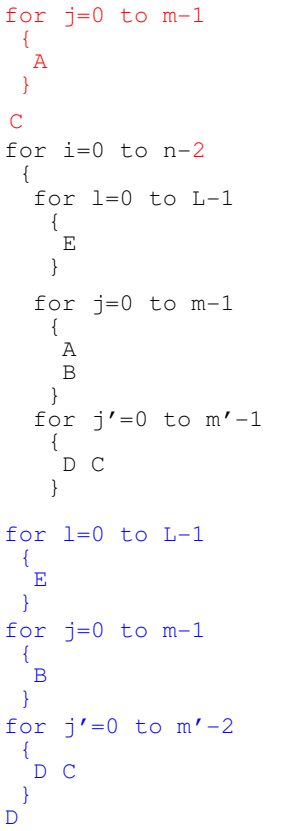
- nb_occurrence_in_prolog(A) = 0
- k'= 0, r' = 0

- C depends on A, dAC = 0

- k = k' + dAC

- r different than 0

-m occurrence of A should be moved to the prolog



(d)Effect of C Shifting on A

Fig. 9.5. Effect of Prolog-epilog Merging, 1^{st} case

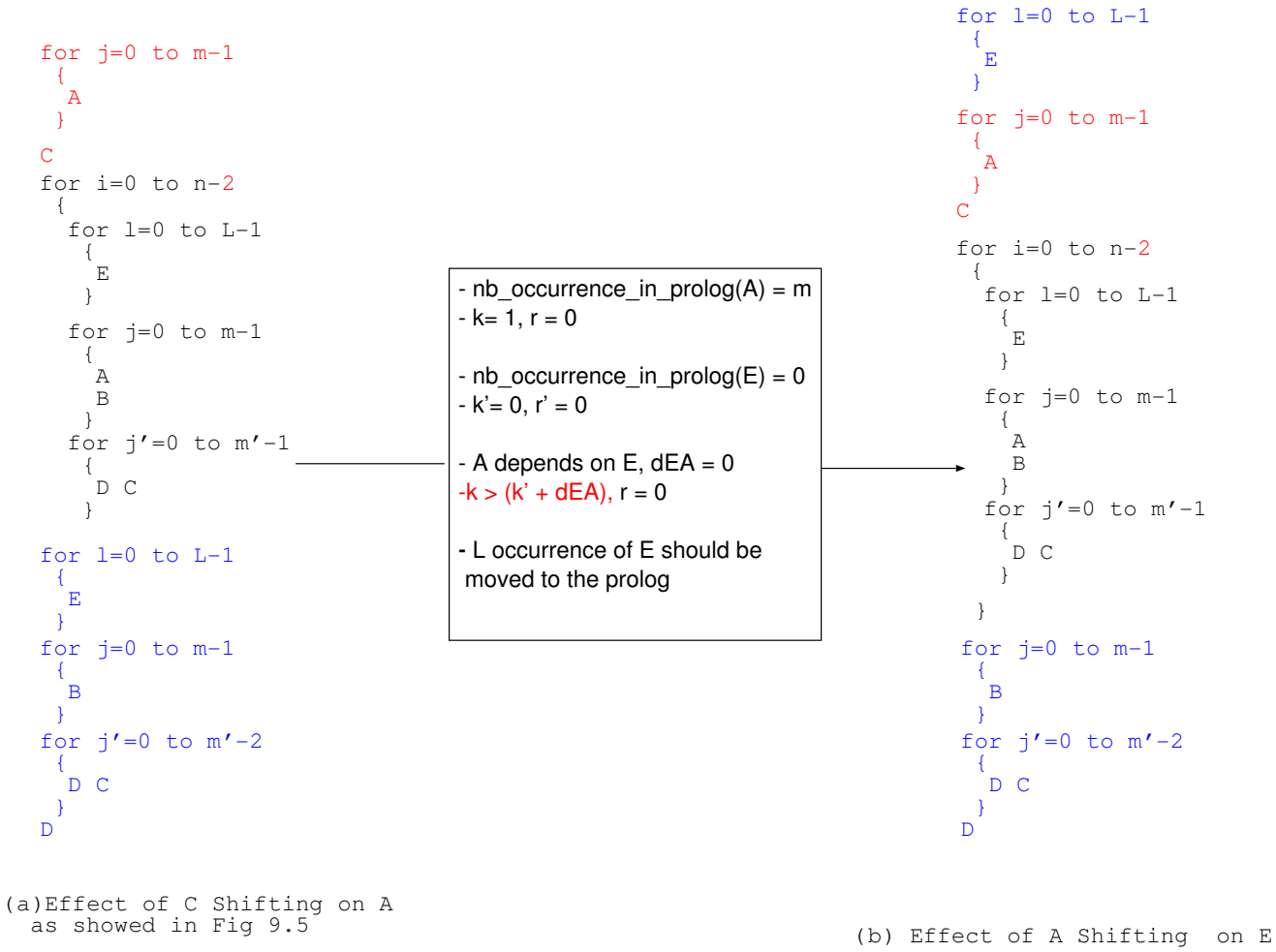


Fig. 9.6. Effect of Prolog-epilog Merging, 2^{nd} case

9.2.3.4 Recomputing of Actor Occurrence Coordinates

Moving an actor to the prolog because of an inter-phase dependence is different from phase prolog-epilog merging. Moving an actor B because of its dependence with another actor A that is in the prolog means we are going to move m' occurrences of B to the prolog. In this case B coordinates (i', j') in global loop are $(i - c, j)$. the algorithm moves $c \times m'$ to the prolog So $c = (k - (k' + depends[B][A]))$ or $c = ((k + 1) - (k' + depends[B][A]))$. Coordinate j doesn't change because the shifting is a multiple of m'.

9.3 Technique Extension

the following propositions are not really studied but it is how we see the future of our technique to be generalized.

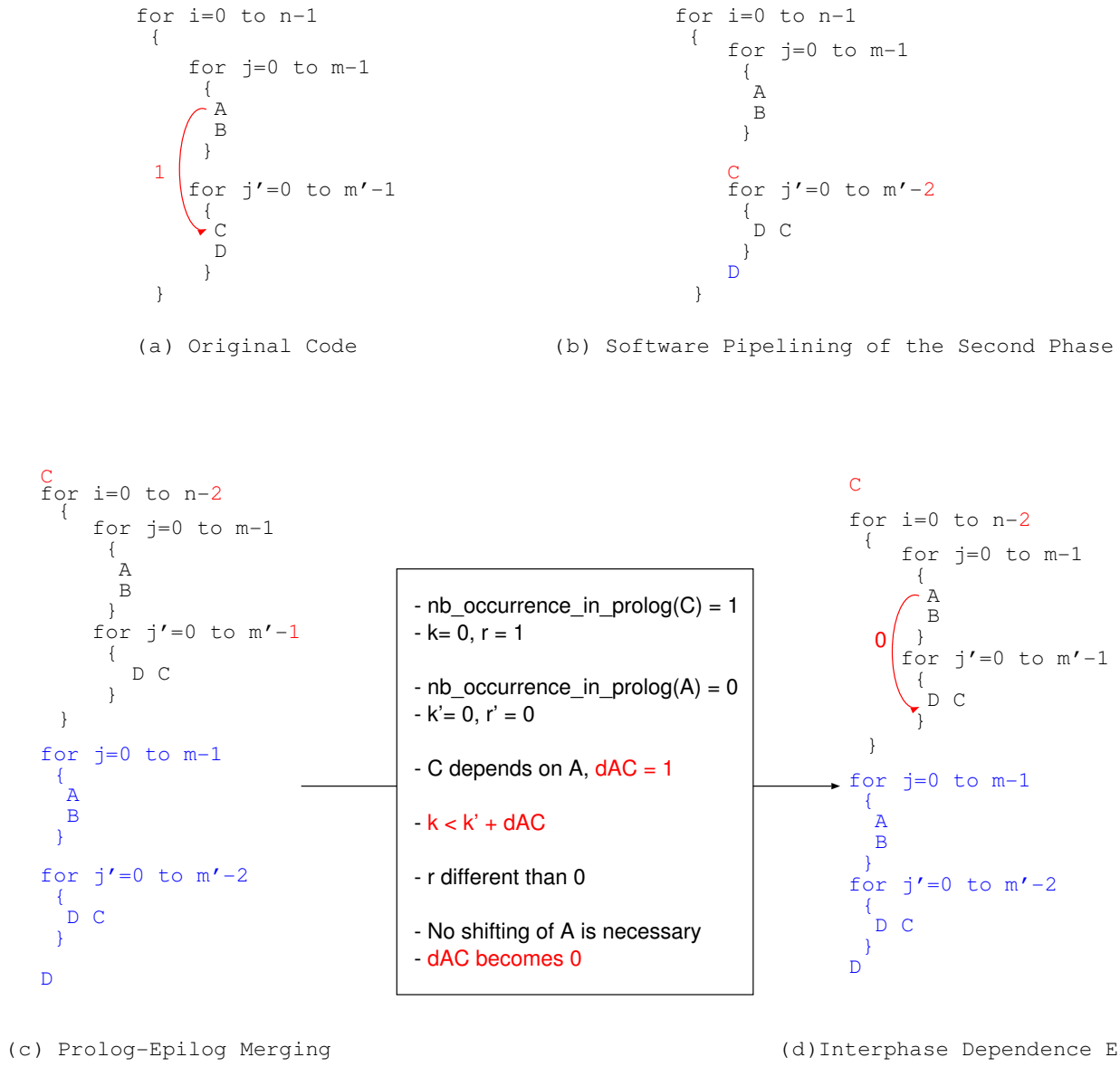


Fig. 9.7. Effect of Prolog-epilog Merging, 4^{th} case

9.3.1 Parametric Depth Prolog-epilog Merging

From the beginning, 6.3, we always suppose that the loop nest consists of two levels, what if the number of levels is more than two? One solution is to specify which level l on which INMS is going to be applied (Prolog-epilog Merging). During this thesis we always apply software pipelining and prolog-epilog merging on the inner phases but selecting on which level we apply them makes the technique more general. We call this generalization 'Parametric Depth Prolog-epilog Merging'.

9.3.2 Combining INMS with Other Optimization Techniques

Another important point is to combine INMS and 'Prolog-epilog Merging' with other loop optimization techniques like loop fusion, loop skewing and loop interchange. Which optimization techniques can be applied before and after INMS? We think INMS may be applied with other techniques that generate prologs and epilogs as well, not only software pipelining.

9.4 Conclusion

Prolog-epilog merging implementation, 'Code Generation', with its both algorithms **Phase Prolog_Epilog Merging** and **Effect of Phase Prolog-Epilog Merging on Loop Nest** solves the problem and generates the code for prolog-epilog merging technique. The first algorithm plays geometry and puts different pieces of code, kernel, prolog and epilog, in the right place and the second one does necessary modifications to respect inter-phase dependences. The result is a code following the "prolog, kernel and epilog" pattern with the right iterator bounds and actor coordinates. The implementation confirms that this technique may be used for code compression as well, as we have seen "Prolog-epilog Merging" tries always to save the global loop code size.

Chapter 10

Experiments

10.1 Introduction

We have seen the 'Prolog-epilog Merging' techniques and its code generation in chapters 8 and 9. Now we are going to evaluate this technique, with real benchmarks. Our technique is applicable especially when it is not possible to apply loo fusion optimization technique. For this reason we will study some common media and signal-processing applications to see if our technique has a chance to be applicable or not then we evaluate the 'Prolog-epilog Merging' efficiency

10.2 Prolog-epilog Merging Applicability

We studied common media and signal-processing applications, including GNU radio, 802.11a (from Nokia), and polyphase image upscaling (from Philips Research).[1] Combining preliminary transformations including inlining, loop rerolling, fusion and if-conversion [2], we could find many occurrences of the "global loop with nested phases" pattern. All these applications exhibit low-trip-count phases, reinforcing the motivation for prolog-epilog merging. This pattern is also found in many scientific codes; since we target embedded systems, we only studied one of those: the computationally intensive part of the 172.mgrid SPEC CPU2000fp benchmark.

We report on the algorithmic underpinnings and optimization interplays of our approach. This step is required before undertaking a large integration effort into a back-end optimizer. Figure 10.1 provides basic statistics about the four applications we studied. It demonstrates the widespread occurrence of loop nests amenable to prolog-epilog merging. The varying trip-counts across neighboring phases (often due to data-dependent control) indicates that loop

[1] Three benchmarks studied in the ACOTES and SARC FP6 European projects: `http://www.hitech-projects.com/euprojects/ACOTES`, `http://www.sarc-ip.org`.

fusion is not generally applicable. We also verified the presence of many dependence cycles, at all depths, in the four benchmarks; all such cycles contain output/anti-dependences. This confirms the relevance of our global optimization problem with array renaming.

Benchmark	Lines of code	Phases at depth...					Dependences			$\neq$ trip counts
		1	2	3	4	5	Flow	Anti	Output	across phases
`GNU radio`	427	10	n.a.	n.a.	n.a.	n.a.	3	3	0	100%
`802.11a`	1502	16	n.a.	n.a.	n.a.	n.a.	5	5	2	50%
`Upscaling`	150	16	n.a.	n.a.	n.a.	n.a.	10	18	6	25%
`172.mgrid`	502	5	3	20	17	4	37	37	92	100%

Fig. 10.1. Applicability of prolog-epilog merging

10.3 Experiment Results on the Polyphase Image Upscaling Benchmark

This section presents our first results on the polyphase image upscaling benchmark. This code iterates on SD (720×480) YUV video frames and interpolates pixels to double the resolution in both dimensions (1440×960). It accesses a $N^2 \times 512$ lookup-table and two N^2 temporary arrays to iteratively apply filtering, interpolation, and image-enhancement steps over $N \times N$ blocks of pixels. Most time is spent in three-dimensional, imperfectly nested loops, spanning over 150 lines of C code whose control-flow skeleton is depicted in Fig. 10.2. The 16 phases are labeled A to P. Most of them have N^2 iterations except a couple with $N^2 - 1$. The value of N can be as low as 2 for low-quality interpolation and can grow beyond 5 for very high quality filtering. The default value is $N = 3$ (a typical 3×3 stencil).

```
for (ln = 0; ln < iheight; ln++) {
  for (px = 0; px < iwidth; px++) {
    //// Phase A
    for (index = 0; index < N*N; index++) { ... }
    ...
    //// Phase P
    for (index = 0; index < N*N; index++) { ... }
  }
}
```

Fig. 10.2. Skeleton of the interpolation code

All 16 phases can be pipelined (independently of prolog-epilog merging): the dependence graph for this kernel features *some intra-phase loop-carried* dependences but those are associated with reductions and do not hamper pipelining. There are no flow (read-after-write) *inter-phase* dependences, but many output and anti-dependences on the two N^2 temporary arrays; those dependences can be removed through array renaming. The strongly connected components are $\{A, B, C\}$, $\{D, E, H, K, N\}$, $\{F, G\}$, $\{I, J\}$, $\{L, M\}$, $\{O, P\}$.

Memory	Profit
0	918
16	918
36	1107
72	1161
108	1215
144	1269

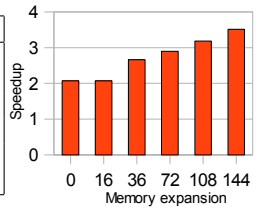


Fig. 10.3. Trading memory for performance

To break all dependence cycles in the inter-phase dependence graph, the maximal renaming cost is $4 \times N^2 \times 4$ bytes: 4 dependences removed through the renaming of arrays of N^2 32 bit integers. This is very little, both w.r.t. the size of most local memories or L2 caches, and w.r.t. the code size itself. It suggests that array renaming may be a practical solution to allow to pipeline most phases while maintaining the memory overhead close to zero.

Nevertheless, considering the default value $N = 3$, we evaluated the impact of array renaming, varying the upper bound on memory expansion from 0 to $16N^2 = 144$ bytes. The static cycle count for one iteration of the global loop is 1773; the linear optimization problem yields the profit (in static cycles) and speedup figures in Fig. 10.3. This experiment exhibits 5 steps where extra memory expansion translated into effective improvement of the total profit. It confirms the soundness and relevance of the array renaming for pipelining tradeoff, but more benchmarks should be studied before lessons about the analytical properties of this tradeoff can be learnt.

The next experiment we conducted concerns the interplay of our technique with loop nest optimizers. We studied the behavior of ICC 10.1, the state-of-the-art optimizing compiler from Intel, targeting the Itanium 2 processor (Madison) 1.3 GHz. Among the high-level loop transformations, ICC can perform loop tiling, unroll-and-jam and loop fusion. Only the latter is relevant for this streaming code with little temporal locality. The optimization log shows that 3 pairs of phases are fused, the only relevant ones (in terms of performance) being phases A and B. ICC fails to fuse two phases because of mismatching loop trip counts ($N^2 - 1$), and it fails to fuse phases L to P due to non-uniform or misaligned dependences. After fusion, 13 phases remain to exercise our technique. This example first shows that our technique is interesting as a *complement* to loop fusion: phases A and B could be fused yet still exhibit opportunities for pipelining and prolog-epilog merging; in addition, our technique is applicable in cases where loop fusion is not.

The last set of experiments aim to evaluate the overheads of prolog-epilog merging w.r.t. plain inner loop pipelining. Those overheads correspond to the extra instruction to compute shifted index variables (see Section 8.6.2) and to the register pressure induced by live inter-phase variables (see Section 8.7.1). We considered multiple architecture-compiler pairs: Intel Core 2 Duo 2.4 GHz and Intel Itanium 2 (Madison) 1.3 GHz with GCC 4.3 and ICC

10.1, IBM Cell PowerPC 3.2 GHz with GCC 4.1, STMicroelectronics ST231 400 MHz (embedded VLIW, 4-issue) with st200cc 1.9.0B (Open64). We used -O3 optimization, with pipelining turned off, with and without loop unrolling, and manually pipelined the most significant phases (source-level). In all cases, prolog-epilog merging performed better than unpipelined code, and sometimes even better than plain pipelining (phase L with GCC). We also verified that lower values of N improve the benefit of prolog-epilog merging: no iteration is spent on startup/flush except at the very beginning/end of the global loop [40]. This preliminary experiment confirms that the intrinsic overheads of our technique can be amortized.

10.4 Conclusion

The study of these benchmarks demonstrates the widespread occurrence of loop nests amenable to prolog-epilog merging. The varying trip-counts across neighboring phases (often due to data-dependent control) indicates that loop fusion is not generally applicable and 'Prolog-epilog Mergining', especially with renaming, has proved its efficiency.

Chapter 11

Conclusion

Streaming applications are not any general applications. Much better results can be exepected when using special languages designed for them. One of the most important point in these languages design is Scheduling. In this thesis we study how to schedule and optimize a streaming application statically, taking into account code size and buffer size simultaneously. We will see in the following sections our contributions and future work

11.1 Our Contributions

The goal of our work is to focus on the problem of streaming application scheduling, it is one of the most important and difficult point in conception of special-purpose stream language. Our contributions are:

1. **Conception Idea:** developing a fine-grained scheduling approach: an actor is a block of few instructions without loop or branch statement, this is useful to build a fine-grained schedule and to exploit offered parallelism; an SDF schedule is an infinite execution of a periodic sequence, steady state and this sequence schedule is exactly a loop nest schedule where actors replace instructions, with some differences like the interphase dependences. Hence, all our scheduling works are based on this similitude between nested loop schedule and steady state one. In this way, we take advantage of nested loops scheduling methods and exploit machine parallelism perfectly. Also, loops scheduling and parallelism take advantage of our proposed techniques and algorithms because they are applicable for both nested loops and steady state (Sections 6.2) and 6.3).
2. **Pre-scheduling:** a coarse-grain scheduling algorithm. It tries to find a schedule that respects code size and buffer size constraints by choosing a schedule that respects code size constraint and trying to modify it to respect buffer size as well (Section 6.4). .

3. **INMS Framework:** it is a dedicated form of multidimensional retiming for streaming computations, to break down dependence relations between actors and exploit parallelism. It does both an explicit shifting of inner loops and an implicit shifting of the outer loop. By shifting some actor occurrences it gives more chance to phases to be pipelined and to parallelism extraction and by merging prolog and epilog it tries to respect the code size constraint (Section 6.5).
4. **Pattern Table Shifting:** in this heuristic, we have tried to implement INMS using "resource table shifting" idea for "epilog filling in with other phases firings" shifting technique. It represents a good solution but only for simple cases, no cycle and at most one actor occurrence per phase (Chapter 7).
5. **Prolog-epilog Merging:** it is the solution we have proposed for both streaming applications and nested loops. It may appear as the most natural extension to inner loop pipelining. Indeed, it avoids the code size and startup time overhead of nested prologs and epilogs: these advantages over loop unrolling are exactly the motivations that drove to the design of software pipelining algorithms [32]. We formalized the concept of prolog-epilog merging, combining inner loop pipelining with multidimensional retiming (Chapter8).
6. **Combining Prolog-epilog Merging with Renaming:** we combine our technique with array renaming to pipeline more phases. This results in a global scheduling and memory expansion tradeoff, modeled as a tractable, integer linear optimization problem (Section 8.6).
7. **Code generation process:** with its both algorithms **Phase Prolog-Epilog Merging** and **Effect of Phase Prolog-Epilog Merging on Loop Nest**, it solves the problem and generates the code for for prolog-epilog merging technique. The first algorithm puts different pieces of code, kernel, prolog and epilog, in the right place and the second one does necessary modifications to respect inter-phase dependences. The result is a code looks like "prolog, kernel and epilog" with right iterators bounds and actor coordinates.

11.2 Future Work

We believe that prolog-epilog merging is a good idea and is a very natural extension of inner loop pipelining to imperfectly nested loops. We will continue working on it trying to:

- focus on combining our technique with other optimization techniques like loop fusion, loop skewing, loop interchange.

- automatize the solution, to develop a tool that has as input a streaming application code or nested loops and as output a code well scheduled and optimized with prolog-epilog merging technique.
- integrate prolog-epilog merging in used compilers like GCC.
- generalize our technique for any number of nest levels.
- apply prolog-epilog merging with any technique that generates prolog and epilog.
- apply prolog-epilog merging to solve code compression problems.
- answer the question if is it interesting to combine between both shifting techniques, 'Prolog-epilog Merging' and 'Epilog Filling'?

Bibliography

[1] V. H. Allan, R. B. Jones, R. M. Lee, and S. J. Allan. Software pipelining. *ACM Computing Surveys*, 27(2):367–432, 1995.

[2] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan and Kaufman, 2002.

[3] R. Allen, K. Kennedy, and Randy. *Optimizing Compilers for Modern Architectures:A Dependence-based Approach*. Morgan and Kaufman, 2001.

[4] S. Amarasinghe, M. L. Gordon, M. Karczmarek, J. Lin, D. Maze, R. M. Rabbah, and W. Thies. Language and compiler design for streaming applications. (3), 2005.

[5] R. Andonov and V. Poirriez. Unbounded knapsack problem : dynamic programming. *European Journal of Operational Research*, 123: 2. 168–181, 2000.

[6] D. F. Bacon, S. l. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, (7), 1993.

[7] V. Bove, Jr., and J.A. Cheops: A reconfigurable data-flow system for video processin. *IEEE trans. On Circuits and Systems for Video Technology*, 5(6):140–149, April 1995.

[8] T. Brandes. The importance of direct dependences for automatic parallelization. *International Conference of Supercomputing*, pages 407–417, 1988.

[9] S. Carr, C. Ding, and P. Sweany. Improving software pipelining with unroll-and-jam. In *Proceedings of the 29th Hawaii Intl. Conf. on System Sciences (HICSS'96) Volume 1: Software Technology and Architecture*. IEEE, 1996.

[10] P. Caspi, G. Hamon, and M. Pouzet. Lucid synchrone, un language de programmation des systèmes réactifs. In *Systèmes Temps-réÌel: Techniques de Description de VéÌrification - ThéÌorie et Outils*, pages 217–260. Nicolas Navet. Hermes, 2006.

[11] L.-F. Chao. Scheduling and behavioral transformations for parallel systems. *PhD thesis, Dept. of Comput. Sci.Princeton Univ*, 1993.

[12] A. Cohen, M. Duranton, C. Eisenbeis, C. Pagetti, F. Plateau, and M. Pouzet. N-synchronous kahn networks , a relaxed model of synchrony for real-time systems. *ACM International Conference on Principles of Programming Languages POPL'06*, (1), january 2006.

[13] J. Cortadella, R. M. Badia, and F. Sanchez. A mathematical formulation of the loop pipeling problem. (10), 1995.

[14] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Prog. Lang. Syst.*, 13(4):451–490, Oct. 1991.

[15] A. Darte and G. Huard. Loop shifting for loop parallelization. *Intl. J. of Parallel Programming*, 28(5):499–534, 2000.

[16] A. Darte, G.-A. Silber, and F. Vivien. Combining Retiming and Scheduling Techniques for Loop Parallelization and Loop Tiling. *Parallel Processing Letters*, 7(4):379–392, 1997.

[17] S. L. G. David F. Bacon and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4), 1994.

[18] J. C. Dehnert, P. Y. Hsu, and J. P. Bratt. Overlapped loop support in the Cydra 5. In *Intl Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'89)*, pages 26–38, Apr. 1989.

[19] C. Dulong, R. Krishnaiyer, D. Kulkarni, D. Lavery, W. Li, J. Ng, and D. Sehr. An overview of the Intel IA-64 compiler. *Intel Technical Journal*, Q4, 1999.

[20] P. Feautrier. Array expansion. In *Intl. Conf. on Supercomputing (ICS'88)*, pages 429–441, St. Malo, France, July 1988.

[21] P. Feautrier. Dataflow analysis of array and scalar references. *Int. J. Parallel Programming*, 20(1):23–51, 1991.

[22] P. Feautrier. Some efficient solutions to the affine scheduling problem, part I, multidimensional time. 21(6):315–348, Dec. 1992.

[23] P. Feautrier, M. Griebl, and C. Lengauer. On index set splitting. In *(PACT'99)*, Newport Beach, CA, Oct. 1999.

[24] M. P. Gerlek. Beyond induction variables: detecting and classifying sequences using a demand-driven ssa form. *ACM Trans. Prog. Lang. Syst.*, 17(1):85–122, Jan. 1995.

[25] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *Intl. J. of Parallel Programming*, 2006. Special issue on Microgrids. 57 pages.

[26] M. Hagog and A. Zaks. Swing modulo scheduling for gcc. *GCC developers' Summit*, (14), 2004.

[27] J. L. Hennessy and D. A. Patterson. *Computer Architecture: a quantitative approach (3rd ed.)*. Morgan Kaufmann, 2003.

[28] Z. L. Junjie Gu and G. Lee. Symbolic array dataflow analysis for array privatization and program parallelization. *Supercomputing 95*, 1995.

[29] M. Karczmarek. Thesis, constrained and phased scheduling of synchronous data flow graphs for streamit language. (13), December 2002.

[30] M. Karczmarek, W. Thies, and S. Amarasinghe. Phased scheduling of stream programs. *LCTES'03*, (12), 2003.

[31] M. Lam. Software pipelining: an effective scheduling technique for vliw machines. *SIGPLAN*, pages 318–328, June 1988.

[32] M. S. Lam. Software pipelining: An effective scheduling technique for vliw machines. In *ACM Principles, Logics, and Implementations of High-Level Programming Languages*, 1988.

[33] C. E. Leiserson and J. B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1):5–35, Dec. 1991.

[34] S. Martello and P. Toth. A mixture of dynamic programming and branch-and-bound for the subset-sum problem. *Manag. Sci*, 30:765-771.

[35] S. Martello and P. Toth. Knapsack problems: Algorithms and computer implementation. *John Wiley and Sons*, 1990.

[36] D. E. Maydan, S. P. Amarasinghe, and M. Lam. Array dataflow analysis and its use in array privatization. *In Principles of Programming Languages*, 1993.

[37] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam. Array dataflow analysis and its use in array privatization. In *Principles of Programming Languages (PoPL'93)*, pages 2–15, Charleston, South Carolina, Jan. 1993.

[38] K. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, july 1996.

[39] C. McNairy and D. Soltis. Itanium 2 processor microarchitecture. *IEEE Micro*, pages 44–55, Mar. 2003.

[40] K. Muthukumar and G. Doshi. Software pipelining of nested loops. In *Intl. Conf. on Compiler Construction (CC'01)*, 2003.

[41] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, 1986.

[42] T. M. Parks. Bounded scheduling of process networks. *PhD thesis*, (21), 1995.

[43] R. Parra-Hermandez and N. J. Dimopoulos. A new heuristic for solving the multichoice multidimensional knapsack problem. *IEEE Transactions on Systems, Man, and Cybernetics — Part A: Systems and Humans*, 35(5), Sept. 2005.

[44] Y. R. Pierre-Yves Calland, Alain Darte and F. Vivien. On the removal of anti and output dependences. *Application Specific Systems, Architectures and Processors. IEEE Computer Society Press*, pages 353–364, 1996.

[45] G. Plateau and M. Elkihel. A hybrid algorithm for the 0-1 knapsack problem. *Methods of Oper. Res. Journal*, 49:277–293, 1985.

[46] M. Pouzet. Lucid synchrone, un langage synchrone d'ordre superieur. mémoire d'habilation à diriger des recherches. (11), 2002.

[47] M. Pouzet. Lucid synchrone version 3.0, tutorial and reference manual. (16), April 2006.

[48] J. Puchinger, G. R. Raidl, and U. Pfershy. The multidimensional knapsack problem: Structure and algorithms. *Technical Report No. 006149 INFORMS Journal of Computing*, Mar. 2007.

[49] J. Ramanujam. Optimal software pipelining of nested loops. In *International Symposium on Parallel Processing*, pages 335–342, Washington D.C., 1994.

[50] B. R. Rau. Iterative modulo scheduling: an algorithm for software pipelining loops. In *MICRO 27: Proceedings of the 27th annual international symposium on Microarchitecture*, pages 63–74, New York, NY, USA, 1994. ACM Press.

[51] S. Rixner, W. J. Dally, U. J. Kapasi, B. Khailany, A. L?pez-Lagunas, P. R. Mattson, and J. D. Owens. A bandwidth-efficient architecture for media processing. *International Symposium on Microarchitecture*, (18), November 1998.

[52] H. Rong, Z. Tang, R. Govindarajan, A. Douillet, and G. R. Gao. Code generation for single-dimension software pipelining for multi-dimensional loops. In *Proceedings of the International Symposium on Code generation and Optimization(CGO'04)*, pages 175–186, Mar. 2004.

[53] H. Rong, Z. Tang, R. Govindarajan, A. Douillet, and G. R. Gao. Single-dimension software pipelining for multi-dimensional loops. In *Proceedings of the International Symposium on Code generation and Optimization(CGO'04)*, pages 163–184, Mar. 2004.

[54] D. P. S. Martello and P. Toth. Dynamic programming and strong bounds for the 0-1 knapsack problem. *Manag. Sci*, 45:414–424, 1999.

[55] R. Stephens. A survey of stream processing. *Acta Informatica*, 34(19):491–541, October 1995.

[56] A. Stoutchinin. An integer linear programming model of software pipeling for the mips r8000. *PACT: Proceedings of the 4th International Conference on Parallel Computing Technologies*, (4), 1997.

[57] B. Thies, M. Karczmarek, and S. Amarasinghe. Streamit: A language for streaming application. *Technical Memo*, august 2001.

[58] W. Thies. An empirical characterization of stream programs and its implications for language and compiler design. *PACT*, September 2010.

[59] W. Thies, M. Karczmarek, and S. Amarasinghe. Streamit: A language for streaming application. *International Journal of parallel programming*, 33(17), june 2005.

[60] S. Touati and C. Eisenbeis. Early Control of Register Pressure for Software Pipelined Loops. In *Intl. Conf. on Compiler Construction (CC'03)*, Warsaw, Poland, Apr. 2003. Springer-Verlag.

[61] A. N. U. Banerjee, R. Eigenmann and D. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, 1993.

[62] N. Vasilache, A. Cohen, and L.-N. Pouchet. Automatic correction of loop transformations. In *Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'07)*, Brasov, Romania, Sept. 2007.

[63] S. Verdoolaege, M. Bruynooghe, G. Janssens, and F. Catthoor. Multi-dimentsional incremetal loops fusion for data locality. In *ASAP*, pages 17–27, 2003.

[64] N. Y. Vincent Poirriez and R. Andonov. A hybrid algorithm for the unbounded knapsack problem. *Discrete Optimization Journal*, 2009.

[65] E. Waingold and et al. Baring it all to software: Raw machines. *IEEE Computer*, 30(8):86–93, September 1997.

[66] J. Wang, C. Eisenbeis, M. Jourdan, and B. Su. Decomposed software pipelining: a new perspective and a new approach. *Int. J. Parallel Program.*, 22(22):351–373, 1994.

[67] M. Wolfe. More iteration space tiling. *Supercomputing Journal*, pages 655–664, 1989.

[68] M. Wolfe. *High Performance Compilers For Parallel Computing*. Addison-Wesley Publishing Company, 1996.

[69] M. E. Wolfe and M. A. Lam. Data locality optimizing algorithm. *PLDI Journal*, pages 30?44, 1991.

[70] H. Zima and B. Chapman. Supercompilers for parallel and vector computers. *ACM Press*, 1990.