A PETRI-NET MODEL FOR LOOP SCHEDULING

by

Yue-Bong Wong

School of Computer Science
McGill University, Montréal

November 1991

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE

Abstract

This thesis describes a compile-time loop scheduling scheme and a supplementary storage reduction scheme to generate code for computer architectures which exploit fine-grain parallelism, such as superscalar, VLIW, and superpipeline machines.

In the first part we propose a new loop scheduling technique based upon the notion of *dataflow software pipelining*. We use Petri-net theory as the modeling framework, both for describing program behavior and for proving the feasibility of our approach. The time complexity of computing a schedule for an ideal machine model is examined under various program structures; a polynomial bound is established for the scheduling approach. We then integrate resource limitations into the model and construct a unified Petri-net model for schedule generation. Simulation results are conducted on a number of Livermore loops to verify the feasibility of the approach.

In the second part we discuss the application of a program restructuring scheme, known as *limited balancing*, for storage reduction [GHW90a, GHW90b]. With this technique, storage is systematically reduced across a loop body according to a *balancing ratio*. A guideline is derived to determine an appropriate ratio for maintaining a given pipeline utilization.

Résumé

Cette thèse décrit une technique statique d'ordonnancement de boucles et une méthode de réduction d'usage de mémoires pour générer du code pour des architectures d'ordinateurs qui utilisent le parallélisme "fine-grain," telles que les architectures de type "superscalar," "VLIW," et "superpipeline."

Dans la première partie de cette thèse, nous proposons une nouvelle technique pour l'ordonnancement d'instructions basée sur la notion de "Software Pipelining." Nous utilisons la théorie des Réseaux de Pétri en tant que support du modèle, pour décrire le comportement des programmes, et pour prouver la validité de notre approche. La complexité en temps de calcul d'un ordonnancement est évaluée pour plusieurs structures de programmes en utilisant un modèle de machine idéale; une limite polynomiale est établie pour la méthode d'ordonnancement. Nous intégrons alors les limites sur les ressources dans le modèle et construisons un modèle de réseaux de Pétri unifié pour la génération d'ordonnancement. Des simulations sont effectuées sur plusieurs boucles de Livermore pour vérifier la faisibilité de cette méthode.

Dans la seconde partie, nous décrivons l'application d'une méthode de réduction de l'espace mémoire pour supporter la méthode d'ordonnancement proposée précédemment [GHW90a, GHW90b]. Avec cette amélioration, l'espace mémoire est systématiquement réduit partout dans la boucle, en accord avec le *rapport de balancement* de la boucle. Nous dérivons alors une technique pour estimer un rapport de balancement approprié pour le taux d'utilisation du pipeline.

Acknowledgments

I would like to express my gratitude to my thesis advisor, Professor Guang R. Gao, for his constant support, guidance, and extensive discussion throughout this research.

I am also very grateful to my colleagues and friends for helping me in various ways. Jean-Marc Monti solved a lot of the technical problems I encountered on the system. Russell Olsen spent much time as my proof reader. Russell and his wife Yoshiko also invited me to so many of their delightful and fun dinners. Jing Wu and Jean-Marc accompanied me during many long hours in the lab. All of these people made my stay in Montreal a most enjoyable experience. Last but not least, I am also very grateful to Herbert Hum and Qi Ning for their many valuable discussions related to this research. Without their valuable input this work would not have been possible.

Finally, I owe my greatest thanks to my parents and brother, for their generous love, support, and care, especially my brother Dominic who cooked all of our meals while our parents were on vacation. Without my family's support, my education would not have been possible.

Contents

Abstract ii

Résumé iii

Acknowledgments iv

1 Introduction 1
- 1.1 Architecture Model Assumption 2
- 1.2 Software Pipeline Scheduling 2
- 1.3 Storage Reduction: Limited Balancing 5
- 1.4 Overview of Results 6
- 1.5 Thesis Outline 7

2 Dataflow Model 9
- 2.1 Static Dataflow Model 10
- 2.2 Dataflow Software Pipelining 12
  - 2.2.1 Dataflow Software Pipelining on Ideal Machines 13
- 2.3 Loop Representation and Loop Domain 14

3 Petri-net Modeling 16
- 3.1 The Model and Related Notation 16
- 3.2 Marking and Firing Rules 17
- 3.3 Liveness, Boundness, and Persistence 18
- 3.4 Some Special Structures 18
- 3.5 Marked Graphs 19
- 3.6 Timed Petri Nets 19
- 3.7 Optimal Computation Rate 20

4 Software Pipeline Scheduling on an Ideal Machine 23
- 4.1 Modeling a SDSP with a Petri Net 23

4.2 The Behavior graph of SDSP-PN 25
4.3 Steady State 27
4.4 Complexity to Reach a Cyclic Frustum 28
4.4.1 An SDSP-PN having One Critical Cycle 29
4.4.2 An SDSP-PN having Multiple Critical Cycles 35
4.4.3 Tightness of the Bound 39
4.5 Initial Token-Distribution Constraint 41
4.5.1 A Tighter Initial Period 41
4.5.2 A Second Approach for a Tighter Initial Period 43
4.6 Remarks 45

5 Software Pipeline Scheduling with Pipeline Constraint 47
5.1 Model with a Single Clean Pipeline—SDSP-SCP-PN 47
5.2 Multiple Clean Pipelines—SDSP-MCP-PN 52
5.3 Simulation Results 53
5.4 Discussion 56

6 Storage Allocation 58
6.1 Memory Model 59
6.2 Limited Balancing of an SDSP-PN 60
6.3 Limited Balancing of an SDSP-SCP-PN 65
6.4 Limited Balancing of an SDSP-MCP-PN 68
6.5 Experimental Results 69

7 Related Work 73
7.1 Software Pipelining 73
7.1.1 Perfect Pipelining 74
7.1.2 Enhanced Pipelining 75
7.1.3 The URPR Algorithm 76
7.1.4 The Systolic Array Optimizing Compiler 77
7.1.5 Remarks 78
7.2 Storage Allocation 78

8 Conclusion and Future Research 81

A Example: A-code graph for Loop 3 83

Bibliography 85

List of Tables

4.1 Single Source 40
4.2 Multiple Sources 40
5.1 Results for SDSP-PN Model 54
5.2 Results for SDSP-MCP-PN Model with Eight Stages 55
6.1 Results for Utilization Rate Estimation 71

List of Figures

1.1 Schedule Computation 4
1.2 The Concept of Balancing Ratio 6
2.1 Dataflow Graph 9
2.2 Execution Snapshot 10
2.3 Static Dataflow Graph 12
2.4 Software Pipelining of a Dataflow Program 13
2.5 Example L2 14
4.1 SDSP-PN of L1 and L2 24
4.2 An Example of the Behavior Graph for the SDSP-PN of L1 25
4.3 An Example of Steady-State Equivalent Net 28
4.4 Code Sequence with Single Source 38
4.5 Code sequence with Multiple Sources 38
4.6 A Code Sequence with an $\mathcal{O}(n)$ Lower Bound 40
5.1 SDSP-SCP-PN and the Behavior Graph 48
5.2 SDSP-MCP-PN and the Behavior Graph 53
6.1 Storage Usage of Argument-flow Model versus Argument-Fetch Model 59
6.2 Minimum Storage Allocation 62
6.3 A-code for Loop 9 63
6.4 Loop 9 under Partial Limited Balancing 64
6.5 Loop 9 under Aggressive Limited Balancing 72
A.1 A-code Graphical View of Loop 3 84

Chapter 1

Introduction

With today's technology, multiple functional units can be incorporated onto a single chip, significantly increasing parallel processing power. Superscalar, Very Long Instruction Word (VLIW), and superpipeline machines are typical architectures created using VLSI technology [Fis83, GO90, KM89, Lam89, Mel89]. To effectively utilize the increased machine parallelism of these machines requires fine-grain (instruction-level) parallelism within the source application. Therefore, the exploitation of fine-grain parallelism becomes a major issue in effective compiler design. Efficient loop execution, in particular, has attracted much attention because this is where a processing unit spends a significant amount of time during program execution. The first part of this thesis explores the use of *dataflow software pipelining* for compile-time loop scheduling for the exploitation of fine-grain parallelism. Dataflow software pipelining is a scheme for structuring fine-grain parallelism in the loop body in a way so that it can be exploited by static dataflow computer.

The second part of the thesis discusses the application of a graph restructuring scheme called *limited balancing* [GHW90c]. Limited balancing can be used to reduce the amount of storage usage in dataflow software pipelining. Numerous surveys have shown that the response time of main memory is a major bottleneck which prevents ideal speedup from being achieved in high performance computer architectures. The use of expensive high-speed memory, or *register sets*, for temporary storage to reduce memory accesses plays an important role in maintaining processor throughput. Unfortunately, register sets are a scarce resource, and their ineffective use leads to significant performance degradation. As a result, the study on storage reduction to reduce the amount of register required is another crucial element in compiler design.

1.1 Architecture Model Assumption

For pipelined computer architectures, *hazards* are a main cause of performance degradation [HP90, Kog81]. *Structural* hazards arise from resource conflicts when hardware cannot support simultaneous operations by two, possibly independent, instructions. Structural hazards increase the difficulty of code generation. The standard software approach to avoid pipeline anomalies caused by hazards is to insert delays, such as NOOPs (NO OPeration instruction), between the two operations that conflict. The length of the required delay is called the *interlock* period. To resolve structural hazards, the compiler must find sufficient parallel instructions to fill the interlock, thus keeping the pipeline usefully busy. However, it is unlikely that an efficient code scheduling approach can be found since scheduling with structural hazards is NP-hard [NPA88].

Code scheduling has also been examined under conditions in which the pipeline is free of structural hazards [BG89, HG83]. Processor pipelines of this type are called *clean*. Much of the scheduling effort focused only on acyclic constraint graph. It has been proven that scheduling a clean pipeline is NP-complete if the maximal delay on directed edges of the constraint graph is unbounded [HG83] and is polynomial time solvable if the delay equals one [BG89]. Note that the latter case applies to a single clean pipeline consisting of two stages. Nonetheless, the notion of building a clean pipeline has not been widely adopted because no code scheduling technique yet developed could justify its worthiness.

Recent findings by Nicolau, Pingali, and Aiken on clean pipeline scheduling presents a new insight into the problem of loop scheduling [AN88, NPA88]. They propose a polynomial time loop-scheduling scheme and prove that time-optimal results are always achievable for a class of loop programs that have no loop-carried dependence while suboptimal results are guaranteed for the same class of loops with loop-carried dependence. Based upon these findings, they conclude that the trend in architectures will be to avoid structural hazards as much as possible. In this thesis we also focus our loop scheduling on machines which use *clean* execution pipelines, and all references to an execution pipeline in subsequent sections will be clean pipelines, unless otherwise stated.

1.2 Software Pipeline Scheduling

In this thesis we are interested in applying the concept of dataflow software pipelining to a compile-time loop scheduling scheme for computer architectures other than

dataflow, such as tightly-coupled synchronous parallel machines (e g , superscalar and VLIW machines) and various other pipelined architectures

Dataflow software pipelining is an effective loop structuring scheme for a static dataflow architecture by enabling the architecture to exploit fine-grain parallelism during loop execution [GP90] The strength of the scheme lies in its ability to expose fine-grain parallelism across loop boundaries Intuitively, it arranges code (a static dataflow graph) for loop bodies into a *software* pipeline so that successive iterations can be initiated one after the other. In other words, dataflow software pipelining allows the initiation of a new iteration before the previous iteration ends, achieving the same effect as a hardware pipeline. In the dataflow model of computation, an instruction is eligible for execution as soon as all of its required inputs are available. As a result, many waves of computation can proceed in a pipelined fashion through one copy of the dataflow program graph.

Compile-time scheduling involves the generation of a static schedule which preplans virtually all run time behavior. To apply the concept of dataflow software pipelining into compile-time loop scheduling requires two fundamental schemes. A code mapping scheme which *compiles* the given loop body into a semantic equivalent software pipeline, expressed at the *instruction* level, and a static schedule computation scheme which generates code from the software pipeline A rigorous study on pipelined-code mapping for scientific applications can be found in [Gao90]. In this thesis we focus on establishing a static schedule computation scheme. Here we assume that a loop body has already been compiled into a software pipeline. To avoid ambiguity, we refer to the static schedule computation scheme as *Software Pipeline Scheduling*, or SPS for short.

In SPS, static schedule generation for loops relies on the existence of a *repetitive execution sequence*, also known as the *steady state*. This repetitive execution sequence remains the same regardless of the number of iterations. Thus, the run-time behavior of the loop can always be expressed finitely with the same schedule. This finite schedule comprises three segments: *prelude sequence*, steady state, and *postlude sequence*. The prelude is the sequence of operations leading to the steady state, while the postlude is the sequence of operations required to complete loop execution following the steady state. From the perspective of a hardware pipeline, prelude and postlude sequences correspond to the sequence of operations which fill and drain the software pipeline.

Figure 1.1 illustrates the method of generating a static schedule using SPS. Figure 1.1(a) gives an example loop body, and Figure 1.1(b) lists the dataflow information

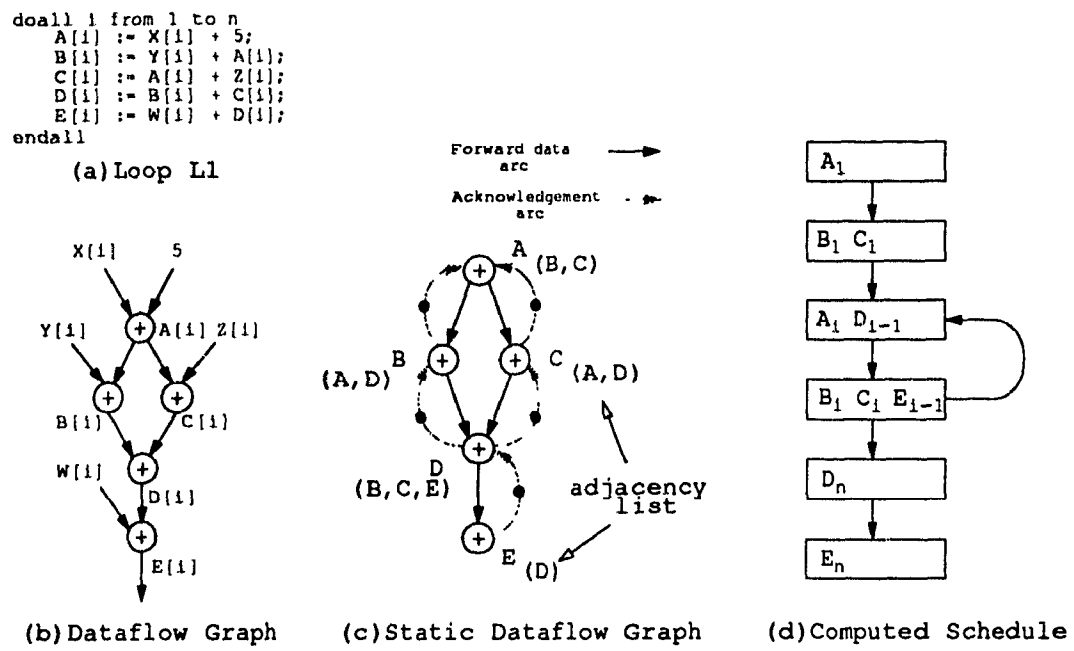


Figure 1.1: Schedule Computation

required to map the loop body to a software pipeline. The instruction-level representation we employ for the loop body is a *static dataflow graph* (see Figure 1.1(c)). The first advantage of using a static dataflow graph is that it operates naturally as a software pipeline. With its one-token-per-arc policy it also constrains execution to a bounded amount of storage while exploiting fine-grain parallelism. To derive a static schedule from the software pipeline, we apply to the static dataflow graph the execution rule of dataflow computation, the rule being, an instruction is eligible for execution as soon as all inputs are available. Pictorially, the token on an arc represents the availability of the particular input. The execution of a node is represented by the removal of an input token from each input arc and the production of a result token on each output arc. Figure 1.1(d) shows the resulting execution sequence. Initially only node A is eligible for execution. Once the execution of node A completes, nodes B and C start and are followed by nodes A and D. From then on the repeated firing sequence is formed by alternately activating the two groups of nodes BCE and AD, the third and fourth rows of the computed schedule. Note that the execution sequence is a semantic equivalent schedule for loop L1. If L1 is executed n times, the steady state of the schedule is iterated for $n-1$ times.

For SPS to be an effective compile-time loop scheduling scheme, several questions need to be answered: Does there always exist a steady state for loop execution? What is the time complexity required to generate a schedule? How does the scheme

work for a machine with multiple pipelined-units? The methodology used to study these questions is based upon Petri-net theory [Chr84, CHEP71, Mur89, Ram74]. The strong resemblance between the Petri-net model and the dataflow model allows a direct and natural application of the developed theories in Petri net to dataflow.

Before we move on we need to point out that, while the software pipeline is a mirror image of a hardware pipeline, the microprogram used to control the hardware pipeline under a time-stationary microprogramming scheme shows a strong resemblance to the static schedule derived from the software pipeline [Kog77]. In fact, the earliest idea of software pipelining was first applied to optimize the microprogramming control of a pipelined processor. Just as the function of a microprogram is to control the operation of the hardware pipeline, the computed static schedule can also be viewed as the microprogram for controlling the software pipeline. The resemblance of the two structures relates the notion of software pipelining directly to loop scheduling.

1.3 Storage Reduction: Limited Balancing

Limited balancing is a program restructuring scheme for reducing the synchronization overhead which is intrinsic in a static dataflow design [GHW90c]. However, its application has a significant impact on storage reduction, thus presenting a new perspective for register allocation in SPS. This storage reduction scheme is performed before the scheduling phase. By applying a balancing technique, storage can be systematically reduced across a loop body according to the loop's particular *balancing ratio*—a computed parameter that characterizes the achievable *computation rate* of the final schedule. In effect, the computation rate is the frequency of executing a node over a long period of time. As will be shown, the computation rate of a loop executing on an ideal machine equals the computation rate of the slowest simple cycle in the graph. Such a cycle is called a *critical cycle*. Limited balancing of a graph thus requires all simple cycles to decrease their computation rates as much as possible to the same rate imposed by the critical cycles. One important outcome is that the computation rate of the modified loop does not change.

Shown in Figure 1.2 is an example application of limited balancing using the balancing ratio. Suppose that the given code sequence is run on machine with adequate parallelism and the execution time of each node takes l cycles. The tokens used in the graph can be viewed as the amount of resources used. *G1* represents a computation rate of $1/3l$ due to the simple cycle ABC. *G2* represents a restructuring of *G1* with respect to the factor $1/3l$ so that no resources are unnecessary wasted. Note that after limited balancing the computation rate is maintained, and yet the

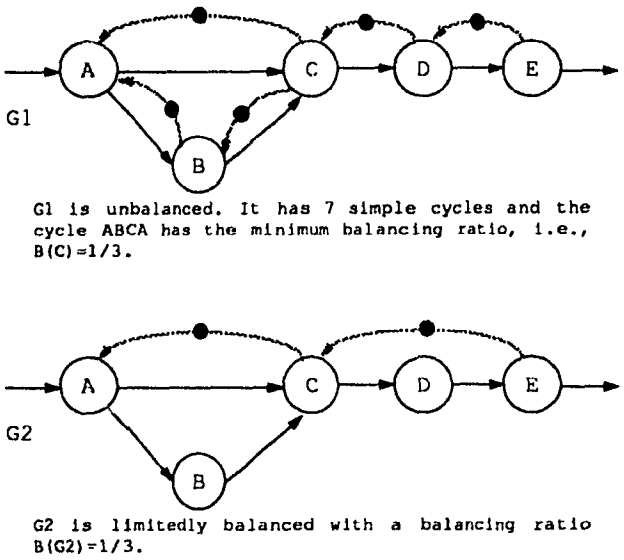


Figure 1.2: The Concept of Balancing Ratio

amount of required resources are reduced. One important issue is how to determine an appropriate balancing ratio such that enough parallelism is exposed to maintain the maximum throughput of the execution pipeline.

1.4 Overview of Results

The results of this thesis are presented in two parts: compile-time loop scheduling and register allocation. At the beginning of the first part, the time complexity required for the formation of the steady state for a class of loops operated on an ideal model is studied with a class of Petri nets known as marked graphs. Here is a summary of our findings:

- Under an ideal machine model, for a class of loops with only one critical cycle, the steady state appears after $\mathcal{O}(n^3)$ iterations, where n denotes the size of the loop body [GWN91a]. For the case of multiple critical cycles, the length of the steady state is directly proportional to a common multiple factor of the critical cycles; we are unaware of any polynomial bound for the length of the prelude sequence in this case.

- Nodes on the critical cycles have one special property—they fire periodically after $\mathcal{O}(n^2)$ iterations [GWN91a, GWN91c].

- A constraint which leads to a tighter polynomial bound for the length of the prelude sequence is established: When the starting condition of the loop body

meets the constraint, the steady state appears after $\mathcal{O}(n)$ iterations, regardless of the number of critical cycles. Most importantly, this constraint can always be satisfied in $\mathcal{O}(n)$ iterations.

The results based upon the first two points indicate that perfect polynomial time complexity cannot be established for the SPS scheme for programs with multiple critical cycles. The third result above indicates, however, that we can bypass this difficulty and derive an approach with polynomial time complexity by fixing an initial condition. Doing so, we were able to achieve a significant improvement in efficiency in finding a schedule.

At the end of the first part of this document, a multiple-pipeline model involving hardware pipelines and program graph is composed and applied to explain the influence and property of clean pipeline on our scheduling approach. Using the multiple-pipeline model we show how the steady-state schedule can again be obtained and serves as a schedule methodology for general pipelined architectures. Preliminary simulation performed on a set of Livermore Loops illustrates the feasibility of computing compile-time schedule by this approach. In the second part of this document, register allocation is discussed for an ideal model and a multiple-pipeline model. For the former case, the execution rate of the critical cycle can be used directly as the balancing ratio in the limited balancing scheme. For the latter case, a mathematical guideline is derived to estimate the appropriate balancing factor so that a certain percent utilization of the machine pipeline can be guaranteed.

1.5 Thesis Outline

This thesis can be boardly divided into four major parts: The first part provides the necessary background to understand the dataflow execution model and Petri-net model. It also reviews the concept of dataflow software pipelining and defines the class of loop upon which we focus. The second part introduces approaches for establishing a static schedule with SPS for an ideal machine, a single-clean-pipelined machine, and a multiple-clean-pipelined machine. In addition, the time complexity of generating a static schedule is studied. The third part discusses limited balancing, its application on reducing the synchronization cost, particularly for the dataflow model, and its application to the problem of storage reduction. Then derivations of an appropriate balancing ratio for the three models are presented. The last part of the thesis consists of a discussion of related work, a conclusion and future work. The appendix attached at the end describes briefly the graphical view of A-code- a program representation in the form of a signal graph (for the static dataflow argument-fetch model developed

at McGill University) to facilitate understanding of the code sequences illustrated in Chapter 6.

Chapters 2 and 3 are especially important to understand this document: Chapter 2 gives an introduction to the general dataflow execution model and the static dataflow model. It then introduces the concept of dataflow software pipelining and defines the class of loops under examination at this stage. Chapter 3 provides the reader with the necessary background to understand the Petri-net theory we use.

Part two of the thesis consists of two chapters: Chapter 4 introduces the framework for generating a static schedule in SPS for an ideal machine. It shows that the steady state is always reachable in polynomial time based upon Petri-net theory. Chapter 5 establishes the framework for the multiple pipelines.

Part three of the thesis consists of a single chapter: Chapter 6 introduces *limited balancing* and discusses its application on synchronization cost reduction and storage reduction based upon two models: the static dataflow *argument-flow* model and the static dataflow *argument-fetch* model. The concept of balancing ratio and its estimation are also discussed.

Part four of the thesis consists of Chapters 7 and 8: Chapter 7 compares the set of related works from other groups: Nicolau and Aiken's perfecting pipelining, Ebcioglu's enchancing pipelining, Su, Ding, and Xia's URPR algorithm, Lam's software pipelining, and the valid schedule computation originally by Reiter. Finally, Chapter 8 concludes the thesis by summarizing our achievements and pointing out topics for future research.

Chapter 2

Dataflow Model

The dataflow model of computation offers a natural way of expressing and exploiting fine-grain parallelism in an application [AC86]. An abstract view of the operational model is best explained using a directed graph. In the context of dataflow, such graphs are called dataflow graphs. Each node in the dataflow graph stands for an operation or an instruction, also known as *actor*. Directed arcs drawn between operations decide the partial order implied by data dependences. The generation and consumption of data values in the course of computation are pictorially depicted by data tokens traveling along the arcs in the graph. A token on a directed arc indicates the availability of an input for the dependent node produced by the source node. A node is said to be *enabled* if all of its input data are available, and it is indicated by the arrival of tokens on each input arc of the node. An enabled instruction is eligible for execution (or firing) at any time. This type of synchronous control for computation is known as *data-driven*, as opposed to the technique of using a program counter, as in conventional computer designs. The result of executing a node is indicated by removing a token from each input arc and generating a result token on each output arc. Multiple instructions can be executed simultaneously, depending upon machine parallelism. Token distribution at an instant reflects the current state of the model.

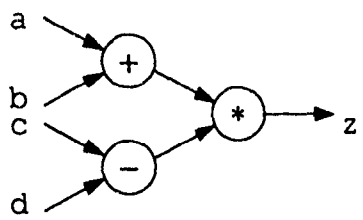


Figure 2.1: Dataflow Graph

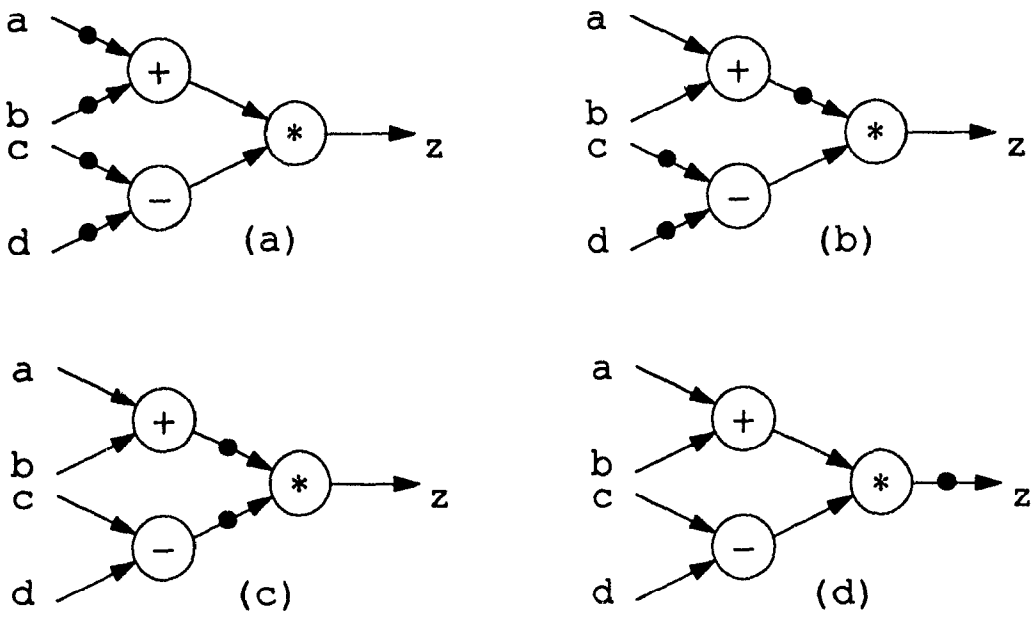


Figure 2.2: Execution Snapshot

An example dataflow graph for the computation $z = (a + b) \times (c - d)$ is shown in Figure 2.1. Figure 2.2 portrays the execution of the abstract model using three execution snapshots. Initially, the addition and subtraction nodes are enabled with one token on each of their input arcs (Figure 2.2(a)). After execution of the addition node, a result token is generated on the node's output arc, and the subtraction node remains enabled (Figure 2.2(b)). As the subtraction node carries out its operation, the multiplication node becomes enabled (Figure 2.2(c)). Figure 2.2(d) shows a final snapshot of the computation.

2.1 Static Dataflow Model

There are two variations of dataflow: *static* dataflow [Den84, DG88, Den91] and *dynamic* dataflow [Aea83, AG82, ADNP88]. The static model enforces a one-token-per-arc policy in which a node is said to be enabled, thus ready for execution, as soon as all of its input arcs are filled and all of its output arcs are empty. This restriction constrains the graph to at most one activated instance per node at any time. Since each token stands for a data value, this restriction assures the use of finite space during graph execution. In contrast, dynamic dataflow has no restriction on the number of tokens per arc; there is no limit on the number of activated instances per node at any time. Thus, there is no a-priori bound on the amount of storage

required to support concurrent activations. In addition, the arrival of tokens might be out of order. To ensure no ambiguity during execution, tokens generated from the same iteration, or the same invocation, are tagged with the same color. A node is said to be enabled as soon as there exists a token with the same color on each input arc. The dynamic dataflow model is known to be able to exploit the maximum parallelism in an application. In this thesis we use static dataflow since storage is under program control.

To formally enforce the one-token-per-arc constraint required for static dataflow, *acknowledgement* arcs are used. For each data arc (a, b) in the graph, an acknowledgement arc (b, a) is attached pointing in the opposite direction. As node b completes execution, it deposits a result token on each data arc and a *signal* token on each acknowledgement arc. The signal token serves to notify the predecessor (node a) that it can safely begin a new execution without damaging an earlier result. Besides the acknowledgement arc, the firing rule for the abstract model is refined to preserve the *static* quality of the graph:

- An node is enabled if there is a token on each of its input arcs.
- An enabled node can be fired by removing a token from each input arc and depositing a token on each output arc.

Figure 2.3(a) gives an example static dataflow graph for the computation $z = (a + b) \times (c - d)$. Figure 2.3(b) shows a snapshot after executing both the addition and subtraction operations. As illustrated, all input tokens are consumed; output tokens carring the associated results are sent to their data arcs to be used by the multiplication node, and signal tokens are sent along their acknowledgement arcs to notify the unseen operations presumably on the left to reload their inputs again. Suppose that the input arcs of the addition and subtraction nodes are reloaded at this point as shown in Figure 2.3(c); both operations cannot restart without receiving a control signal from the multiplication node to confirm the use of the earlier result. As a result, the one-token-per-arc policy is enforced by the acknowledgement arcs. After the multiplication completed, the addition and subtraction nodes, as well as the unseen operations presumably waiting for the result of z, are eligible for execution.

The above examples exhibit two types of parallelism: *spatial* parallelism and *temporal* parallelism. The former one is represented by any two simultaneously enabled nodes which have no dependence, such as the addition and subtraction actors. The latter parallelism is demonstrated by the pipelining of independent waves of data through the graph, for example, the re-enabled addition and subtraction nodes.

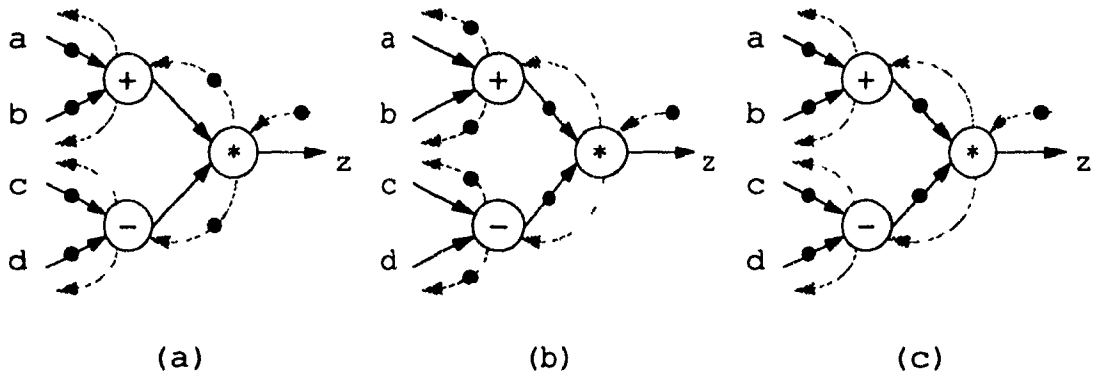


Figure 2.3: Static Dataflow Graph

Since the dataflow model has no notion of a single locus of global control, as does its Von Neumann counterpart, the execution of enabled actors are not restricted to any particular order. For implementation of loop and conditional constructs in a dataflow graph, a set of *well-behaved* graph schemas have been developed to govern interconnection [DFL72]. Under the schemata restriction, a computation always yields the same result, unaffected by the execution order of enabled actors. In other words, the schemas ensure *determinate* computation. A larger program is merely a hierarchical composition of elementary sub-schemas.

2.2 Dataflow Software Pipelining

Dataflow software pipelining is a pipelined code mapping strategy performed on units of program text called *code blocks*. Code blocks define the major structured values involved in a computation. Dataflow software pipelining is particularly effective for implementing array operations on a static dataflow computer. For example, the following loop body takes as input two arrays A and B and produces another array X:

for i **in** $1, n$
$\quad X[i] = ((2 * A[i])^2 + (2 * B[i])^2)^2$
end for

For this block, the corresponding dataflow graph can be easily software pipelined. The technique of dataflow software pipelining involves the arrangement of machine code such that successive computations can follow each other through one copy of the code. If we present a sequence of values to the inputs of the dataflow graph,

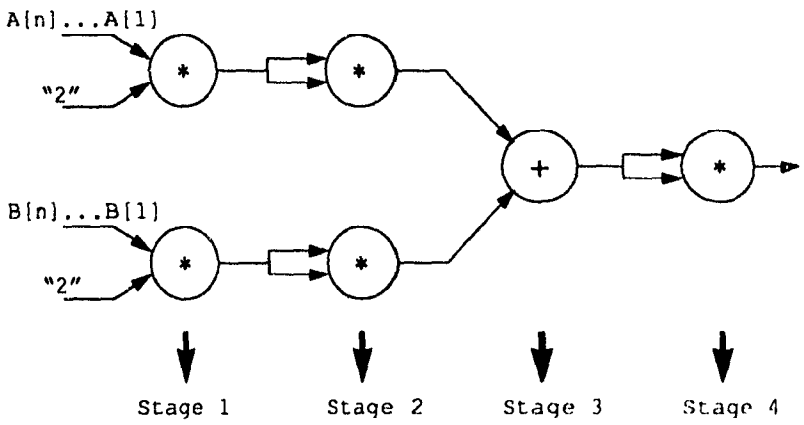


Figure 2.4: Software Pipelining of a Dataflow Program

these values flow through the program in a pipelined fashion. In the above example, successive elements of the input array A and B are fetched and fed into the dataflow graph, e.g., $A[1], A[2], \ldots, A[n]$ and $B[1], B[2], \ldots, B[n]$; thus, computation proceeds in a pipelined fashion. Instructions which belong to the same stage can be executed in parallel since there are no data dependencies among them. Moreover, during the pipelined execution of the program, multiple stages can be executed concurrently stages 1 and 3 are enabled and can be executed in parallel, and the same applies to stage 2 and stage 4. The power of fine-grain parallelism can be derived from programs which form a large pipeline in which many instructions in multiple stages execute concurrently. For the static dataflow model, software pipelining is essential for exploiting the parallelism within a loop body, and thus, it is a necessary optimization for numerical scientific applications.

2.2.1 Dataflow Software Pipelining on Ideal Machines

Dataflow software pipelining which was proposed as a model for structuring fine-grain parallelism has been studied mostly under the conditions of an idealized dataflow architecture, one having infinite resources [Gao89]. Here we provide a summary of some of the main results of previous research using this model. A graph is *balanced* if every path from an input node to an output node contains exactly the same number of actors. A graph is maximally pipelined if it is balanced [Gao86]. To achieve maximum pipelining, a basic technique called *balancing* is used to transform an unbalanced dataflow graph into a balanced one. This is done by introducing FIFO buffers on certain arcs. To optimally balance a graph, a minimum amount of buffering is introduced such that execution can be fully pipelined. It is known that optimal balancing

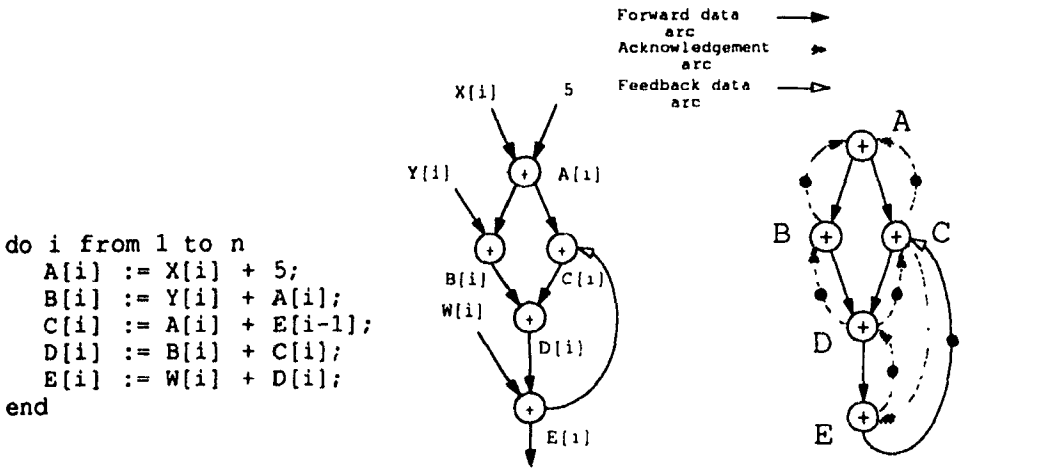


Figure 2.5: Example L2

of an acyclic dataflow graph can be formulated into a linear programming problem which has efficient algorithmic solution. A dataflow compiler uses these algorithms to perform code optimization.

2.3 Loop Representation and Loop Domain

Static dataflow graphs are used as the instruction-level representation for the loop body. The advantages of using a static dataflow graph are that it operates naturally as a software pipeline, and it constrains the execution model to use a bounded amount of storage, by its one-token-per-arc policy, while exploiting fine-grain parallelism.

The class of loops which we focus on here are called *Static Dataflow Software Pipeline* (SDSP) loops. They are non-nested. For the case of nested loops, our code generation technique is applied directly to the innermost loop where the processor often spends most its execution time. Conditional constructs are omitted from the loop body. The existence of conditional branches presents a harder problem for constructing a static schedule because of unpredictable branching behavior. The subject of including conditional constructs is currently under research. Loop-carried dependence is restricted to span across one iteration only, so the size of a loop body stays within a manageable limit. This class of loops has a simplicity which allows the corresponding software pipeline to be obtained in a straightforward manner.

Figure 2.5(a) is an example of a loop body with a loop-carried dependence, and Figure 2.5(b) shows the associated dataflow graph. The arc which expresses the loop-carried dependence is called a *feedback* data arc. The rest of the arcs are called

forward data arcs. Note that this graph only presents dataflow information within the loop body. To obtain the corresponding static dataflow graph, acknowledgement arcs are introduced (see Figure 2.5(c)). Note that a complete representation of a loop in the formal dataflow model involves control actors, such as *merge* and *switch* [DFL72]. Control actors are omitted in the discussion for simplicity.

A SDSP G, consistent with above assumptions, can formally be expressed as the tuple:

$$(V, E, E', F, F')$$

V is the set of actors (or vertices) in G. E and E' are respectively the set of forward data arcs and the set of feedback data arcs. F and F' are the set of acknowledgement arcs for E and E'. Figures 1.1(c) and 2.5(c) illustrate two possible candidates of an SDSP, one with and the other without loop-carried dependence.

Chapter 3

Petri-net Modeling

Since the original dissertation of C. A. Petri was published in 1962 [Pet62], Petri-net theory has emerged as an important tool for system analysis and modeling of a wide range of applications Petri-net theory allows a system to be modeled by a Petri net, a mathematical representation. Analyzing the modeled system can reveal important information about the system's structure and dynamic behavior. This information can be beneficially used to suggest system improvements

3.1 The Model and Related Notation

A Petri net PN is a three-tuple (P, T, A), where P is a non-empty set of places denoted by $\{p_1, p_2, \ldots, p_n\}$, T is a non-empty set of transitions denoted by $\{t_1, t_2, \ldots, t_m\}$, and A is a non-empty set of directed arcs such that $P \neq \emptyset$, $T \neq \emptyset$, $P \cap T = \emptyset$, $A \subseteq P \times T \cup T \times P$. Pictorially, P, T, and A are represented by circles, bars, and directed arcs, respectively.

By convention, *dot notation* has been employed as a means of simplifying the representation for a set of places and for a set of transitions. Shown below is the list of possible usages of dot notation, where P_1 and T_1 denote the subset of P and T in PN. In addition, (t, p) denotes the directed arc from t to p while (p, t) denotes the directed arc from p to t.

- $\cdot p = \{t \mid (t, p) \in A\}$ (the set of input transitions),
- $p\cdot = \{t \mid (p, t) \in A\}$ (the set of output transitions),
- $\cdot P_1 = \bigcup_{\forall p_i \in P_1} \cdot p_i$,
- $P_1\cdot = \bigcup_{\forall p_i \in P_1} p_i\cdot$,

- $\cdot t = \{p \mid (p,t) \in A\}$ (the set of input places),
- $t\cdot = \{p \mid (t,p) \in A\}$ (the set of output places),
- $\cdot T_1 = \bigcup_{\forall t_i \in T_1} \cdot t_i$,
- $T_1\cdot = \bigcup_{\forall t_i \in T_1} t_i\cdot$,
- $|\cdot s|$ and $|s\cdot|$ denote the number of elements in the set $\cdot s$ and $s\cdot$ respectively, where s can be a place/transition or set of places/transitions.

3.2 Marking and Firing Rules

A *marking* of a net is a function $M : P \rightarrow Z^+$, where Z^+ is the set of non-negative integers. The non-negative integer associated with a place p, denoted by $M(p)$, represents the number of *tokens* on the place. A Petri net with a marking is always referred as a *marked* Petri net. Marking M_0 is always referred to as the initial marking of a net.

A transition t in Petri net PN is said to be *enabled* by the marking M, denoted by $M \xrightarrow{t}$, if and only if $\forall p \in \cdot t$, $M(p) > 0$. An enabled transition can be *fired*. The firing of an enabled transition t is done by removing one token from each of the input places $p \in \cdot t$ and depositing one token on each of its output places $p \in t\cdot$. Assuming the marking which enables t be M and the marking which is obtained by firing t be M', firing can be expressed as $M \xrightarrow{t} M'$.

A marking M' is said to be *reachable* from M if M' can be obtained by firing an enabled transition t, $M \xrightarrow{t} M'$, or by firing a sequence of transitions $\pi = t_a t_b \ldots t_i$, $M \xrightarrow{t_a} M_a \xrightarrow{t_b} \cdots \xrightarrow{t_i} M'$. In the latter case, $\pi = t_a t_b \ldots t_i$ is termed the *firing sequence*. A firing sequence π is called *cyclic firing sequence* if, for any marking M, $M \xrightarrow{\pi} M$ and π is not empty. Let σ be a firing sequence. Then $f(\sigma)$ is called the *firing vector* of σ, where $f(\sigma)_i$, denoting the i-th element in the vector, is the number of occurrences of transition t_i in σ.

The *forward marking class* $\vec{M}$ of a marking M is the set of markings that are reachable from M. Conceptually, each distinct marking of a Petri net represents a distinct state in the modeled system. Similarly, the forward marking class $\vec{M_0}$ of the marking M_0 represents the set of reachable states of the modeled system

3.3 Liveness, Boundness, and Persistence

A marking M is *live* for a transition t if and only if for every marking M_i in the forward marking class $\vec{M}$ there exists a firing sequence which fires t. A marking M is live for a Petri net PN if and only if it is live for every transition in the net. If PN represents a model of a system, the liveness property of PN implies that the modeled system will never deadlock.

A marking M is *bounded* for a place p if and only if there exists an integer N such that for every marking $M_i \in \vec{M}$, $M_i(p) \leq N$. If $N = 1$, the marking M is called safe for p. A marking M is bounded (or safe) for the Petri net PN if and only if M is bounded (or safe) for every place in the net. Note that if PN is bounded, the set of reachable states of the modeled system must be finite.

A Petri net is *persistent* if and only if for all $t_1, t_2 \in T, t_1 \neq t_2$ and any reachable marking M, $M \xrightarrow{t_1}$ and $M \xrightarrow{t_2}$ imply $M \xrightarrow{t_1 t_2}$ (the firing of transition t_2 after the firing of transition t_1); i.e., if t_1 and t_2 are enabled at a reachable marking, the firing of one cannot disable the other; otherwise, it is said to have *choice.*

3.4 Some Special Structures

- *Self-loop* is a transition which has both input and output from the same place.
- A Petri net PN is said to have structural *conflict* if there exists a place p in PN such that $|p\cdot| > 1$. The existence of structural conflicts is a necessary condition for situations where choice might occur.
- A Petri net model is said to be *consistent* if and only if there exists a non-zero integer assignment to each transition in the net (where each arc is assumed to carry the integer of its attached transition), such that at each place, the sum of the integers assigned to each of its input arc equals to the sum of the integers assigned to each of its output arc. The assignment ensures the existence of repeatable behavior in the model so that it is meaningful to talk about *cycle time.* Here are the two known theorems on consistency [Ram74]:

Theorem 3.1 *A Petri net PN is consistent if and only if there exists an initial marking M for which there exists a cyclic firing sequence.*

Theorem 3.2 *A Petri net PN which has a live and bounded marking is consistent.*

A corollary of the above theorem is that, if a Petri net has a live and bounded marking, there exists a cyclic firing sequence.

3.5 Marked Graphs

A class of Petri nets which is important to our work is the class referred to as marked graphs.

Definition 3.1 *A Petri net* PN $= (P, T, A)$ *is called a marked graph if and only if* $\forall p \in P,\ |\cdot p| = |p \cdot| = 1$.

Marked graph must be *persistent* because, for each place in the graph, there is only one associated output transition. Here are some significant results for marked graphs (for proofs, see [CHEP71]):

Theorem 3.3 *A marking is live if and only if the token count of every simple cycle is positive.*[1]

Theorem 3.4 *A live marking is safe if and only if every edge in the graph is in a simple cycle with token count 1.*

Theorem 3.5 *If* π *is a cyclic firing sequence such that* $M \xrightarrow{\pi} M$, *all transitions have been fired an equal number of times.*

3.6 Timed Petri Nets

Adding the notion of time to the basic Petri-net model enables the characterization of system performance. In this thesis we consider that a deterministic time, expressed by a non-negative integer number, can be assigned to each transition in the basic Petri-net model [Ram74, RH80]. The model described below is made up of the original timed Petri-net model introduced by Ramchandani [Ram74], and the concept of instantaneous state subsequently developed by Chretienne [Chr85].

Formally, a *timed Petri net* is defined as a pair (PN, Ω), where PN is the basic Petri-net tuple (P, T, A) and Ω is a function that assigns a non-negative integer τ_i to each transition t_i in the net (i.e., $\Omega : T \rightarrow Z^+$, where Z^+ is the set of non-negative integers). The value τ_i denotes the *execution time* (or the firing time) of transition t_i.

The state of the timed Petri net at time u is no longer described only by the current marking at time u (M_u) because some transitions might still be processing at time u. A new concept of *residual firing time vector*, R, is introduced to keep track of on-going executions at each time step. $R_u(t_i)$ stores the remaining execution time

[1] A simple cycle is a directed path $p_i t_j p_k \ldots t_l p_m$ such that all places and transitions are different except p_i and p_m.

of transition t_i at time u. Accordingly, M_u and R_u together define the *instantaneous state* of a timed Petri net. We also make the following two assumptions regarding the firing rule of enabled transitions:

Assumption 3.1 *Two distinct firings of the same transitions cannot overlap. To formally enforce this rule, each transition in the net is assigned a distinct self-loop of its own with only one token in it. Though we do not draw them explicitly they are implicitly assumed.*

Assumption 3.2 *Transitions are fired as soon as they are enabled; this is termed the earliest firing rule.*

3.7 Optimal Computation Rate

Timed Petri nets have been applied in the study of concurrent systems to determine the *cycle time* or equivalently the *computation rate.* We next review the method for obtaining the cycle time of a marked graph.

Definition 3.2 *The cycle time of transition t_i is defined as*

$$\lim_{n\to\infty} \frac{X_i^n}{n}$$

where X_i^n is the time at which transition t_i initiates its $n+1$ execution.

Here are some important results for timed marked graph from Ramamoorthy and Ho [RH80]:

- The number of tokens in a simple cycle remains the same after any firing sequence.
- All transitions in a marked graph have the same cycle time.
- Cycle time is computed by

$$\alpha = \max\left\{\frac{\Omega(C_k)}{M(C_k)}, \Omega(t_i)\right\}$$

where $k = 1, 2, \ldots, q$ and $t_i \in T$;

$\Omega(C_k) = \sum_{t_i \in C_k} \Omega(t_i)$ is the sum of the execution times of the transition in simple cycle C_k;

$M(C_k) = \sum_{p_i \in C_k} M(p_i)$ is the total number of tokens in simple cycle C_k;

q is the number of simple cycles in the net, excluding the self-loop implicitly assumed for each transition; and

the cycle time of each self-loop is reflected by $\Omega(t_i), \forall t_i \in T$.

- The computation rate γ of a transition is the average number of firings of that transition in unit time and is computed by the reciprocal of the cycle time.

$$\gamma = \min\left\{\frac{M(k)}{\Omega(C_k)}, \frac{1}{\Omega(t_i)}\right\}$$

where $k = 1, 2, \ldots, q$ and $t_i \in T$

- The simple cycle C_k which gives the maximum cycle time, or equivalently the minimum computation rate, is known as the *critical cycle*.

In addition, Ramamoorthy and Ho demonstrates that a valid execution schedule supporting the optimal computation rate can always be computed for a live-bound marked graph. This result is stated in the following lemma and is used subsequently to deduce a polynomial bound in Chapter 4.

Lemma 3.1 *A valid execution schedule for each transition t_i can be derived with the following time constraint, once the cycle time α is determined:*

$$S_i^h = a_i + \alpha h$$

where S_i^h is the time at which transition t_i commences the $h+1$ firing, and a_i is the time at which transition t_i commences the first firing.

Cycle time α is

$$\max\left\{\frac{\Omega(C_k)}{M(C_k)}, \Omega(t_i)\right\}, \text{ for all simple cycles } C_k \text{ in } PN.$$

The starting time a_u of transition t_u can be assigned as follows:

1. Define the distance from transition t_i to transition t_j to be $\Omega(t_i) - \alpha M(p_{ij})$, where p_{ij} is the place in between transitions t_i and t_j.

2. Find a transition t_s, which is enabled initially and assign 0 to a_s.

3. Assign a_u to each transition t_u such that a_u is the greatest distance from t_s to t_u, i.e.,

$$a_u = \max_R\{\sum_{t_w \in R} \Omega(t_w) - \alpha \sum_{p_{ab} \in R} M(p_{ab})\}$$

where R is a path from t_s to t_u. Note that the single-source longest path algorithm can be applied here.

The cycle time of a timed marked graph can be obtained by enumerating every simple cycle in the associated graph; however, the time complexity of the enumerating process can be exponential because there exists a marked graph with an exponential number of simple cycles [Mag84]. A more efficient method for finding cycle time is given in [Mag84] where the problem is formulated as a linear programming problem having a theoretical polynominal bound.

The above computation rate γ is *optimal* or *time-optimal* in the sense that it is the maximum achievable computation rate under any machine model [RH80, Ram74]. It can be achieved when a model has enough parallelism to execute all enabled transitions as soon as they become enabled. Such an ideal machine model will be used in the next chapter.

Chapter 4

Software Pipeline Scheduling on an Ideal Machine

In this chapter we use Petri nets to examine the feasibility and complexity of software pipeline scheduling. Section 4.1 describes formally the SDSP using a Petri net; the resulting model is called a SDSP-PN. Once the SDSP-PN is constructed, we are then able to examine the repetitive behavior, or the steady state, resulting from the execution of the SDSP, using an ideal machine model and with the earliest firing rule enforced. Section 4.2 introduces the notion of a behavior graph [Ram74] which, together with the live-safe properties of an SDSP-PN, provide the means for proving the existence and uniqueness of the steady state discussed in Section 4.3. In Section 4.4 we determine the time-complexity required to reach steady state. In Section 4.5 we discuss an marking constraint which lowers the time-complexity requirement for reaching steady state.

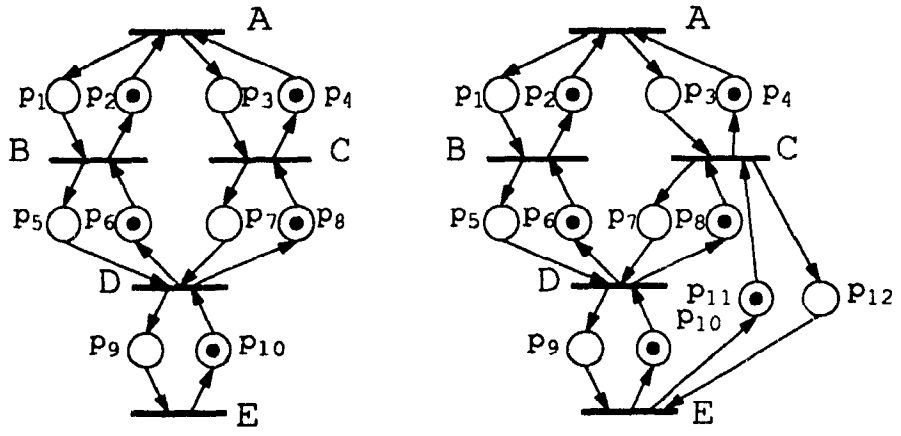
4.1 Modeling a SDSP with a Petri Net

It is straightforward to translate a SDSP into a Petri net called an SDSP-PN. The following algorithm performs the translation:

Algorithm I SDSP to SDSP-PN transformation

Input: A SDSP $G = (V, E, E', F, F')$, where $V = \{v_1, \ldots, v_n\}$, $E \cup E' \cup F \cup F' = \{e_1, \ldots, e_m\}$, and l is a constant (we assume all nodes have the same execution time).
Output: A SDSP-PN $PN = (P, T, A, \Omega)$

- For each vertex v_i in V, we introduce a corresponding transition t_i in T, i.e., $T = \{t_1, \ldots, t_n\}$.



(a) SDSP-PN of L1 (b) SDFP-PN of L2

Figure 4.1: SDSP-PN of L1 and L2

- $\forall t_i \in T, \Omega(t_i) = l$.
- For each directed edge $e_u = (v_i, v_j)$ in $E \cup E' \cup F \cup F'$, we introduce (1) a corresponding place p_u in P, i.e., $P = \{p_1, \ldots, p_m\}$; and (2) a set of directed arcs in A denoting the flow relations $\cdot p_u = t_i$ and $p_u \cdot = t_j$. A can be expressed as:

$$A = \bigcup_{e_u=(v_i,v_j) \in (E \cup E' \cup F \cup F')} ((t_i, p_u) \cup (p_u, t_j)).$$

In addition, the initial marking M_0 associated with PN will simply be

$$M_0(p_u) = 1, \text{ if } e_u \in E' \cup F,$$
$$M_0(p_u) = 0, \text{ if } e_u \in E \cup F'.$$

Figures 4.1(a) and (b) give the corresponding Petri-net representations for L1 and L2. The resulting PN is a marked graph due to the fact that $|\cdot p| = |p \cdot| = 1$. Accordingly, it is also *persistent.* Furthermore, the initial marking M_0 is live and safe due to the following two theorems:

Theorem 4.1 *SDSP-PN with initial marking M_0 is live.*

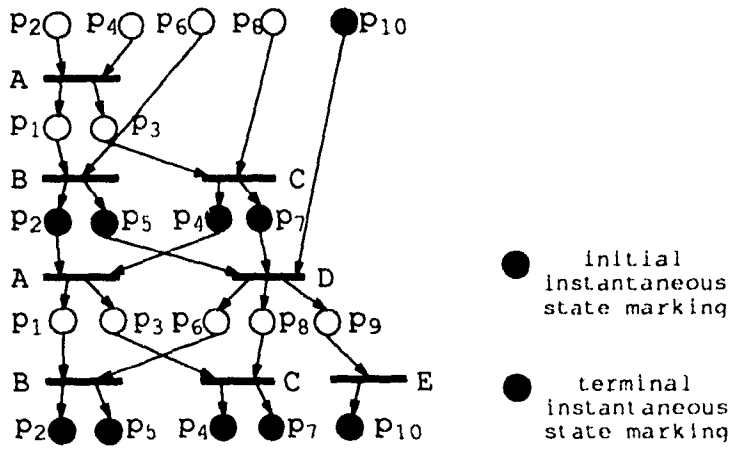


Figure 4.2: An Example of the Behavior Graph for the SDSP-PN of L1

Proof of Theorem 4.1.
Note that each possible cycle found in the resulting SDSP-PN contains at least one token. This can be seen by noting that every arc in the SDSP-PN pointing in the backward direction is initialized with one token. The validity of Theorem 4.1 therefore follows immediately from Theorem 3.3. □

Theorem 4.2 *SDSP-PN with initial marking M_0 is safe.*

Proof of Theorem 4.2.
For each arc $(v_i, v_j) \in E \cup F'$ (in G), the corresponding arc $(v_j, v_i) \in E' \cup F$ initially holds a token and points in the opposite direction. Accordingly, each edge in the resulting SDSP-PN is within a simple cycle having a token count of 1. Hence, the validity of this theorem follows directly from Theorem 3.4. □

4.2 The Behavior graph of SDSP-PN

The construction of a behavior graph provides an alternative way to describe the behavior of a Petri net, besides a reachability tree [Pet81]. A behavior graph is particularly useful for describing the concurrency and cyclic firing sequences of a Petri net. From a different standpoint, the behavior graph is actually a trace generated while executing the SDSP-PN, according to the earliest firing rule. At each time step, the behavior graph records the set of newly marked places and the set of enabled transitions to be fired at that step. In addition, directed arcs are introduced among the places and transitions to denote the token flow relation from place to transition (token consumption) and from transition to place (token production). The instantaneous

state of the behavior graph at time i can be described by the current residual firing time vector R_i and the current marking M_i. The algorithm for constructing the behavior graph is given below:

Algorithm II Behavior Graph Construction

Input: a SDSP-PN $PN = (P, T, A, \Omega)$ and initial marking M_0

Output: behavior graph B

Step 1 Initially $i = 0$. Let M' denote the set of currently marked places in PN. Duplicate M' in B. Initialize all entries of the residual firing time vector R_0 to 0.

Step 2 Fire all enabled transitions in PN. Let T' denote the set of transitions just fired. Duplicate a copy of T' in B. Update the residual firing time vector and current marking as follows:
$R_i(t_j) = \Omega(t_j), \forall t_j \in T'$,
$M_i(p_k) = M_i(p_k) - 1, \forall p_k \in {}\cdot T'$.

Step 3 Introduce directed arcs among places M' and transitions T' in B to indicate the token flow relation.

Step 4 $i = i + 1$. $M'' = \bigcup_{t_j \in S} t_j{}\cdot$, where $S = \{t_i \mid R_{i-1}(t_i) - 1 = 0, \forall t_i \in T\}$, i.e., M'' is the set of newly marked places. Duplicate M'' in B and update the residual firing time vector and the current marking as follows:
$R_i(t_j) = R_{i-1}(t_j) - 1$, if $R_{i-1}(t_j) > 0, \forall t_j \in T$,
$R_i(t_j) = R_{i-1}(t_j)$, if $R_{i-1}(t_j) = 0, \forall t_j \in T$,
$M_i(p_k) = M_{i-1}(p_k), \forall p_k \in P$ and $\notin M''$,
$M_i(p_k) = M_{i-1}(p_k) + 1, \forall p_k \in M''$.

Step 5 Introduce directed arcs among transitions T' and places M'' in B to indicate token flow.

Step 6 Let M' denote the set of currently marked places in PN. Repeat from Step 2.

Figure 4.2 illustrates the behavior graph constructed for the marked graph, SDSP-PN L1, shown in Figure 4.1(a), where the execution time of all transitions are assumed to be equal.

4.3 Steady State

As can be seen, the construction process of the behavior graph can continue forever, and the behavior graph can be infinitely extended. A key observation is that the behavior graph exhibits repetitive behavior after an *initial period* the amount of time elapsed before the repetitive behavior is reached. This is shown by the following lemmas:

Lemma 4.1 *Behavior graph is unique for SDSP-PN.*

Proof of Lemma 4.1.

Obviously the original marking of the marked graph is unique. Since there is no structural conflict in a marked graph, the firings of all enabled transitions at each time step with respect to the earliest firing rule are unique. Therefore, the validity of the lemma is immediate. □

Lemma 4.2 *There exists an instantaneous state in the behavior graph of SDSP-PN that appears repeatedly.*

Proof of Lemma 4.2.

The total number of distinct M_i is finite because SDSP-PN has a safe marking. Similarly, the total number of distinct R_i is also finite because each transition in SDSP-PN has a known firing time. As a result, the total number of possible instantaneous states are also finite. Hence, if the behavior graph is infinitely extended, some instantaneous states must be repeated. □

From Lemmas 4.1 and 4.2 we can see that an instantaneous state once repeated will do so forever. As a result, the region of the behavior graph between two repeated instantaneous states can be used to represent the steady-state behavior of the SDSP-PN executed under the earliest firing rule. Thus we have the following definition:

Definition 4.1 *A Cyclic Frustum (or steady state) of a behavior graph B is the portion of B between two consecutive occurrences of some repeated instantaneous state. In addition, the two instantaneous states that surround the frustum are termed the initial instantaneous state and the terminal instantaneous state.*

The marking portion of both the initial and terminal instantaneous state found in the behavior graph for L1 are marked in Figure 4.2, where the two associated residual firing time vectors are vectors composed of zero entries. Notice that the cyclic frustum is actually a cyclic firing sequence since it fires each transition at least once and returns

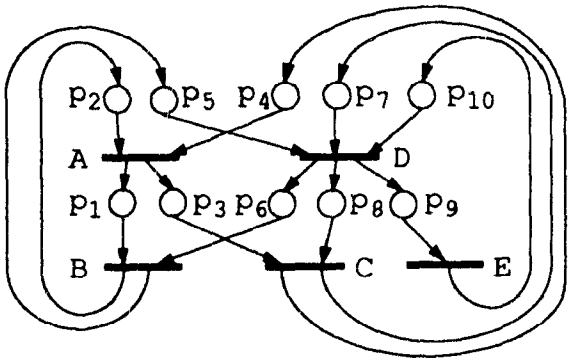


Figure 4.3: An Example of Steady-State Equivalent Net

the net to its initial state. Once the behavior graph reaches its frustum, it will keep repeating. This simply suggests a way of capturing the repetitive behavior of the studied system. Instead of extending the behavior graph indefinitely, we can extract the cyclic frustum and coalesce the initial and terminal instantaneous states to form a strongly-connected Petri net, called a *steady-state equivalent net.* Figure 4.3 shows an example of the steady-state equivalent net derived from the behavior graph shown in Figure 4.2. Note that the steady-state equivalent net is itself a marked graph. As we initialize it with an initial marking, by assigning one token to each coalesced place (i.e., the top row of places), the net captures the steady state behavior of the corresponding SDSP-PN, yielding the same computation rate.

4.4 Complexity to Reach a Cyclic Frustum

As was shown in Lemmas 4.1 and 4.2, a repetitive execution pattern can always be found for an SDSP-PN executed under the earliest firing rule within a finite number of steps. The length of the initial period is examined in two sections: In Section 4.4.1 we impose a theoretical bound on the length of the initial period for an SDSP-PN having one critical cycle, while in Section 4.4.2, we deal with the case of multiple critical cycles, noting the barrier confronted and giving some partial results. In Section 4.4.3 we provide an indication of the tightness of the theoretical bound obtained using sample code which requires $\mathcal{O}(n)$ iterations for the initial period. In Section 4.5 we introduce an initial token-distribution constraint. As the initial marking of an SDSP-PN meets the constraint, a tighter polynomial bound can be established for the initial period. More importantly, the result can be generalized to the case of multiple critical cycles, imposing a polynomial bound on the initial period. The work described in this section benefited both from Chretienne's thesis on Petri-net theory [Chr84] and from

Aiken and Nicolau's work [AN88]. The notations and assumptions which we will use are defined below:

- Let G denote a SDSP-PN having n transitions, and let X_i^h denote the time at which transition t_i commences its h+1 firing. We assume that the execution time τ_i of each transition t_i is one time unit. In general, however, the following results can be extended to cases in which transitions have different execution times.

- If P is a path in G, then $M(P)$, the *token* sum, denotes the sum of the tokens on each place in P.[1] The token in one place is taken in the sum as many times as the place is embedded in P. Similarly, $\Omega(P)$, the *value* sum, denotes the sum of τ_i of each transition t_i in P. The τ_i of transition t_i is taken in the sum as many times as the transition is embedded in P. Let $P_h(t_i, t_j)$ denote the set of possible paths in G from t_i to t_j having exactly h tokens along the path, and let $a_h(t_i, t_j)$ denote the value sum of the maximum value path in $P_h(t_i, t_j)$. We also use the notation $P_h^y(t_i, t_j)$ to denote the subset y of $P_h(t_i, t_j)$ and the notation $a_h^y(t_i, t_j)$ to denote the maximum path value of subset $P_h^y(t_i, t_j)$. Since each transition has a self-loop with one token on it (Assumption 3.1), $P_h(t_i, t_j) \neq \emptyset$, for $h \geq h_0$ where h_0 is a positive integer.

- A simple cycle C^* in G is *critical* if the ratio of the value sum to the token sum is maximal, i.e., if

$$\frac{\Omega(C^*)}{M(C^*)} \geq \frac{\Omega(C_i)}{M(C_i)},$$

where C_i denotes the other simple cycles in G. Let α_i denote the cycle time of the simple cycle C_i, i.e., $\alpha_i = \Omega(C_i)/M(C_i)$; likewise, let α^* $(= \Omega(C^*)/M(C^*))$ denote the cycle time of C^*.

4.4.1 An SDSP-PN having One Critical Cycle

Chretienne shows that a precise description of the action of each transition t_i under the earliest firing rule obeys the *global* time constraint $X_i^{h+k} - X_i^h = p, h \geq h_0$, where h_0 is a non-negative integer, k equals the least common multiple of the token sum of all critical cycles in G, and p equals $\alpha \times k$, where α is the maximum cycle time of G [Chr84].[2] This time constraint means that after the h_0 firings, every k-th firing of a transition t_i must be p time steps apart; that is, the steady state appears. In other

[1] Note that a cycle is allowed along a path

[2] The constraint is global in the sense that it is applicable to describe the behavior of all transitions

words, the length of the steady state is p time cycles, and the steady state consists of k firings of transition t_i.

Since, in this section, we only consider SDSP-PN having one critical cycle C^*, k and p equals $M(C^*)$ and $\Omega(C^*)$. This implies that the length of the steady state of G is bounded, bypassing the problem of determining an upper bound for the least common multiple of token sums in the case of multiple critical cycles. Theorem 4.3 states that in the single critical-cycle case, the global time constraint for each $t_i \in G$ is satisfied (i.e., the steady state is reached) after $\mathcal{O}(n^3)$ iterations, i.e., $h_0 = \mathcal{O}(n^3)$. Before this claim is proven, we introduce several important lemmas: The first is Lemma 4.3 which relates the time at which transition t_j starts its $h+1$ firing to the computation of $a_h(t_i, t_j)$, the value sum of the maximal value path in $P_h(t_i, t_j)$ [CCG84, Chr84, Chr85]. Lemma 4.4 establishes the criterion that the maximum value path must pass through the critical cycle. Lemma 4.5 states that a subset of a given set of k integers can always be found such that the total sum is a multiple of k [AN88]. Lemma 4.6 is an inequality based upon the fact that the value-per-token ratio of a critical cycle is always larger than that of a non-critical cycle.

Lemma 4.3 *For any G executed under the earliest firing rule, the time X_j^h at which transition t_j starts its $h+1$ firing equals*

$$\max_{t_i} a_h(t_i, t_j),\ t_i \in \textit{set of enabled transitions at time zero.}$$

Lemma 4.4 *For $h \geq \mathcal{O}(n^3)$, the maximum value path in $P_h(t_i, t_j)$ in G must pass through the critical cycle C^*.*

Proof of Lemma 4.4

Let Ψ be a path in $P_h(t_i, t_j)$ which does not touch C^*. Let P_a be a path in $P_h(t_i, t_j)$ which passes through C^*. For any given $h \geq h_0$, we choose $P_a = \mu(C^*)^m \nu$, where h_0 is an integer, μ and ν are respectively the directed path from t_i to t_y and the directed path from t_y to t_j, t_y is a transition on C^*, and m is the number of times that C^* is consecutively iterated. The value m and the paths μ and ν can respectively be computed and constructed as follows:

- $m = \left\lfloor \frac{M(P_a)-2n}{M(C^*)} \right\rfloor$
- Let $m_1 = (M(P_a) - 2n) \bmod M(C^*) + n$; μ is a path from t_j to t_y with m_1 tokens while ν is a path from t_y to t_j with n tokens.[3]

[3]Such paths must exist because the token sum of any simple path (cycle free) is bounded by n, and by Assumption 3.1 $P_h(t_a, t_b) \neq \emptyset$ provided $h \geq n$ in a safe marked graph.

Notice that under such construction

$$M(\mu) + M(\nu) \leq 3n. \tag{4.1}$$

Let us assume $S_m = M(\mu) + M(\nu) + m \times M(C^*)$ for the given h. Thus by definition both Ψ and P_a belong to $P_{S_m}(t_i, t_j)$. To prove this lemma, we first construct an upper bound on the value sum $\Omega(\Psi)$. Then we show that $\Omega(P_a)$ is always greater than the upper bound of $\Omega(\Psi)$ for $m \geq \mathcal{O}(n^2)$ or $h \geq \mathcal{O}(n^3)$.

- **Construction of the upper bound for $\Omega(\Psi)$**

  This construction was taken from Chretienne's Ph.D thesis [Chr84]. Let $\{C_i; i = 1, \ldots, a\}$ denote the set of non-critical simple cycles in G, and let ε and ε_i be defined as follows:

$$\begin{aligned} \varepsilon_i &= \alpha^* - \alpha_i, \ \forall i, 1 \leq i \leq a \\ \text{and } \varepsilon &= \min_{i=1,\ldots,a} \{\varepsilon_i\} \\ &> 0 \end{aligned}$$

  Recall that α^* and α_i are respectively the cycle time for the critical cycle C^* and the simple cycle C_i. Then, ε is the cycle time difference of the critical cycle and the simple cycle that has the second largest cycle time in G

  Notice that path Ψ can be decomposed into a simple path q (cycle free) running from t_i to t_j and a set of non-critical simple cycles, where for each C_i we associate an integer $\eta_i \geq 0$ to denote the number of times C_i is iterated in Ψ. The value sum of Ψ is computed as follows:

$$\begin{aligned} \Omega(\Psi) &= \Omega(q) + \sum_{i=1}^{a} \eta_i \Omega(C_i) \\ &= \Omega(q) + \sum_{i=1}^{a} \eta_i \alpha_i M(C_i) \\ &= \Omega(q) + \sum_{i=1}^{a} \eta_i (\alpha^* - \varepsilon_i) M(C_i) \\ &\leq \Omega(q) + \sum_{i=1}^{a} \eta_i (\alpha^* - \varepsilon) M(C_i) \\ &= \Omega(q) + (\alpha^* - \varepsilon) \sum_{i=1}^{a} \eta_i M(C_i) \\ &\leq \Omega(q) + (\alpha^* - \varepsilon) M(\Psi) \end{aligned}$$

  Since $\Psi \in P_{S_m}(t_i, t_j)$, $M(\Psi)$ can be expanded and bounded as follows:

$$\Omega(\Psi)$$

$$\begin{aligned} &\leq \Omega(q) + (\alpha^* - \varepsilon)\ (M(\mu) + M(\nu) + m\ M(C^*)) \\ &= m\ (\alpha^* - \varepsilon)\ M(C^*) + b \end{aligned} \tag{4.2}$$

where $b = \Omega(q) + (\alpha^* - \varepsilon)\ (M(\mu) + M(\nu))$

- **Evaluation of h so that $\Omega(P_a) > \Omega(\Psi)$**

Note that the path $P_a = \mu(C^*)^m\ \nu$ also belongs to $P_{S_m}(t_i, t_j)$, and $\Omega(\mu(C^*)^m\ \nu)$ equals $\Omega(\mu) + \Omega(\nu) + m\ \alpha^*\ M(C^*)$. As we compare $\Omega(\mu(C^*)^m \nu)$ and the upper bound of $\Omega(\Psi)$ (Equation 4.2), we see for $m > m_0$, where m_0 is some positive integer, Ψ can never be the maximum value path in $P_{S_m}(t_i, t_j)$. Moreover, this is true for all possible $\Psi \in P_{S_m}(t_i, t_j)$ that do not touch C^*.

$$\begin{aligned} &m\ (\alpha^* - \varepsilon)\ M(C^*) + b \\ <\ &m\ \alpha^*\ M(C^*) + \Omega(\mu) + \Omega(\nu) \end{aligned} \tag{4.3}$$

m_0 can be estimated by solving Equation 4.3 for m:

$$m\ >\ \frac{b - (\Omega(\mu) + \Omega(\nu))}{\varepsilon\ M(C^*)} \tag{4.4}$$

Next we simplify the right-hand side of Equation 4.4 and construct an upper bound. Without loss of generality in i, assume that ε_i is the smallest value in $\{\varepsilon_1, \ldots, \varepsilon_a\}$, i.e., $\varepsilon = \varepsilon_i = \alpha^* - \alpha_i$:

$$\begin{aligned} &\frac{b - (\Omega(\mu) + \Omega(\nu))}{\varepsilon\ M(C^*)} \\ \leq\ &\frac{b}{\varepsilon\ M(C^*)} \\ =\ &\frac{\Omega(q) + (\alpha^* - \varepsilon)\ (M(\mu) + M(\nu))}{(\alpha^* - \alpha_i)\ M(C^*)} \\ =\ &\frac{\Omega(q) + \alpha_i\ (M(\mu) + M(\nu))}{(\alpha^* - \alpha_i)\ M(C^*)} \\ =\ &\frac{\Omega(q) + \frac{\Omega(C_i)}{M(C_i)}\ (M(\mu) + M(\nu))}{(\frac{\Omega(C^*)}{M(C^*)} - \frac{\Omega(C_i)}{M(C_i)})\ M(C^*)} \\ =\ &\frac{\Omega(q) + \frac{\Omega(C_i)}{M(C_i)}\ (M(\mu) + M(\nu))}{\Omega(C^*) - \frac{\Omega(C_i) M(C^*)}{M(C_i)}} \\ =\ &\frac{M(C_i)\ \Omega(q) + \Omega(C_i)\ (M(\mu) + M(\nu))}{\Omega(C^*)\ M(C_i) - \Omega(C_i)\ M(C^*)} \end{aligned} \tag{4.5}$$

Note that G is a live-safe marked graph composed of n transitions. The token sum for any simple path or any simple cycle is bounded by n. Similarly, the

value sum of any simple path or any simple cycle is bounded by n. Note also that by Equation 4.1 $M(\mu) + M(\nu) \leq 3n$. Equation 4.5 can be further reduced and bounded by

$$\begin{aligned} & \frac{n \times n + n \times 3n}{1} \\ < \; & \mathcal{O}(n^2) \end{aligned} \tag{4.6}$$

As a result, for $m \geq \mathcal{O}(n^2)$ the maximum value path in $P_{S_m}(t_i, t_j)$ must pass through C^*, where $S_m = M(\mu) + M(\nu) + m \times M(C^*)$. Or equivalently, for $h \geq \mathcal{O}(n^3)$ the maximum value path in $P_h(t_i, t_j)$ must pass through C^*.

□

Lemma 4.5 *Given K integers $I_1, \ldots, I_k$, there is a subset S of I_i such that*

$$\left(\sum_{I_i \in S} I_i \right) \bmod k = 0.$$

In other words, the sum of all $I_i \in S$ is a multiple of k.

Lemma 4.6 *Let C^* be the critical cycle in G, m be a positive integer, and $C_a, \ldots, C_b$ be the set of simple cycles in G such that $M(C_a) + \cdots + M(C_b) = m \times M(C^*)$. Then,*

$$m \times \Omega(C^*) > \sum_{C_i \in \{C_a, \ldots, C_b\}} \Omega(C_i).$$

Proof of Lemma 4.6

$$\begin{aligned} \sum_{C_i \in R} \Omega(C_i) &= \Omega(C_a) + \cdots + \Omega(C_b) \\ &= \alpha_a \times M(C_a) + \cdots + \alpha_b \times M(C_b) \\ &< \alpha^* \times M(C_a) + \cdots + \alpha^* \times M(C_b) \\ &= \alpha^* \times m \times M(C^*) \\ &= m \times \Omega(C^*) \end{aligned}$$

□

Theorem 4.3 *For any G with only one critical cycle C^* executed under the earliest firing rule and for $h \geq \mathcal{O}(n^3)$, the time constraint $X_j^{h+k} - X_j^h = p$ is obeyed by all $t_j \in G$, where $k = M(C^*)$ and $p = \Omega(C^*)$.*

Proof of Theorem 4.3

By Lemma 4.3, this theorem can be proven by showing that for $h \geq \mathcal{O}(n^3)$:

$$\max_{t_i} a_{h+k}(t_i, t_j) - \max_{t_i} a_h(t_i, t_j) = p, \forall t_j \in G$$

where t_i is a member of the set of initially enabled transitions at time zero. Or equivalently, we show that for $h \geq \mathcal{O}(n^3)$:

$$a_{h+k}(t_i, t_j) = a_h(t_i, t_j) + p, \forall t_j \in G$$

for all t_i in the set of initially enabled transitions at time zero.

Notice that $P_z(t_i, t_j)$, the set of paths from t_i to t_j with exactly z tokens, can be partitioned into three disjoint subsets: $P_z^a(t_i, t_j)$, $P_z^b(t_i, t_j)$, and $P_z^c(t_i, t_j)$, where $z > 0$. Subset $P_z^a(t_i, t_j)$ denotes the set of paths that iterate through C^* at least once. Subset $P_z^b(t_i, t_j)$ denotes the set of paths which only touch C^*, that is, C^* is not embedded entirely in the path, and subset $P_z^c(t_i, t_j)$ denotes paths that do not contain C^* at all.

We would like to show that the maximum value path in $P_h(t_i, t_j)$ for $h \geq \mathcal{O}(n^3)$ can only be found in subset $P_h^a(t_i, t_j)$. By Lemma 4.4 we know that the maximum value path in $P_h(t_i, t_j)$ for $h \geq \mathcal{O}(n^3)$ can never be found in subset $P_h^c(t_i, t_j)$. For every path in subset $P_h^b(t_i, t_j)$ there always exists a corresponding path in subset $P_h^a(t_i, t_j)$ which has a higher value sum, provided $h \geq (n+1)k + n$. For $h \geq (n+1)k + n$, there exists at least k cycles along any possible path in $P_h(t_i, t_j)$. By Lemma 4.5 there exists a subset S of those cycles C_i such that $\sum_{C_i \in S} M(C_i)$ is a multiple of k. Recall that $k = M(C^*)$. Assume $\sum_{C_i \in S} M(C_i) = m \times M(C^*), m \in$ integer, and $m > 0$ for any path $P_x \in P_h(t_i, t_j)$. Either S is composed of C^* m times, i.e., $P_x \in P_h^a(t_i, t_j)$; otherwise, P_x could never have the maximum path value. This is so because a path P_y can be constructed from P_x by replacing all $C_i \in S$ with exactly m C^*. P_y must also exist in $P_h(t_i, t_j)$, and by Lemma 4.6, it has a higher value sum. Therefore, the maximum value path of $P_h(t_i, t_j)$ for $h \geq \mathcal{O}(n^3)$ must be a member of subset $P_h^a(t_i, t_j)$.

In addition, notice that subset $P_{h+k}^a(t_i, t_j)$ can be constructed by having every path in subsets $P_h^a(t_i, t_j)$ and $P_h^b(t_i, t_j)$ iterate through C^* one more time. However, as was shown previously, subset $P_h^b(t_i, t_j)$ does not contain the maximum value path. Consequently,

$$\begin{aligned} a_{h+k}(t_i, t_j) &= a_{h+k}^a(t_i, t_j) \\ &= a_h^a(t_i, t_j) + \Omega(C^*) \\ &= a_h^a(t_i, t_j) + p \\ &= a_h(t_i, t_j) + p \end{aligned}$$

□

Theorem 4.3 states that all nodes in the loop (including both the nodes on or not on the critical cycles) will enter a periodic firing pattern after $\mathcal{O}(n^3)$ iterations. This suggests that we can "simulate" the loop execution at compile-time by constructing the behavior graph in $\mathcal{O}(n^3)$ iterations to reach this pattern. Since each iteration has $\mathcal{O}(n)$ firings in the "simulation" process, the actual number of time steps to reach the pattern is $\mathcal{O}(n^4)$, as shown by the following theorem.

Theorem 4.4 *Under the earliest firing rule, the cyclic frustum of G having one critical cycle can be found in $\mathcal{O}(n^4)$ time steps.*

Proof of Theorem 4.4

By Theorem 4.3, the time constraint $X_t^{h+k} - X_t^h = p$, $\forall t_i \in G$, $k = M(C^*)$, and $p = \Omega(C^*)$ is satisfied when $h \geq \mathcal{O}(n^3)$. In other words, the cyclic frustum appears after $\mathcal{O}(n^3)$ times of G are scheduled. Since G consists of n transitions, a total of $\mathcal{O}(n^4)$ firings will be performed. Note that $\mathcal{O}(n^4)$ firings can be done in at most $\mathcal{O}(n^4)$ time steps.

Since $p = \Omega(C^*)$ is the value sum of the critical cycle, it is bounded by n. Therefore, under the earliest firing rule, the cyclic frustum of G having one critical cycle emerges in $\mathcal{O}(n^4)+n$ time steps, or simply $\mathcal{O}(n^4)$. □

4.4.2 An SDSP-PN having Multiple Critical Cycles

As noted previously, the action of each transition is described by a global time constraint. It states that every k firings of a transition is p period apart, where k equals the least common multiple of the token sum of the critical cycles and p equals $\alpha \times k$. Since the time complexity for finding the steady state involves the determination of the length of the steady state, an upper bound for the least common multiple factor is thus required. We are unaware of any polynomial result and believe that the problem in this case remains open.

The rest of this section demonstrates that, for transitions residing on the critical cycles, a more concise *local* time constraint can be deduced which also coincides with the behavior description of the global time constraint.[4] In addition, the improved constraint is shown to be obeyed after $\mathcal{O}(n^2)$ iterations under the earliest firing rule. This result also applies to the case of single critical cycle. Let $C_1^*, C_2^*, \ldots$ denote the critical cycles. The new time constraint has the same form as the previous one

[4]The time constraint is local in the sense that it is only applicable to describe the behavior of the regarded transitions

except that in this case $k = M(C_i^*)$ and $p = \Omega(C_i^*)$ for all $t_j \in C_i^*$. This constraint justifies that transitions from different critical cycles have a different repeating period p, but they all keep the same computation rate $M(C_i^*)/\Omega(C_i^*)$. To demonstrate these claims, Lemmas 4.3 and 4.5 are used again. In the instance of multiple critical cycles, Lemma 4.6 is revised to Lemma 4.7. Note that the proof of the claim below, Theorem 4.5, is so similar to the one used for Theorem 4.3, in the last section, with only a slight difference.

Lemma 4.7 *Let C^* be a critical cycle in G, m be a positive integer, and $C_a, \ldots, C_b$ be the set of simple cycles in G such that $M(C_a) + \cdots + M(C_b) = m \times M(C^*)$. Then,*

$$m \times \Omega(C^*) \geq \sum_{C_i \in \{C_a, \ldots, C_b\}} \Omega(C_i).$$

Proof of Lemma 4.7

$$\begin{aligned}
\sum_{C_i \in R} \Omega(C_i) &= \Omega(C_a) + \cdots + \Omega(C_b) \\
&= \alpha_a \times M(C_a) + \cdots + \alpha_b \times M(C_b) \\
&\leq \alpha^* \times M(C_a) + \cdots + \alpha^* \times M(C_b) \\
&= \alpha^* \times m \times M(C^*) \\
&= m \times \Omega(C^*)
\end{aligned}$$

□

Theorem 4.5 *For any G executed under the earliest firing schedule and for $h \geq \mathcal{O}(n^2)$, the time constraint $X_j^{h+k} - X_j^h = p$ is obeyed by all $t_j \in C^*$, where C^* is a critical cycle, $k = M(C^*)$, and $p = \Omega(C^*)$.*

Proof of Theorem 4.5

With Lemma 4.3, this theorem is proven by showing that for $h \geq \mathcal{O}(n^2)$,

$$\max_{t_i} a_{h+k}(t_i, t_j) - \max_{t_i} a_h(t_i, t_j) = p, \ \forall t_j \in C^*$$

where t_i is a member of the set of initially enabled transition at time zero. Or equivalently, we show that for $h \geq \mathcal{O}(n^2)$,

$$a_{h+k}(t_i, t_j) = a_h(t_i, t_j) + p, \ \forall t_j \in C^*$$

for all t_i in the set of initially enabled transition at time zero.

Notice that $P_z(t_i, t_j)$, the set of paths from t_i to t_j with exactly z tokens, can be partitioned into two disjoint subsets $P_z^a(t_i, t_j)$ and $P_z^b(t_i, t_j)$, where $z > 0$. Subset $P_z^a(t_i, t_j)$ denotes the set of paths that iterate through C^* at least once, while subset $P_z^b(t_i, t_j)$ denotes the set of paths which only touch C^*, i.e., C^* is not embedded entirely in the path.

We show that for every path in subset $P_h^b(t_i, t_j)$ there always exists a corresponding path in subset $P_h^a(t_i, t_j)$ which has a greater or equal value sum, provided $h \geq (n+1)k + n$. Consequently, the maximum value path in $P_h(t_i, t_j)$ for $h \geq \mathcal{O}(n^2)$ can always be found in subset $P_h^a(t_i, t_j)$. For $h \geq (n+1)k + n$ there exists at least k cycles along any possible paths in $P_h(t_i, t_j)$. By Lemma 4.5 there exists a subset S of those cycles C_i such that $\sum_{C_i \in S} M(C_i)$ is a multiple of k, where $k = M(C^*)$. Assume $\sum_{C_i \in S} M(C_i) = m \times M(C^*), m \in$ integer, and $m > 0$ for any path $P_x \in P_h(t_i, t_j)$. Either S is composed of C^* m times (i.e., $P_x \in P_h^a(t_i, t_j)$) or P_x may not have the maximum path value. This is so because a path P_y can be constructed from P_x by replacing all $C_i \in S$ with exactly m C^*. P_y must also exist in $P_h(t_i, t_j)$, and by Lemma 4.7, it will have a greater or equal value sum. Therefore, the maximum value path of $P_h(t_i, t_j)$ for $h \geq \mathcal{O}(n^2)$ is always a member of subset $P_h^a(t_i, t_j)$.

In addition, notice that subset $P_{h+k}^a(t_i, t_j)$ can be constructed by having every path in the subsets $P_h^a(t_i, t_j)$ and $P_h^b(t_i, t_j)$ iterate through C^* one more time. However, as was shown, the maximum value path can always be found in subset $P_h^a(t_i, t_j)$. Consequently,

$$\begin{aligned} a_{h+k}(t_i, t_j) &= a_{h+k}^a(t_i, t_j) \\ &= a_h^a(t_i, t_j) + \Omega(C^*) \\ &= a_h^a(t_i, t_j) + p \\ &= a_h(t_i, t_j) + p \end{aligned}$$

□

Illustration of the Effect of Multiple Critical Cycles

To conclude the discussion of multiple critical cycles, we illustrate two sample code sequences to demonstrate the least-common-multiple effect of the token sum of critical cycles on the length of the steady state (see Figures 4.4 and 4.5). Figure 4.4 shows a code sequence made of one initial enabled node t_s (or a single source) and two critical cycles C_1 and C_2. The computation rates of C_1 and C_2 are 3/9 and 2/6. The value of the least common multiple k then equals 6 (3×2). Figure 4.5 displays a code sequence composed of three initial enabled nodes: t_{s1}, t_{s2}, and t_{s3} (multiple sources)

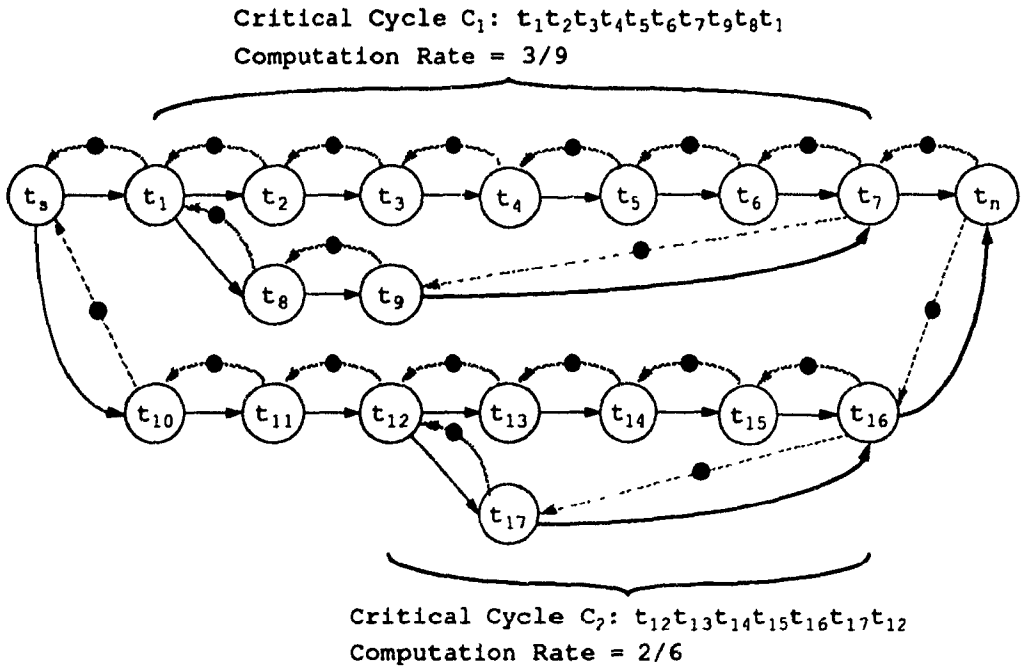


Figure 4.4: Code Sequence with Single Source

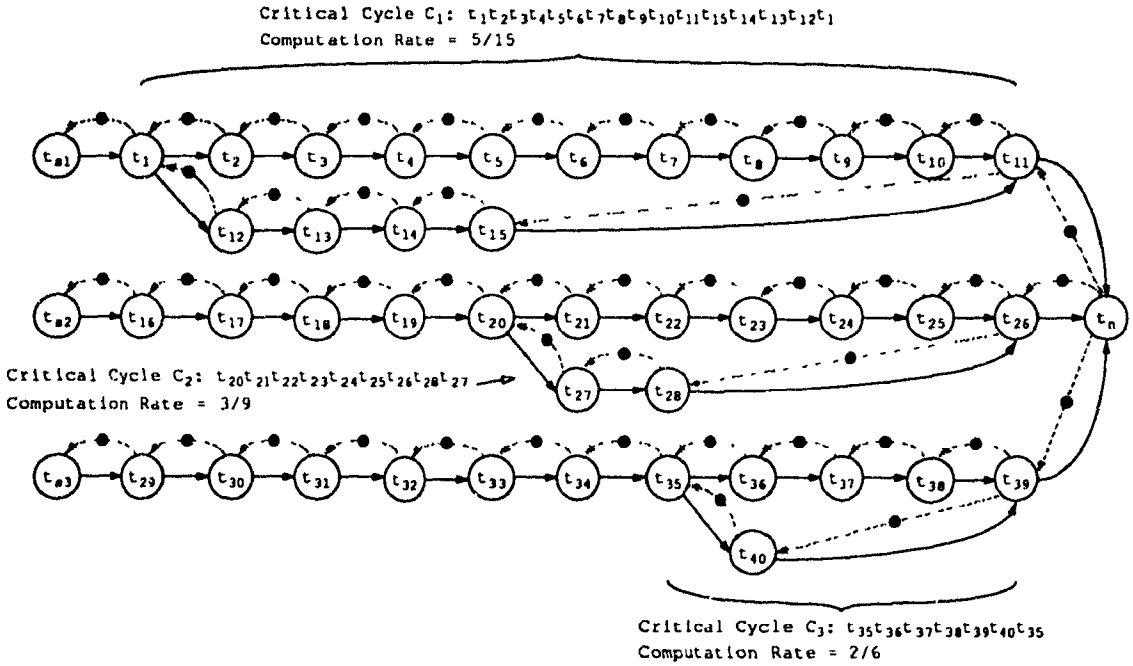


Figure 4.5: Code sequence with Multiple Sources

and three critical cycles: C_1, C_2, and C_3. The computation rates of C_1, C_2, and C_3 in this case equal 5/15, 3/9, and 2/6 respectively; the value of k equals 30 ($2 \times 3 \times 5$).

The corresponding results for the two sample codes under earliest firing rule are depicted on Tables 4.1 and 4.2. Among the results, the *number of firings* indicates the number of firings of a node in a repeated firing sequence, by which we can determine if the local time constraint is obeyed. The *time delay* indicates the delay between consecutive firings in the corresponding repeated firing sequence; the total sum of the time delays thus equals the repeating period of the particular node. The *iteration* indicates the latest iteration instance observed when the steady state is reached. Here are the major observations:

- Nodes t_{11} and t_n in the single-source code sequence and node t_n in the multiple-source code sequence fire respectively six times and thirty times in their steady states, confirming the effect of critical cycles under the global time constraint.

- The number of firings for node on the critical cycle C_j in both code sequences equals $M(C_j)$, the number of tokens on the critical cycle. These incidents verifies the existence of a local time constraint claimed by Theorem 4.5.

- The steady state of the off critical-cycle nodes are not necessary influenced by the least common multiple factor. In the single source case, only nodes t_{11} and t_n demonstrate the impact of critical cycles while nodes t_s and t_{10} obey the local time constraint.

For more examples of the influence of critical cycles, the multiple-sources case presents an extensible platform. For example, we can raise the value of k from $2 \times 3 \times 5$ to $2 \times 3 \times 5 \times 7$ by attaching a new branch of nodes which consists of a new critical cycle C_4 with a rate of 7/21 and a new source node t_{s4}. The resulting graph then consists of four branches corresponding to four critical cycles. We then adjust the other three branches by inserting new nodes until the three old branches regain the same height, i.e., the distance of each branch from the source nodes to t_n equals the distance from t_{s4} to t_n.

4.4.3 Tightness of the Bound

In this section we illustrate the tightness of the derived polynomial upper bound, using the example in Figure 4.6. The example illustrates the need for initiating at least $n-1$ iterations before the repetitive firing pattern is reached. It contains a chain of n nodes with only one critical cycle ($t_{n-2}t_{n-1}t_nt_{n-2}$) located at the right end. The

Table 4.1: Single Source

	Range of Nodes					
	t_s	t_1–t_9	t_{10}	t_{11}	t_{12}–t_{17}	t_n
Label	*Source Node*	C_1			C_2	
Number of Firings k	3	3	3	6	2	6
Time Delays	2,3,4	2,3,4	2,3,4	2,4,2,4,3,3	2,4	2,4,2,4,3,3

† Code Size: 19, Iteration: 14

Table 4.2: Multiple Sources

	Range of Nodes								
	t_{s1}	t_{s2}	t_{s3}	t_1–t_{15}	t_{16}–t_{19}	t_{20}–t_{28}	t_{29}–t_{34}	t_{35}–t_{40}	t_n
Label	*Source Nodes*			C_1		C_2		C_3	
Number of Firings k	5	3	2	5	3	3	2	2	30

† Code Size: 44, Iteration. 43

computation rate of the critical cycle and the chain, in general, is 1/3. All other simple cycles have a computation rate of 1/2. In addition, note that there is a total of $n-2$ tokens along the path from t_n to t_1. Initially at time zero, t_1 is the only initially enabled node. By Lemma 4.3, the time for t_1 to commence its $h+1$ firing can be computed by $a_h(t_1, t_1)$, the maximum path value among the set of possible paths from t_1 to t_1 with exactly h tokens. However, due to the chain of $n-2$ tokens from t_n to t_1, the set of paths from t_1 to t_1 with less than $n-2$ tokens can never reach the critical cycle. Thus, it indicates that t_1 is required to initiate at least $n-1$ times (i.e., $n-1$ iterations, or $\mathcal{O}(n)$ iterations) before the effect of the critical cycle can be propagated back to t_1.

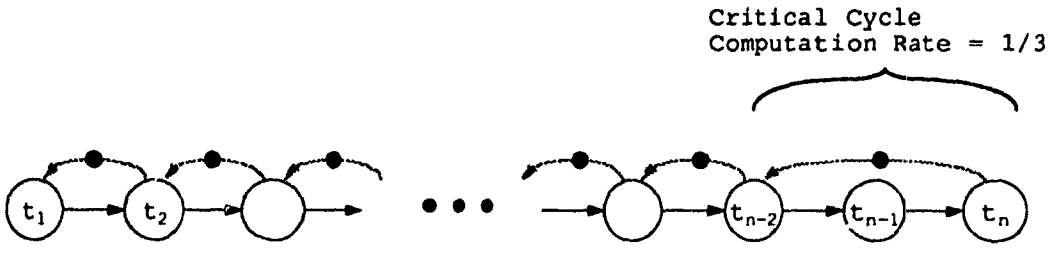


Figure 4.6: A Code Sequence with an $\mathcal{O}(n)$ Lower Bound

4.5 Initial Token-Distribution Constraint

Through our simulations (Table 5.1), we have found surprisingly short initial periods for all benchmarks tested. In fact, the observed bound was within $\mathcal{O}(n)$ iterations. It appears that the bound derived above is too pessimistic, and a tighter bound is possible. In Section 4.5.1 we give a marking condition, obtained by generalizing Theorem 4.5, with a tighter initial period. In Section 4.5.2 we report a significantly improved bound for the same marking found by using a different approach.

4.5.1 A Tighter Initial Period

The Initial token-distribution depicted by Theorem 4.6 characterizes an *initial marking* of G such that a repeated pattern can be found after $\mathcal{O}(n^2)$ iterations regardless of the number of critical cycles. The length of steady state and the number of firings of each transition in steady state are respectively $\Omega(C^*)$ and $M(C^*)$, where C^* is the critical cycle initially holding all enabled transitions. Formally, the time constraints $X_i^{h+k} - X_i^h = p, \forall t_i \in G$ are satisfied after $\mathcal{O}(n^2)$ iterations of G, where $k = M(C^*)$ and $p = \Omega(C^*)$. The validity of Theorem 4.6 is important because the required initial condition can always be reached after at most $\mathcal{O}(n)$ iterations, as discussed in the next paragraph. Consequently, the repetitive firing pattern for a general SDSP-PN G can be found after $\mathcal{O}(n^2)$ iterations of G, regardless of the number of critical cycles.

Token Distribution Constraint Satisfaction: Assume that transition t_i resides on a critical cycle. To meet the initial condition, one simply executes G using the earliest firing rule but prohibits any firing of transition t_i. The firing process soon deadlocks. Since G is strongly connected, there always exists a cycle-free path P from t_i to t_j for all t_j in G. If t_i is never fired, t_j stops firing soon after all tokens along P have been consumed. Note also that there can be at most n tokens along a cycle-free path; that is, t_j can be fired at most n times before the initial condition is met. Equivalently, it requires the scheduling of at most $\mathcal{O}(n)$ iterations of G to reach the required state.

Theorem 4.6 *Under the earliest firing rule, if the set of initial enabled transitions at time 0 all belong to a selected critical cycle C^* in G, $\forall t_j \in G$ the time constraints $X_j^{h+k} - X_j^h = p$ is obeyed for $h \geq \mathcal{O}(n^2)$, where $k = M(C^*)$ and $p = \Omega(C^*)$.*

Proof of Theorem 4.6

Assume that the only enabled transitions at time zero are those on the selected critical cycle C^*. With Lemma 4.3, we prove the theorem by showing that for $h \geq$

$\mathcal{O}(n^2)$,

$$\max_{t_i} a_{h+k}(t_i, t_j) - \max_{t_i} a_h(t_i, t_j) = p, \ \forall t_j \in G$$

where t_i is a member of the set of initially enabled transitions at time zero. Equivalently, we show that for $h \geq \mathcal{O}(n^2)$,

$$a_{h+k}(t_i, t_j) = a_h(t_i, t_j) + p, \ \forall t_j \in G$$

for all t_i in the set of initially enabled transitions at time zero.

Notice that $P_z(t_i, t_j)$, the set of paths from t_i to t_j with exactly z tokens, can be partitioned into two disjoint subsets $P_z^a(t_i, t_j)$ and $P_z^b(t_i, t_j)$, where $z > 0$. Subset $P_z^a(t_i, t_j)$ denotes the set of paths that iterate through C^* at least once, while subset $P_z^b(t_i, t_j)$ denotes the set of paths which only touch C^* (i.e., C^* is not embedded entirely within the path).

We show that for every path in subset $P_h^b(t_i, t_j)$ there always exists a corresponding path in subset $P_h^a(t_i, t_j)$ which has an equal or greater value sum, provided $h \geq (n+1)k + n$, where $k = M(C^*)$. Consequently, the maximum value path in $P_h(t_i, t_j)$ for $h \geq \mathcal{O}(n^2)$ can always be found in subset $P_h^a(t_i, t_j)$. For $h \geq (n+1)k + n$, there exists at least k cycles along any possible path in $P_h(t_i, t_j)$. By Lemma 4.5 there exists a subset S of those cycles C_i such that $\sum_{C_i \in S} M(C_i)$ is a multiple of k. Assume $\sum_{C_i \in S} M(C_i) = m \times M(C^*), m \in$ integer, and $m > 0$ for any path $P_x \in P_h(t_i, t_j)$. Either S is composed of C^* m times (i.e., $P_x \in P_h^a(t_i, t_j)$) or P_x may not have the maximum path value. This is because a path P_y can be constructed from P_x by replacing all $C_i \in S$ with exactly m C^*. P_y must also exist in $P_h(t_i, t_j)$, and by Lemma 4.7, it has a greater or equal value sum. Therefore, the maximum value path of $P_h(t_i, t_j)$ for $h \geq \mathcal{O}(n^2)$ is always a member of subset $P_h^a(t_i, t_j)$.

In addition, notice that subset $P_{h+k}^a(t_i, t_j)$ can be constructed by having every path in the subsets $P_h^a(t_i, t_j)$ and $P_h^b(t_i, t_j)$ iterate through C^* one more time. However, as was shown, the maximum value path can always be found in subset $P_h^a(t_i, t_j)$. Consequently,

$$\begin{aligned} a_{h+k}(t_i, t_j) &= a_{h+k}^a(t_i, t_j) \\ &= a_h^a(t_i, t_j) + \Omega(C^*) \\ &= a_h^a(t_i, t_j) + p \\ &= a_h(t_i, t_j) + p \end{aligned}$$

□

As can be seen, to meet the initial token-distribution constraint, the search for a transition on the critical cycle is significant. One possible approach to find such a

transition is to first determine the computation rate restricted by the critical cycle using Magott's linear programming formulation [Mag84] and then apply the shortest path algorithm with the distance formulation defined by Ramamoorthy and Ho to obtain a critical cycle [RH80]. Using the critical cycle, the required transition can be selected arbitrary. Since Magott's formulation can be solved by linear programming within a theoretical polynomial bound and the shortest path problem can be solved in $\mathcal{O}(n^3)$ steps, the final problem of determining repetitive pattern is polynomial bound also.

An alternative approach to the problem is to appoint a transition the initial enable condition and then construct the behavior graph from then on for n^2+n iterations. If the repetitive execution pattern cannot be found, an untouched transition is selected and the procedure is repeated. In the process a maximum of n transitions will be checked, and at most n iterations are required to satisfy the initial token-distribution constraint for a selected transition. The time complexity of the approach is bounded by the time required to schedule $n(n+n^2+n)$ iterations, i.e., $\mathcal{O}(n^4)$ iterations. Note that this algorithm suggests a totally different way of approaching the problem of determining the computation rate for G.

Though we have established a theoretical solution to the steady-state problem with respect to the earliest firing rule, the platform upon which all the proofs are based (Lemma 4.5), however, seems too general and fails to utilize the safeness properties of the SDSP-PN. Hence, the theoretical bound is loose compared with our simulation results. In the next section we report a significantly improved bound on the length of the initial period for the same token-distribution constraint, using a different approach.

4.5.2 A Second Approach for a Tighter Initial Period

The results described in this section show that the steady state is reachable after k iteration of G, where k is the number of tokens of the critical cycle which holds the initial enabled transition t_s. Rather than relying on the previously introduced maximum-value-path framework, the following proof makes use of theory developed for valid-schedule-computation scheme (see Lemma 3.1) [RH80, Rei68] Precisely, we make use of the relation between the earliest firing schedule X_i^h and the valid schedule S_i^h:

$$X_i^h \leq S_i^h, \quad \forall t_i \in G$$

Our proof consists of two parts: The first part shows that the 1 firing, $k+1$ firing, $2k+1$ firing, ..., $mk+1$ firing of a transition are p-period apart using the valid-schedule-computation framework. The second part of the proof generalizes the result in part

one to h firing and $h+k$ firing are p-period apart using the safeness property of the model.

Theorem 4.7 *Under the earliest firing rule, if the only enabled transition at time zero belongs to C^*, a critical cycle in G, $\forall t_j \in G$ the time constraint $X_j^{h+k} - X_j^h = p$ is obeyed for $h \geq 0$, where $k = M(C^*)$ and $p = \Omega(C^*)$.*

Proof of Theorem 4.7

Let $k = M(C^*)$, $p = \Omega(C^*)$, and t_s be the enabled transition at time zero residing on C^*. First we show that for $m \geq 0$ the $mk+1+k$ firing and the $mk+1$ firing of t_s are p-period apart, i.e.,

$$X_s^{mk+k} - X_s^{mk} = p, \ m \geq 0. \tag{4.7}$$

Observe that with t_s being the only enabled transition in G at time zero, the valid firing time S_i^0 for transition t_i to commence the first firing also equals the earliest firing time X_i^0, according to Lemma 3.1, i.e.,

$$X_i^0 = S_i^0, \ \forall t_i \in G. \tag{4.8}$$

Since t_s resides on C^*, every k-th firing of t_s cannot be shorter than p, i.e.,

$$X_s^{mk+k} - X_s^{mk} \geq p, \ m \geq 0. \tag{4.9}$$

By Lemma 3.1, the valid firing time for the $mk+1$ firing of t_s is computed by:

$$\begin{aligned} S_s^{mk} &= a_s + mp \\ &= mp. \end{aligned}$$

That is, the corresponding valid firing time for t_s to commence the 1 firing, $k+1$ firing, $2k+1$ firing, ..., $mk+1$ firing are $0, p, 2p, \ldots, mp$, according to Lemma 3.1. According to Equations 4.8 and 4.9 it is the earliest firing schedule. Thereby, $X_s^{mk} = S_s^{mk}$ for $m \geq 0$. Hence, Equation 4.7 is proven.

Next we show that for $m \geq 0$ and $\forall t_i \in G$ the $mk+1+k$ firing and the $mk+1$ firing of t_i is p-period apart, i.e.,

$$X_i^{mk+k} - X_i^{mk} = \Omega(C^*), \ m \geq 0 \text{ and } \forall t_i \in G. \tag{4.10}$$

After constraint satisfaction, for each $t_i \in G \neq t_s$ there exists a place $p \in {}\cdot t_i$ with no tokens on it; otherwise, t_i is an enabled transition eligible for firing. Accordingly, there always exists at least one token-free path from t_s to t_i. Let P_i denote the one

which gives X_i^0 ($= S_i^0$) the longest token-free path from t_s to t_i (Lemma 3.1). Note that the amount of time t_s takes to generate and deliver the mk+1 token to t_i along P_i cannot be shorter than a period of X_i^0+mp, i.e.,

$$X_i^{mk} - X_i^0 \geq mp, \ m \geq 0 \text{ and } \forall t_i \in G. \tag{4.11}$$

By Lemma 3.1, the valid firing time for the mk+1 firing of t_i is computed by:

$$\begin{aligned} S_i^{mk} &= a_i + mp \\ &= S_i^0 + mp \\ &= X_i^0 + mp. \end{aligned}$$

According to Lemma 3.1, the corresponding valid firing time for t_i to commence the 1 firing, k+1 firing, 2k+1 firing, ..., mk+1 firing are X_i^0, X_i^0+p, X_i^0+2p, ..., X_i^0+mp. Equation 4.11 confirms that it is the earliest firing schedule. Thereby, $X_i^{mk} = S_i^{mk}$ for $m \geq 0$ and $\forall t_i \in G$. Equation 4.10 is validated. Note that when $k = M(C^*) = 1$, the cyclic frustum is obviously reached after the first iteration

Finally, we are ready to show that $X_i^{h+k} - X_i^h = p$ for $h \geq 0$ and $\forall t_i \in G$. Since G is safe, for $h \geq 1$, X_i^h of transition t_i equals

$$\max_{p_{ji} \in \cdot t_i} \begin{cases} X_j^{h-1} + \Omega(t_j), & |p_{ji}| = 1 \\ X_j^h + \Omega(t_j), & |p_{ji}| = 0 \end{cases} \tag{4.12}$$

For $m \geq 0$, the subsequent earliest firing schedule of transition t_i after each mk+1 firing (i.e., the schedule staring from the mk+2 firing) merely starts on tokens produced by the mk+1 firing or the mk+2 firing of some transitions. Thus, by Equation 4.10 and 4.12, the same earliest firing schedule of t_i repeats after the mk+1 firing for $m \geq 0$. Consequently, $X_i^{h+k} - X_i^h = p$ for $h \geq 0$ and $\forall t_i \in G$. This result implies that a repeated pattern can be found after k iterations of G. Equivalently, $\mathcal{O}(n)$ iterations is required because k is bounded by n in a safe marked graph. □

4.6 Remarks

Note that the requirement for an acknowledgement arc for each data arc and the resulting safeness property are both characteristics of the static dataflow model. To keep the concept of an ideal machine, we have assumed a unit firing time for each transition. In general, however, the proofs presented in Sections 4.4 and 4.5.1 can be extended to cover a more general class of strongly-connected marked graphs where the one-acknowledgement-arc-per-data-arc restriction is eliminated, the individual transition is assigned a different firing time, and the number of tokens residing at a place

are more than one (but bounded). Accordingly, a larger polynomial may be obtained. The proofs presented in Section 4.5.2 can also be extended to a more general class of strongly-connected *safe* marked graphs where the one-acknowledgement-arc-per-data-arc restriction is eliminated, and an individual transition is assigned a different firing time. However, the assumption of having a self-loop on each transition (Assumption 3.1) is required in all cases. Without the self-loop control, the relation between the earliest firing schedule and the maximum path value (Lemma 4.3) cannot be established.

Chapter 5

Software Pipeline Scheduling with Pipeline Constraint

In this section we study the application of SPS for loop scheduling on processor architectures having a number of clean execution pipelines. The current architecture under consideration assumes the existence of multiple clean pipelines that are identical. It serves to represent a series of high-performance computer architectures such as pipelined machines and very long instruction word (VLIW) architectures

In Section 5.1 we introduce a single clean pipeline (SCP) model SDSP-SCP-PN. We then incorporate the ideas of multiple clean pipelines and produce SDSP-MCP-PN in Section 5.2. Based upon the two models, we explore the concept of the behavior graph and the existence of steady state to ensure the feasibility of deriving a static schedule for a machine with multiple clean pipelines. In Section 5.3 we examine the amount of time required to find the steady state on a set of Livermore loops. The fast detection of steady state shown in the results indicates the feasibility of practical compilers using a behavior graph to generate a static schedule Finally, in Section 5.4 we discuss a related work.

5.1 Model with a Single Clean Pipeline—SDSP-SCP-PN

In this section we describe a unified timed Petri-net model SDSP-SCP-PN for fine-grain loop scheduling having resource constraints. The unified model is constructed by incorporating a clean hardware pipeline of l stages into the SDSP-PN model. The main property of a clean pipeline is that an instruction moves through the pipeline in l cycles once it enters, without interference from other instructions. This implies that

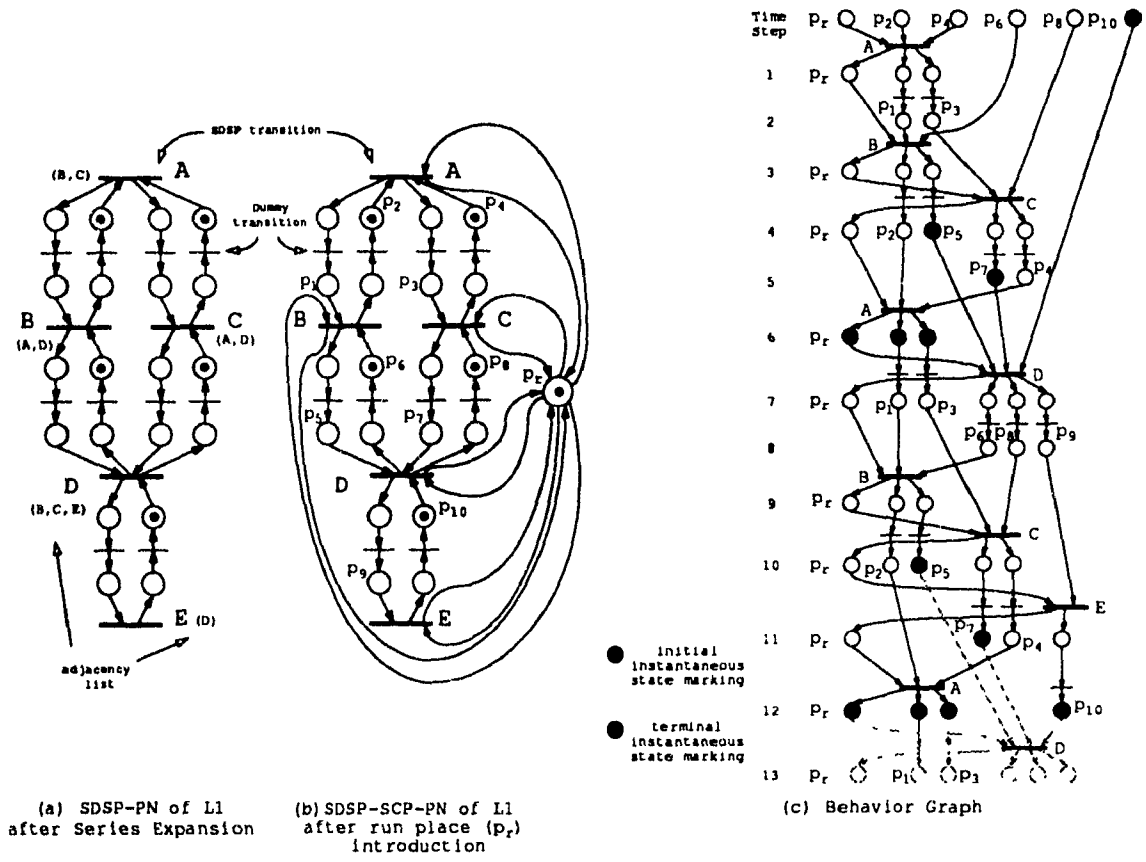


Figure 5.1: SDSP-SCP-PN and the Behavior Graph

the detailed structure of an SCP need not be explicit. The construction of the single pipeline model consists of two steps: *series expansion* and *run-place introduction.*

Run-place introduction: We introduce a place p_r, known as the *run place*, to denote the SCP and modify all transitions t_i in the SDSP-PN to include p_r as both the input and output places. Place p_r is initially marked with a token representing the existence of one SCP. When a transition becomes enabled, it competes for p_r to get fired.

Series expansion: To denote the fact that one traversal through SCP takes l time units, a series expansion is performed which introduces a new transition for each place in the SDSP-PN; this accounts for the time delay. We call the transitions originally appearing in the SDSP-PN the *SDSP transitions* and the ones newly introduced in series expansion the *dummy transitions.* Every SDSP transition is assigned an execution time of 1 while every dummy transition is assigned an execution time of $l-1$, where l denotes the length of the execution pipeline. When l=1 there are no dummy transitions remaining. In Figure 5.1

we distinguish dummy transitions by bars of a different length.

Figure 5.1(a) illustrates the outcome of L1 after series expansion. Figure 5.1(b) shows the result of introducing a run place.

Theorem 5.1 *An SDSP-SCP-PN with an initial marking M_0' is live, safe, and persistent if the SDSP-PN with an initial marking M_0 is live, safe, and persistent.*

Proof of Theorem 5.1

The application of series expansion on a marked graph preserves the liveness and safeness properties in the resulting marked graph [Mur80, MK80]. In addition, the introduction of the run place initially containing a token still preserves persistency since the firing of an enabled transition does not disable other enabled transitions. Although we have introduced a structural conflict, which leads to the possibility of choices, the resulting Petri net still preserves the liveness, safeness and persistency properties of the original marked graph. □

Using Theorem 5.1 we construct the behavior graph for the combined model in a manner similar to the one constructing for an SDSP-PN. With the existence of the run place as a structural conflict, choices appear whenever more than one SDSP transition is enabled. To resolve the choices, we make the following assumption:

Assumption 5.1 *The firing mechanism in the SCP machine always chooses one enabled node to fire—it never idles as long as there is at least one enabled node. The machine can break ties by giving priority to the nodes that simultaneously become enabled. The priority does not matter. We assume only that the machine exhibits repeatable behavior, i.e., it always makes the same choice given its priority rule and machine condition (instantaneous state).*

This assumption provides a means for making the transition process of the instantaneous state unique, and the behavior graph will be unique as long as a particular choice scheme is enforced. We can achieve uniqueness by resolving conflicts in two ways:

- One way is to use a mathematical function to compute the next SDSP transition to fire at every time steps based upon the current set of enabled transitions.
- Another way is to employ an internal decision mechanism as a finite state machine. Newly enabled SDSP transitions are then fed into the mechanism sequentially as input.

The major difference in the two approaches regards the involvement of the internal state in the latter case. As for the former approach, the instantaneous state defined by the marking and residual firing time vector is sufficient and precise. However, with the introduction of a decision mechanism as a part in the SDSP-SCP-PN model, the content of the instantaneous state is required to also incorporate the state of the decision mechanism. For the function approach to conflict resolution, all SDSP transitions can be initially assigned a unique priority value, so a function which always picks the one with the highest priority can be used. Note that it will not cause starving because transitions cannot continue firing infinitely without firing others (in a live-safe Petri net).

Figure 5.1(c) illustrates a possible behavior graph derived from the example in Figure 5.1(b). In this particular case *choice resolution* is done by a decision mechanism which employs a *FIFO queue* and an *adjacency list representation* of the static data-flow graph.[1] Notice that there can only be one instruction coming out of the execution pipe at every pipe beat under the single pipeline architecture. Upon completion of executing a node (i.e., a SDSP transition and its associated dummy transitions), all adjacent nodes (SDSP transitions) will be signaled. Enabled ones are then ordered sequentially onto a FIFO queue according to order in the adjacency list. The choice resolution at any time step is then done by honoring the one at the head of the FIFO queue. In this case, the instantaneous state is made up of the marking, the residual firing time vector, and the state of the FIFO queue.

Similar to the behavior graph of an SDSP-PN, the behavior graph of an SDSP-SCP-PN also exhibits a repetitive behavior. This behavior is described by Lemma 5.1 in conjunction with Assumption 5.1.

Lemma 5.1 *There exists an instantaneous state in the behavior graph of an SDSP-SCP-PN which appears repeatedly.*

Proof of Lemma 5.1

The total number of distinct M_i must be finite because SDSP-SCP-PN has a safe marking. Similarly, the total number of distinct R_i is also finite because each transition in SDSP-SCP-PN has a known firing time. If the behavior graph is infinitely extended, some instantaneous state must be repeated. □

Once the machine returns to a previous instantaneous state the same firing sequence is repeated. As an example, the two sets of highlighted markings shown in

[1] Adjacency lists are a common representation for directed graphs. Node j is said to be adjacent to node i in a directed graph G if the directed arc (i, j) exists in G. The adjacency list for node i is a list, in some order, of all nodes adjacent to i.

Figure 5.1(c) illustrate the marking portion of the initial and terminal instantaneous states. Their associated residual firing time vectors are zero vectors. The firing sequence in the steady state is $ADBCE$. As illustrated, the notion of steady state can be defined by enforcing Assumption 5.1. Similarly, we can define the concept of steady-state equivalent net as the SDSP-PN counterpart. Thus, SPS can be applied equally well on a machine with pipelined constraints by adhering to this assumption. Note also that the assumption degenerates to the earliest firing rule (used in the ideal case) as enough pipelines are available.

With the existence of a resource constraint, imposed by the single pipeline, the computation rate of an SDSP-SCP-PN is no longer reflected directly in the critical cycle. The impact of resource constraints is illustrated with Theorem 5.2. It ultimately imposes an upper bound on the execution rate of each node in the SDSP-SCP-PN. Intuitively, there can be at most one enabled transition for execution during each time step. Thus, it takes at least n cycles to complete one iteration of the loop body even though the cycle time of the critical cycles is far less. Note also that this bound is the result of the constraint imposed by the pipeline and is independent of the approach used for conflict resolution. When such a bound is reached, all pipelines are 100% utilized.

Theorem 5.2 *Let G be an SDSP-SCP-PN with n SDSP transitions. The computation rate of any SDSP transition in G can never be greater than $1/n$, i.e., $\gamma \leq 1/n$.*

Proof of Theorem 5.2

To prove this theorem, it is sufficient to show that there exists a simple cycle in the resulting steady-state equivalent net such that the computation rate for any SDSP transition is $1/n$.

- Let M be the first repeated instantaneous state marking used to form the steady-state equivalent net. Let σ be the cyclic firing sequence such that $M \xrightarrow{\sigma} M$. By Lemma 3.5, each SDSP transition of G in σ occurs an equal number of times.

- Note that none of the SDSP transitions can be fired in parallel because there is only one token in the run place. In addition, any appearance of the SDSP transition in the steady state must be chained together by different instances of the run place since the run place forms a self-loop with each one of the SDSP transitions. Consequently, there exists a simple cycle C' in the resulting steady-state equivalent net containing all of the occurrences of the SDSP transitions.

- Note also that each of the n SDSP transitions is assigned a timing of one time cycle. As a result, $\Omega(C) = a \times n$, where a is the number of times that each SDSP transition occurs in C. The computation rate of any SDSP transition is thus $\frac{a}{a \times n}$, or $\frac{1}{n}$.

□

5.2 Multiple Clean Pipelines—SDSP-MCP-PN

In this section we extend the single clean pipeline model to the case of multiple clean pipelines (MCP), producing a unified Petri-net model SDSP-MCP-PN. The SDSP-MCP-PN models the execution of the SDSP on machine with multiple clean execution pipelines each of l stages. The previous development of the single pipeline model provides an extensible platform upon which the multiple clean pipelines model can be established. Instead of assigning only one token at the step of run place introduction, as many tokens as the number of pipelines are modeled are assigned so that each token denotes a distinct execution pipeline. Figure 5.2(a) shows a model of two pipelines.

The behavior graph for the combined model can be constructed in a way similar to that of the SDSP-SCP-PN described previously. To deal with the problem of choices resulting from structural conflicts, we again assume a firing mechanism which always chooses the particular enabled nodes to fire—it never idles as long as there is at least one enabled node. The machine breaks ties by giving priority to nodes that simultaneously become enabled. The priority does not matter; we assume only that the machine exhibits repeatable behavior, i.e., it always makes the same choice given its priority rule and machine condition (the instantaneous state).

Multiple tokens in a run place can be represented in the behavior graph as multiple instance of the run places, as shown in Figure 5.2(b). In addition, the assumption above implies that a repeatable instantaneous state is encountered if the behavior graph is extended for a sufficient period of time. The notion of cyclic frustum is again used to derive a repetitive schedule for a multiple clean pipeline machine.

Similar to the single pipeline case, the constraint of multiple pipelines imposes an upper bound on the computation rate of each transition in the SDSP-MCP-PN. For a model of R pipelines, there can be at most R enabled transitions sent for execution at each time step. Thus, it takes at least n/R cycles to compute one iteration of the loop body even though the cycle time of the critical cycles is far less.

Theorem 5.3 *Given an SDSP-MCP-PN G which models R clean pipelines and contains n SDSP transitions, the computation rate of any SDSP transition in G can*

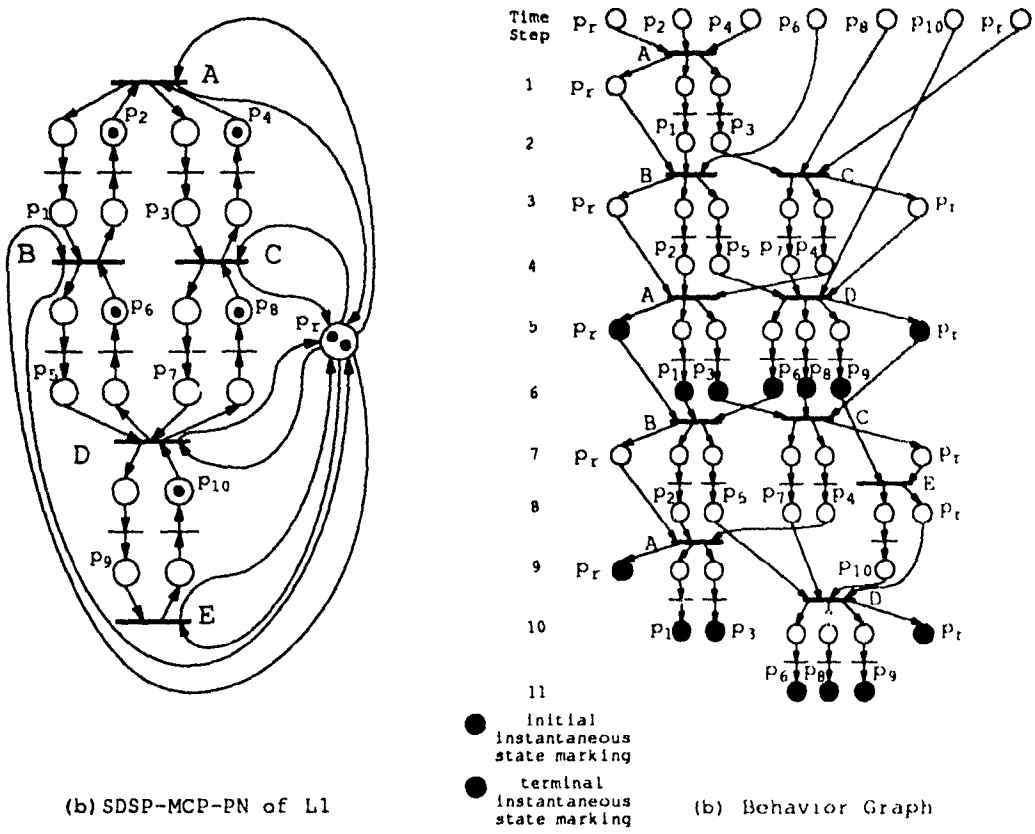


Figure 5.2: SDSP-MCP-PN and the Behavior Graph

never be greater than R/n, i.e., $\gamma \leq R/n$.

5.3 Simulation Results

A set of Livermore Loops was chosen for the study; all were written in SISAL [Feo88, Mea85], and simulations performed on these loops using a compiler/simulator testbed developed at McGill University [GP90]. The testbed consists of a prototype SISAL compiler capable of producing dataflow code, known as A-Code [Tio88a, Tio88b]. For this study, we modified the simulator to permit analysis of cyclic frustums generated for both SDSP-PN and SDSP-MCP-PN models. The simulator takes A-code as input and simulates the corresponding firing sequence.

Table 5.1 shows the results of executing an SDSP on an ideal machine with infinitely many single-stage pipelines. Equivalently, the SDSP-PN was executed under the earliest firing rule with the firing time of each transition equal to one. In the table, the *size* reflects the number of nodes in a loop body that were repeatedly executed

Table 5.1: Results for SDSP-PN Model

	Loops without LCD				Loops with LCD			
	Loop 1	Loop 7	Loop 9[2]	Loop 12	Loop 3	Loop 5	Loop 9	Loop 11
Size	31	64	84	15	15	18	84	14
Iteration	13	7	14	8	5	6	2	7
StartT	33	19	26	16	8	9	14	16
RepeatT	36	23	30	19	10	13	28	20
FrustumLen	3	4	4	3	2	4	14	4
TCount	1	1	1	1	1	1	1	1
CompRate	1/3	1/4	1/4	1/3	1/2	1/4	1/14	1/4

excluding the start-up initiation sequence. *StartT* and *RepeatT* (Start Time and Repeat Time) indicate the times when the initial and the terminal instantaneous states are identified. *Iteration* indicates the number of iterations initiated up to repeat time. *FrustumLen* (Frustum Length) is the difference between repeat time and start time. *TCount* (Transition Count) records the number of occurrences of a transition that appears in the cyclic frustum. Note that all transitions are fired an equal number of times in the cyclic frustum (Theorem 3.5). *CompRate* (Computation Rate) is the average firing rate of each SDSP transition in the loop body and equals

$$\frac{\text{TCount}}{\text{RepeatT} - \text{StartT}}$$

Note that in each example the repeated instantaneous state is found within $\mathcal{O}(n)$ iterations.

Table 5.2 shows the corresponding pattern for the set of benchmarks using one, two, four, and eight clean pipeline(s), respectively. For the case of one pipeline, we include *utilization*, which gives the processor usage as a basis for discussion. The results of this experiment demonstrate that the steady state in all instances can be found efficiently. It also reveals the following facts:

- The condition raised by Theorem 5.2 is verified in the case of one clean pipeline where some test programs keep the single pipeline fully busy. In Loop 1, Loop 7, and Loop 9, the upper bound on the computation rate, $1/n$, imposed by the single pipeline is reached. All three cases indicated that the respective pipelines were fully utilized at all time except when they were filling and draining. The various processor usage in the three loops also reflects the impact of the prelude and postlude execution sequence. Though the length of the postlude sequence

[2]Loop 9 is a potential candidate for parallelizing as a DOALL loop; however, it requires subscript analysis to expose its parallelism. Here we examined the loop both ways, with and without LCDs, to increase the diversity of our testing.

Table 5.2: Results for SDSP-MCP-PN Model with Eight Stages

	Loops without LCD				Loops with LCD			
	Loop 1	Loop 7	Loop 9	Loop 12	Loop 3	Loop 5	Loop 9	Loop 11
$2n \times l$ (BD)	480	1024	1344	240	240	288	1344	224
1 Pipeline:								
Iteration	13	8	14	9	5	6	2	7
StartT	341	296	749	157	86	86	132	138
RepeatT	372	360	833	184	106	121	263	172
FrustumLen	31	64	84	27	20	35	131	34
TCount	1	1	1	1	1	1	1	1
CompRate	1/31	1/64	1/84	1/27	1/20	1/35	1/131	1/34
Utilization	98.9%	99.7%	98.4%	55 8%	72 9%	51 0%	64 1%	40 9%
2 Pipelines:								
Iterations	13	8	14	9	5	7	2	7
StartT	282	206	395	147	82	78	115	133
RepeatT	308	244	438	172	100	145	230	166
FrustumLen	26	38	43	25	18	67	115	33
TCount	1	1	1	1	1	2	1	1
CompRate	1/26	1/38	1/43	1/25	1/18	2/67	1/115	1/33
4 Pipelines:								
Iteration	13	9	14	10	6	7	2	8
StartT	268	207	242	145	81	74	113	154
RepeatT	293	240	277	194	98	139	226	186
FrustumLen	25	33	35	49	17	65	113	32
TCount	1	1	1	2	1	2	1	1
CompRate	1/25	1/33	1/35	2/49	1/17	2/65	1/113	1/32
8 Pipelines:								
Iteration	15	8	16	8	5	6	2	7
StartT	265	174	223	128	64	72	112	128
RepeatT	338	206	323	152	80	104	224	160
FrustumLen	73	32	100	24	16	32	112	32
TCount	3	1	3	1	1	1	1	1
CompRate	3/73	1/32	3/100	1/24	1/16	1/32	1/112	1/32

was not recorded, the shorter prelude sequence in Loop 7, indicated by start time, was obviously a factor accounting for the higher utilization rate.

- Though each transition was fired an equal number of times in the steady state of a marked graph, the number of firings was not necessarily one.

- In general, the amount of time required for the emergence of steady state decreased as the number of pipelines increased, except in a few cases where the value of the transition count was different from the ideal model.

- As the number of pipelines exceeded the amount of parallelism in the loop, the behavior graph obtained was exactly the same as the one obtained for the ideal model. For instance, as Loop 12, Loop 3, Loop 5, Loop 9 with LCD, and Loop 11 were run with eight pipelines, their start time and repeat time were simply eight times the corresponding time derived for the ideal model.

- The number of the iterations initiated to reach steady state for all cases were still less than n, the size of the loop body. Hence, the steady state was reached efficiently. In addition, the counted number of iterations for the ideal model gave a close approximation of the number of iterations required by all of the multiple-pipelines models studied.

5.4 Discussion

To construct a schedule for the multiple pipeline machines, Aiken and Nicolau suggested using the same schedule obtained from the ideal case by scheduling the steady state one row at a time [AN88]. It was also shown when such schedule is adopted for the multiple pipelines machine, the total run time is always bounded by two times the optimal run time obtained for the same machine [NPA88]. Nevertheless, the resulting schedule is still unsatisfactory because, after all instructions from row i are scheduled for execution, a period of $l-1$ idle cycles (where l is the length of the pipeline) is always required to delay the initiation of row $i+1$, in order to avoid possible data conflict between the last operation of row i and the first operation of row $i+1$. Consider the use of the steady state of L1, shown in Figure 4.2, as the schedule for a machine with two clean pipelines and each one having two stages. The part of the schedule which involves the steady state will be

processor1	A	noop	B	E	noop
processor2	D	noop	C	noop	noop

At each iteration, A and D cannot be sent for execution until all B, C, and E complete firing, even though transition A is free for execution right after B and C complete their firing. Similarly, since Ramamoorthy and Ho's schedule is derived on a marked graph which is equivalent to an ideal machine model, it incurs the same inefficiency when it is applied to the multiple clean pipeline case.

In the SDSP-MCP-PN model, the problem of data conflict in a multiple pipeline was considered in the process of constructing the behavior graph. While imposing the earliest firing rule, the gap required in the former case is filled with enabled instructions that are safe to be executed. The corresponding schedule which involves the steady state, derived from the behavior graph (Figure 5.2(b)), is given below.

processor1	B	E	A	D
processor2	C	noop	noop	noop

Thus this scheme will always render better processor usage. In addition, the assurance of a repeatable state in the SDSP-MCP-PN, together with the simulation results obtained so far, reveals the feasibility of employing the behavior graph to generate a static schedule in practical compilers.

As a final remark, note that the pipelined models presented are general enough to allow the existence of multiple function units within each individual pipeline. Note that for each SDSP transition the regarded dummy transitions serve to account for the time delay for a particular pipeline. Hence, the assignment of the different time delay implies the use of a different function pipeline.

Chapter 6

Storage Allocation

Memory requests can slow down a computation considerably. If they occur during the steady state of a pipelined schedule the computation cannot proceed efficiently. The use of registers as temporary storage to reduce memory accesses is important to maintain the steady-state computation rate and processor throughput. In this chapter we discuss the application of a program restructuring scheme known as *limited balancing* to reduce the amount of storage requirement in SPS, making the use of fast memory feasible.

The objective behind limited balancing is to expose in a software pipeline only the amount of parallelism that is exploitable by the machine. This is accomplished by restructuring a static dataflow graph, prior to pipeline scheduling, according to a *balancing ratio*—a parameter which characterizes the achievable computation rate of the final schedule. The utilization of execution units is not affected since only excessive parallelism is suppressed. During restructuring, storage requirements are systematically reduced across the loop body. The one-token-per-arc policy of the static dataflow graph originally needs one unit of storage per data arc; limited balancing reduces the storage requirement below this level.

In Section 6.1 we introduce two memory models for a static dataflow graph due to two different architecture designs: *argument-flow* and *argument-fetch*. The argument-fetch model describes how conventional data fetching and storing can be implemented using a static dataflow graph. In Section 6.2 we introduce the notion of limited balancing, using the SDSP-PN model, and we discuss its application to the two memory models. Since the computation rate of a critical cycle in an SDSP-PN determines the computation rate of an entire net, it naturally represents the balancing ratio of the model. In Sections 6.3 and 6.4 we derive a guideline to estimate a balancing ratio for a machine with a pipeline constraint. We then validate the guideline through experimental results.

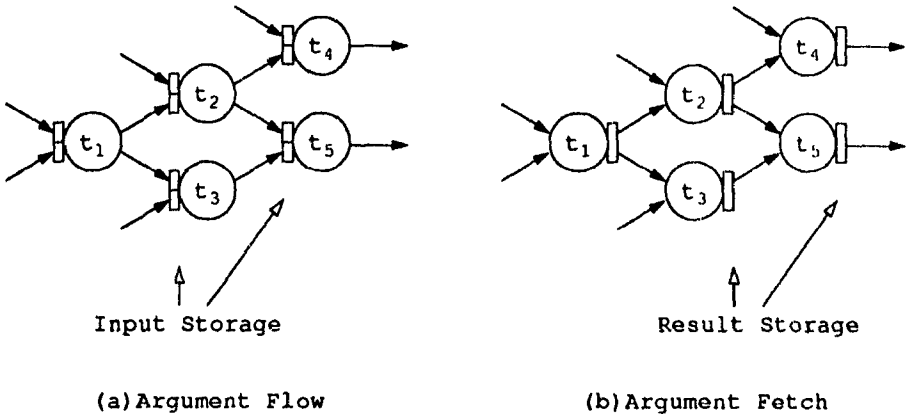


Figure 6.1: Storage Usage of Argument-flow Model versus Argument-Fetch Model

6.1 Memory Model

Thus far our discussion of static dataflow graphs have been based upon an abstract notion of *data flow*, i.e., the notion that tokens flow from a source node to its receivers. In this section two memory models are presented: *argument-flow* and *argument-fetch* [Den84, DG88]. The argument-fetch model describes how conventional data fetching and storing can be implemented using a static dataflow graph.

Since, in the original development of dataflow model, data is viewed as flowing from a source node to destination nodes, the model is called *argument-flow*. Figure 6.1(a) presents an abstract view of storage organization for conventional static dataflow. Storage for input data is local to each node. Accordingly, the result is required to be replicated and dispatched separately to each destination node. The arrival of an input token serves two purposes: it signals the receiver of the availability of data (a control role), and it transmits the data value (a data role). In early designs of the architecture, data packets composed of a data portion and a destination-address portion were used [Den84]. Unfortunately, duplication of data for multiple destinations causes unnecessary data traffic. Such inefficiency results from binding control information and data information within the same packet.

The *argument-fetch* dataflow model overcomes the argument-flow model's inefficiency. The key difference in the two models is the separation of data and control information. After each computation only a control packet is sent to acknowledge the availability of data. This packet is called a *signal*. The result of each computation either remains in a register or is returned to memory close to the execution pipeline where it can be easily fetched by successor nodes (see Figure 6.1(b)) The

major improvement of argument-fetch is a significant reduction of data traffic; the number of data storage is also considerably reduced. For a detail discussion of the argument-fetch dataflow model, see [DG88].

The abstract model of argument-fetch is depicted by a directed graph. However, the interconnection of *signal* arcs in this model merely represents sequencing information based upon data dependence. The abstract graph is called a *signal graph*, and each node in the graph represents an operation, as before. Acknowledgement arcs still serve to preserve the one-token-per-arc principle. Upon node execution, signaling performs two functions: Signals along the signal arcs notify successors of the availability of results, and signals along acknowledgement arcs inform predecessors of the consumption of their output and are ready for new inputs. Figures 1.1(c) and 2.5(c) could be a signal graph if all arcs were treated as signal arcs.

The development of the argument-fetch dataflow model at McGill has led to the construction of an abstract machine code known as A-code. A-code is an example of a signal graph. To take advantage of software pipelining, new control constructs, such as an index generator (IGEN), are introduced in A-code. Appendix A contains an example of the A-code for Loop 3 of the Livermore loops, and it briefly describes the A-code structure. For a detailed discussion of A-code and design decisions related to argument-fetch, see [GT89, GTHS8, GP90, Par88, Par90].

6.2 Limited Balancing of an SDSP-PN

We start our description of limited balancing by introducing the concept using the SDSP-PN model. Based upon this ideal model, we will establish a connection between balancing ratio and computation rate, and we will demonstrate a savings of data storage and synchronization both for the *argument-flow* and for the *argument-fetch* graph models. Our discussion begins, however, with an ideal model in which resource savings seem unnecessary. In practical situation one often encounters the situation in which sufficient resources (execution pipelines) are available (cf. the ideal model). In these cases we minimize the data storage required while sustaining maximum computation.

Limited balancing evolved from the notion that the attainable computation rate of a static dataflow graph was constrained by critical cycles [GHW90a, GHW90b]. Recall for an SDSP-PN PN, the computation rate γ is determined by the cycle time of the critical cycle in the net:

$$\gamma = \min\left\{\frac{M(C_k)}{\Omega(C_k)}, \frac{1}{\Omega(t_i)}\right\}, \text{ for all simple cycles } C_k \text{ in } PN$$

This relationship suggests an opportunity to modify the structure of other non-critical simple cycles without suffering any loss of speed, provided the computation rate of all altered cycles is larger than or equal to the computation rate of the critical cycles.

Previous studies on *balancing* were only carried out using an ideal machine model, where the intention was to exploit maximum fine-grain parallelism; *full* balancing on an acyclic graph was then the main focus [Gao89]. The consideration of loops with loop-carried dependence leads to limited balancing [GHW90a]. If a critical cycle is composed entirely of data arcs (either forward or backward), the computation rate cannot be altered without modifying input program. In this case the computation rate of a critical cycle forces a *hard* upper bound on the achievable computation rate of the graph. One immediate problem is to determine the minimum amount of storage necessary to maintain the computation rate imposed by the critical cycle. For loops without loop-carried dependence, limited balancing also plays an essential role. In some cases the parallelism of a program is higher than that the machine is able to exploit, but maintaining excess parallelism wastes machine resources.

For our definition of the balancing ratio we adopt an ideal model, one which assumes that all operations require the same amount of time to execute, denoted by l [Gao89]. Note that this time constraint can be relaxed. To facilitate our discussion, we define a balancing ratio based upon dataflow graph representation. If G is the original static dataflow software pipeline, the formula for calculating the computation rate of G is:

$$\gamma = \min\left\{\frac{M(C_k)}{N_k \times l}\right\}, \text{ for all simple cycles } C_k \text{ in } G,$$

where N_k is the number of nodes in cycle C_k, and $M(C_k)$ is the number of tokens within cycle C_k. For each C_k the ratio $\frac{M(C_k)}{N_k}$ is called the *balancing ratio*. Accordingly, critical cycles always have the smallest balancing ratio and slowest computation rate. We now demonstrate the application of limited balancing using both the argument-flow and the argument-fetch computation models.

Recall that in the argument-flow model each forward/feedback data arc corresponds to one unit of storage; an associated acknowledgement arc controls its usage. A token on an acknowledgement arc denotes the vacancy of storage, while a token on a forward/feedback arc denotes occupancy.[1] For the argument-flow model, a reduction of balancing ratio on non-critical cycles suggests a way of dealing with both the storage-usage problem and the synchronization-cost problem (caused by the flow of the acknowledgement signals). In general the reassignment of acknowledgement arcs in a graph can be a means of decreasing the balancing ratio. The insertion of

[1]The token and space duality in the SDSP is actually the same as the duality in Kung's augmented dataflow graph [KLL86].

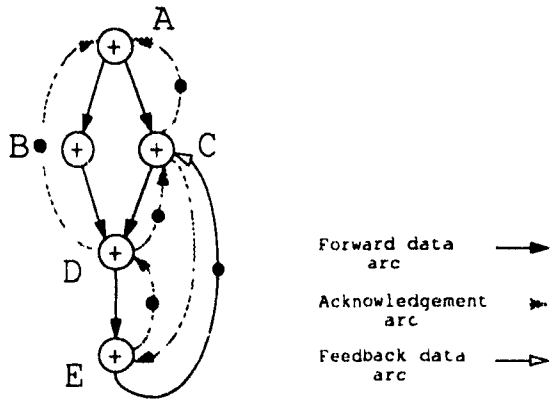


Figure 6.2: Minimum Storage Allocation

dummy nodes as buffers will increase the balancing ratio [Gao89]. For instance, L2 consists of a critical cycle C^* $(CDEC)$ consisting entirely of data arcs, a situation which imposes a hard upper bound on the computation rate of the graph, $1/3l$ (see Figure 2.5). On the other hand, simple cycles C_1 (ABA) and C_2 (BDB) possess a larger balancing ratio, $1/2$, and allow the opportunity to alter the balancing ratio by reassigning acknowledgement arcs. Figure 6.2 illustrates the consequences of limited balancing by acknowledgement-arc rearrangement. Now the new cycle C_3 $(ABDA)$ has a balancing ratio of $1/3$. The immediate saving in signal traffic is obvious. Note also that acknowledgement arc (D, A) controls the usage of the input storage for nodes B and D, creating an opportunity for nodes B and D to reuse the same space as their input storage. More importantly, all of these treatments can be carried out without sacrificing execution speed.

For the argument-fetch model, the signal graph is the target of limited balancing. Similar methods such as acknowledgement-arc rearrangement and buffer insertion can be employed, and similar savings on signal traffic achieved. To illustrate how storage is reduced in this case, we present a simple example in which a set of nodes safely reuses output space. Figure 6.3 shows the A-code for Loop 9 of the Livermore Loops. The loop body of Loop 9 consists of one critical cycle (node72, node73, node77 node74, node72). The maximum computation rate is $1/4l$, and hence the suggested balancing ratio is $1/4$. Figure 6.4 shows a possible limited-balancing of Loop 9 (with a balancing ratio of $1/4$). As can be seen, plenty of simple cycles are found after arc rearrangement. The unique serial formation of nodes in a cycle not only assures the order of activation of each node but, also provides a safe situation for the nodes to reuse the same output space. For instance, nodes 30, 31, and 32 in the simple cycle located at the left edge of the graph can reuse the same output storage.

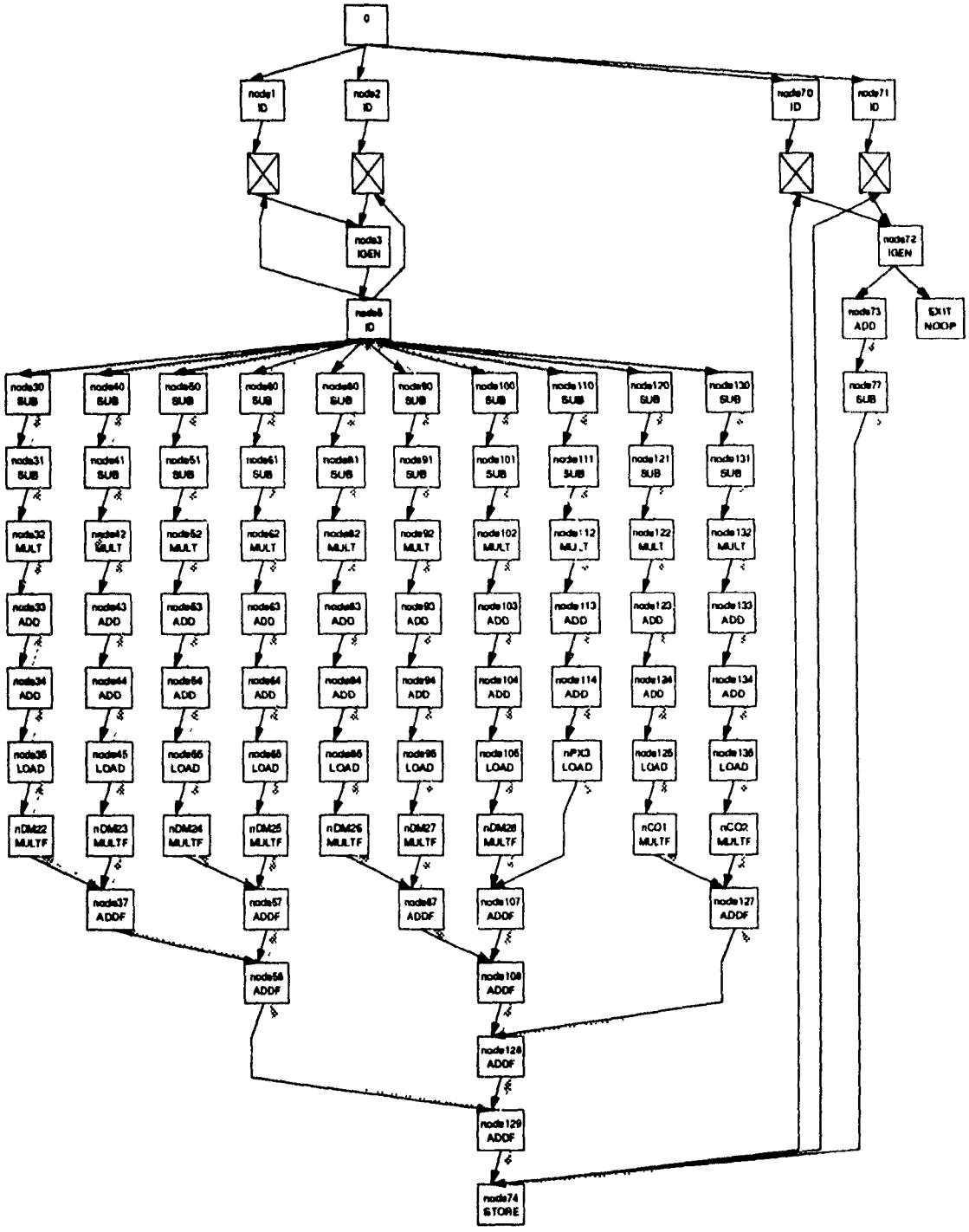


Figure 6.3: A-code for Loop 9

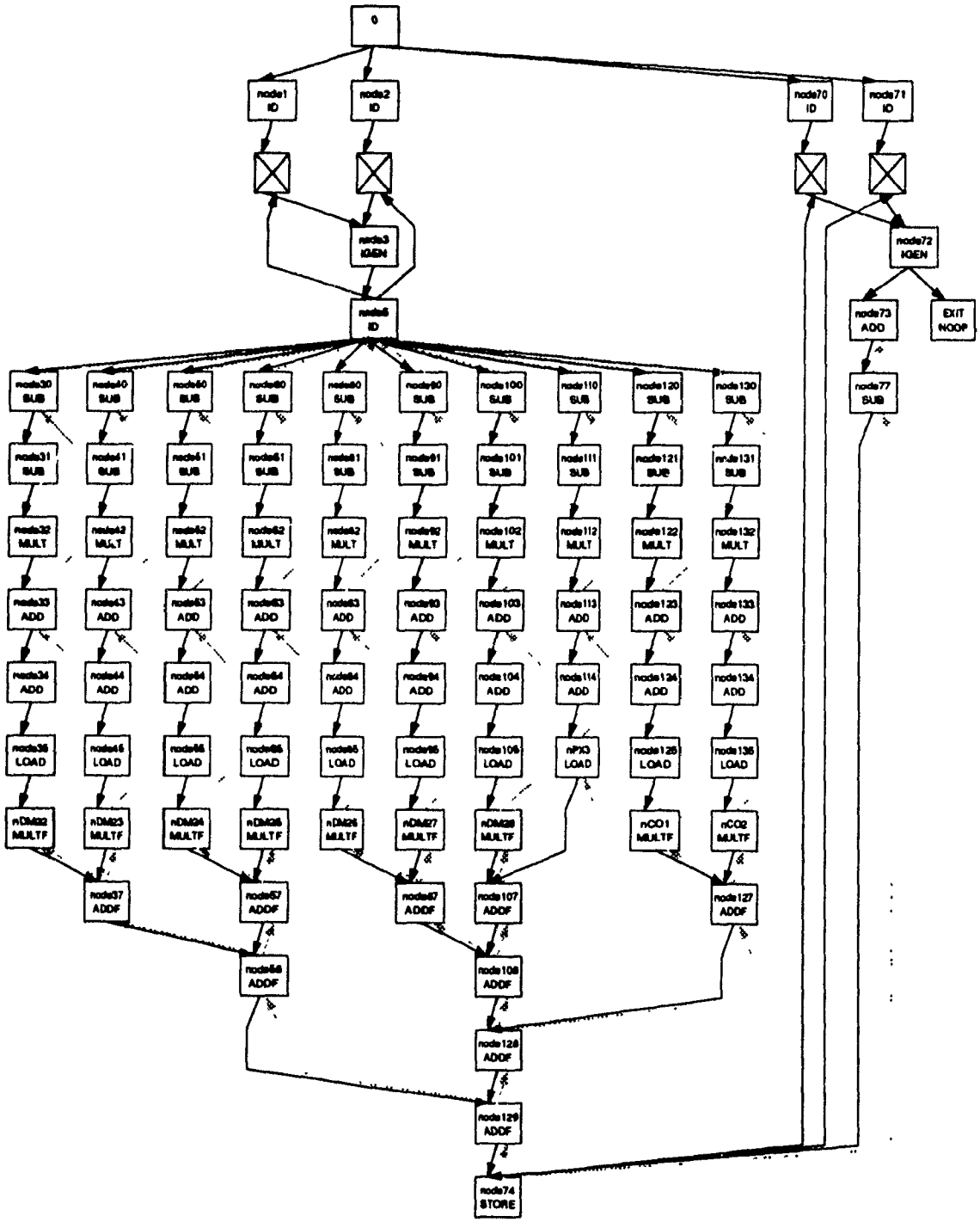


Figure 6.4: Loop 9 under Partial Limited Balancing

6.3 Limited Balancing of an SDSP-SCP-PN

For the ideal model the balancing ratio provides a means of addressing, within a unified framework, both the scheduling problem and the storage-allocation problem. There are two important characteristics in the previous discussion. First, the balancing ratio can be computed precisely by determining the critical cycle. Second, limited balancing of a graph with the same balancing ratio as its critical cycle does not alter the computation rate. Unfortunately these two characteristics do not hold for the SDSP-SCP-PN model. Though a larger balancing ratio will likely increase the amount of parallelism, the computation rate of a single clean pipeline with n instructions can never exceed $1/n$. Therefore, the balancing ratio of a graph alone cannot determine the computation rate for the SDSP-SCP-PN model. Rather than seeking a definite relation between balancing ratio and computation rate, as in the case of the ideal model, we establish a balancing ratio and utilization rate relationship. An approximation is derived to estimate an adequate balancing ratio for the loop body such that a certain utilization rate of the clean pipeline is maintained under the SPS approach. Once the balancing ratio is determined, limited balancing can be applied throughout the graph to reduce storage.

To estimate a correct and appropriate balancing ratio under SPS, we compute a balancing ratio based upon a more conservative scheduling scheme B, where B is inferior to SPS in terms of longer running time. Accordingly, the computed balancing ratio for scheme B always provides a conservative measure of the target ratio required for SPS and can be safely used. The chosen scheme B is called a *naive* scheduling scheme (see Chapter 5.4) [AN88]. Based upon the steady state obtained from the ideal case, scheme B schedules one row at a time the repetitive pattern on a machine with one execution pipeline. This scheme has an important characteristic, it has predictable scheduling behavior so that the total execution time can be easily computed. To estimate the required balancing ratio for scheme B, the relationship between balancing ratio and utilization rate is established using the fundamental pipeline utilization-rate formula shown below:

$$\text{Utilization rate } U \;=\; 1 - \frac{\text{Idle time}}{\text{Total execution time}} \tag{6.1}$$

where the idle time and the total execution time are the corresponding time expected for scheme B. In addition, we will use the following notations in our analysis:

- G, n, and l will denote respectively an SDSP, the total number of instructions in G, and the length of the execution pipeline

- F will denote the steady state for the associated SDSP-PN of G, i.e., the cyclic frustum obtained from the ideal case. I will denote the number of times F is iterated at execution, and $J_1, J_2, \ldots, J_p$ will denote respectively the p rows in F; $N(J_i)$ will denote the number of transitions fired at row i of F.

- k will denote the total number of occurrences of a transition in F. Recall that all transitions appear the same number of times in F (by Theorem 3.5). For the ideal case, $\frac{k}{p}$ is then the computation rate as well as $B(G)$, the current balancing ratio of G.

Listed below are two terms used to compute an upper bound for the idle time expected on scheme B. Equation 6.2 shows the total execution time required to execute F, I times on a single pipeline machine using scheme B, while Equation 6.3 shows the minimal execution time required. The idle time can thus be obtained by subtracting Equation 6.3 from Equation 6.2.

- **Total execution time:**

$$\begin{aligned} & I \times (N(J_1) + l - 1 + N(J_2) + l - 1 + \cdots + N(J_p) + l - 1) \\ = \; & I \times (kn + pl - p) \end{aligned} \tag{6.2}$$

- **Minimal execution time:**

$$Ikn + l - 1 \tag{6.3}$$

We are now ready to establish the relationship between balancing ratio and utilization rate for scheme B, using the pipeline utilization-rate formula. By substitution of Equations 6.2 and 6.3 into Equation 6.1, we have

$$\begin{aligned} 1 - U &= \frac{\text{Eqn. 6.2} - \text{Eqn. 6.3}}{\text{Eqn. 6.2}} \\ 1 - U &= 1 - \frac{\text{Eqn. 6.3}}{\text{Eqn. 6.2}} \\ U &= \frac{\text{Eqn. 6.3}}{\text{Eqn. 6.2}} \\ U &= \frac{Ikn + l - 1}{I \times (kn + pl - p)} \end{aligned} \tag{6.4}$$

The resulting Equation 6.4 can be evaluated in various ways:

- By approximating Equation 6.4 for U one can determine the guaranteed utilization rate U of the execution pipeline for a given loop body G of n instructions

having a balancing ratio $B(G)$.

$$\begin{aligned} U &\geq \frac{Ikn}{I \times (kn + pl - p)}, \quad l \geq 1 \\ &= \frac{kn}{p \times (\frac{kn}{p} + l - 1)} \\ &= \frac{nB(G)}{nB(G) + l - 1} \end{aligned} \tag{6.5}$$

- By solving Equation 6.5 for $B(G)$ one can determine for a given loop body G of n instruction a sufficient balancing ratio $B(G)$ such that a pipeline utilization rate U is maintained.

$$B(G) \leq \frac{U(l-1)}{n(1-U)}$$

For loops with no loop-carried dependence, if the balancing ratio cannot be improved due to restricted parallelism in a loop body, one might be able to unroll the loop body to increase parallelism of a software pipeline as described by the following guideline.

- Finally, by approximating Equation 6.4 for n, one can determine for a given balancing ratio $B(G)$ the required number of instructions n in a loop body G such that U utilization of the execution pipeline is maintained.

$$\begin{aligned} n &= \frac{(UIp - 1) \times (l - 1)}{(1 - U) \times Ik} \\ &< \frac{UIpl}{(1 - U) \times Ik} \\ &= \frac{Upl}{(1 - U) \times k} \end{aligned}$$

For loops with no loop-carried dependence, if the loop body does not match the size n, a new loop body can be obtained by unrolling until the number of instructions matches the requirement.

The establishment of the guidelines based upon naive scheduling provides the conservative estimations for the corresponding parameters in SPS. For example, the estimation of $B(G)$ can be safely used as a conservative estimate for the balancing ratio used in limited balancing for SPS. Note that these guidelines can only be applied in cases where the execution time of all nodes is the same.

6.4 Limited Balancing of an SDSP-MCP-PN

For multiple-clean pipelines we explore a similar set of relational guidelines. As in the previous section, naive scheduling is assumed as the basis for deriving an upper bound on the execution time of multiple pipelines. Naive scheduling schedules one row at a time from the repetitive schedule under the ideal case, while each row is divided equally among all available execution pipelines. With the aid of the utilization-rate formula, relation between balancing ratio and utilization rate is again established. We use R to denote the number of execution pipelines available. Equation 6.6 shows an upper bound on the execution time required by naive scheduling, and Equation 6.7 shows the expression for the best achievable time when a hundred percent utilization of all pipelines is achieved.

- **Total Execution Time**

$$\begin{aligned} & I \times \left(\left\lceil \frac{N(J_1)}{R} \right\rceil + l - 1 + \left\lceil \frac{N(J_2)}{R} \right\rceil + l - 1 + \cdots + \left\lceil \frac{N(J_p)}{R} \right\rceil + l - 1\right) \\ < \; & I \times \left(\frac{N(J_1)}{R} + l + \frac{N(J_2)}{R} + l + \cdots + \frac{N(J_p)}{R} + l\right) \\ = \; & Ipl + \frac{Ikn}{R} \end{aligned} \tag{6.6}$$

- **Lower bound on Execution Time**

$$\frac{Ikn}{R} + l - 1 \tag{6.7}$$

Equation 6.8 shows the result of substituting the minimal execution time and the required execution time expressions (Equations 6.6 and 6.7) into the utilization rate formula (Equation 6.1).

$$\begin{aligned} 1 - U \; & < \; \frac{\text{Eqn. 6.6} - \text{Eqn. 6.7}}{\text{Eqn. 6.6}} \\ 1 - U \; & < \; 1 - \frac{\text{Eqn. 6.7}}{\text{Eqn. 6.6}} \\ U \; & > \; \frac{\text{Eqn. 6.7}}{\text{Eqn. 6.6}} \\ & = \; \frac{\frac{Ikn}{R} + l - 1}{Ipl + \frac{Ikn}{R}} \end{aligned} \tag{6.8}$$

- Equation 6.9 shows the result of reducing Equation 6.8 further. For the given program factors, $B(G)$ and n, and the hardware factors, l and R, Equation 6.9 imposes a lower bound of the utilization rate of all R execution pipelines.

$$\begin{aligned} U &> \frac{\frac{lkn}{R} + l - 1}{lpl + \frac{lkn}{R}} \\ &\geq \frac{\frac{lkn}{R}}{lpl + \frac{lkn}{R}}, \ l \geq 1 \\ &= \frac{lkn}{pRl(l + \frac{kn}{pR})} \\ &= \frac{B(G)n}{Rl + B(G)n} \end{aligned} \tag{6.9}$$

- By solving Equation 6.9 for $B(G)$, Equation 6.10 imposes an upper bound on the balancing ratio to keep a utilization rate of U on all R execution pipelines for the loop G consisting of n instructions.

$$B(G) < \frac{URl}{n(1-U)} \tag{6.10}$$

- Finally, by solving Equation 6.8 for n, one can determine for a given balancing ratio $B(G)$ the required number of instructions n in a loop body such that a utilization rate of U for all R execution pipelines is maintained. For loops without loop-carried dependence, if the loop body does not match the size n, a new loop body can be obtained by unrolling until the number of instructions matches the requirement.

$$\begin{aligned} UIpl + \frac{UIkn}{R} &> \frac{Ikn}{R} + l - 1 \\ UIpl - l + 1 &> \frac{Ikn(1-U)}{R} \\ n &< \frac{R(UIpl - l + 1)}{Ik(1-U)} \\ &\leq \frac{RUIpl}{Ik(1-U)}, \ l \geq 1 \\ &= \frac{RUB(G)l}{1-U} \end{aligned} \tag{6.11}$$

6.5 Experimental Results

To substantiate the correctness of our guideline, we provide simulation results of pipeline utilization conducted on Loop 9 of the Livermore Loops. Figure 6.3 shows the

signal graph of Loop 9. The loop body of Loop 9 consists of 84 nodes and one critical cycle (node72, node73, node77 node74, node72). The maximum computation rate is $1/41$. In the experiment we balanced Loop 9 using 1/4 and compared the observed utilization rate against the estimated lower bound. Table 6.1 provides estimations and results under various machine configurations. *PL* and *Number of Stages* indicate the number of pipelines and their associated length used at the simulation. Under each configuration the estimated utilization rate is computed using Equations 6.5 and 6.9, and is listed under the row *Estimate*. To appreciate the impact of limited balancing, we examine Loop 9 under three levels of limited balancing, indicated in the table as *Before*, *Phase1*, and *Phase2*.

Before: Original Loop 9 contains one critical cycle, i.e., no limited balancing.

Phase1: Limited balancing is partially applied across Loop 9, as shown in Figure 6.4.

Phase2: Limited balancing is applied more aggressively across Loop 9, as shown in Figure 6.5.

For each of the three cases, recorded results includes computation rate for steady state execution (on the right) and pipeline utilization (on the left). The following summarizes our observations:

- All of the estimated pipeline utilization rates are a correct lower bound for observed processor usage, confirming our guideline.

- The accuracy of the approximation increases under two extremes; the first extreme is the result of two behavioral factors: The first factor is the increase in the number of execution pipelines. In this case the behavior graphs of the program respectively under SPS and under naive scheduling both converge towards the behavior graph under ideal model. As the number of execution pipelines exceeds the amount of exploited parallelism, the schedules of SPS and naive scheduling are the same as the one produced for the ideal case. The second factor is the increase in the number of stages in the execution pipeline. The amount of parallelism tends to decrease as pipeline length increases. Thus, a longer pipeline decreases the utilization rate over the same program. As both factors are influencing the result, the utilization rate decreases rapidly towards the estimated bound. The other extreme is a decrease in the number of pipelines and pipeline length. As both the number of pipelines and the number of stages drop to a small value, the calculated lower bound approaches 100% utilization.

Table 6.1: Results for Utilization Rate Estimation

		Number of Stages							
		4		8		16		32	
PL		Rate	Util	Rate	Util	Rate	Util.	Rate	Util
1	Estimate		87.5%		75%		56 8%		40 4%
	Before	1/84	99.4%	1/84	98.4%	1/86	93.8%	1/141	58 0%
	Phase1	1/84	99.8%	1/84	99.3%	1/85	96 9%	1/131	62 3%
	Phase2	1/84	99.8%	1/84	99.3%	1/87	94 7%	1/142	57 7%
2	Estimate		72.4%		56.8%		39 6%		24 7%
	Before	1/42	98.4%	1/43	93.9%	1/70	58 4%	1/134	30 2%
	Phase1	1/42	99 3%	1/42	98.0%	1/65	62.8%	1/129	31 6%
	Phase2	1/42	99 3%	1/43	95.7%	1/71	57 7%	1/135	30 3%
4	Estimate		56 8%		39 6%		24 7%		14 1%
	Before	2/43	93 9%	1/35	58.4%	1/67	30 6%	1/131	15 6%
	Phase1	1/21	98.0%	1/32	63 7%	1/64	31 9%	1/128	15 9%
	Phase2	2/43	95 7%	1/35	58 4%	1/67	30 5%	1/131	15 6%
8	Estimate		39.6%		24 7%		14 1%		7 6%
	Before	3/52	58 9%	3/100	30.7%	3/196	15 6%	3/388	7 9%
	Phase1	1/16	63.6%	1/32	31 9%	1/64	15.9%	1/128	8 0%
	Phase2	1/17	60 0%	1/33	30 9%	1/65	15.7%	1/129	7 9%
16	Estimate		24 7%		14 1%		7 6%		3 9%
	Before	3/50	30 7%	3/98	15.0%	3/194	7 9%	3/386	4 0%
	Phase1	1/16	31 8%	1/32	15 9%	1/64	8 0%	1/128	4 0%
	Phase2	2/33	30 9%	2/65	15.1%	2/129	7 9%	2/257	4 0%

- For each machine configuration, the three versions of Loop 9 all reflect similar processor usage. From this observation we can conclude that the computation rate of a loop is insensitive to the number of critical cycles in its body.

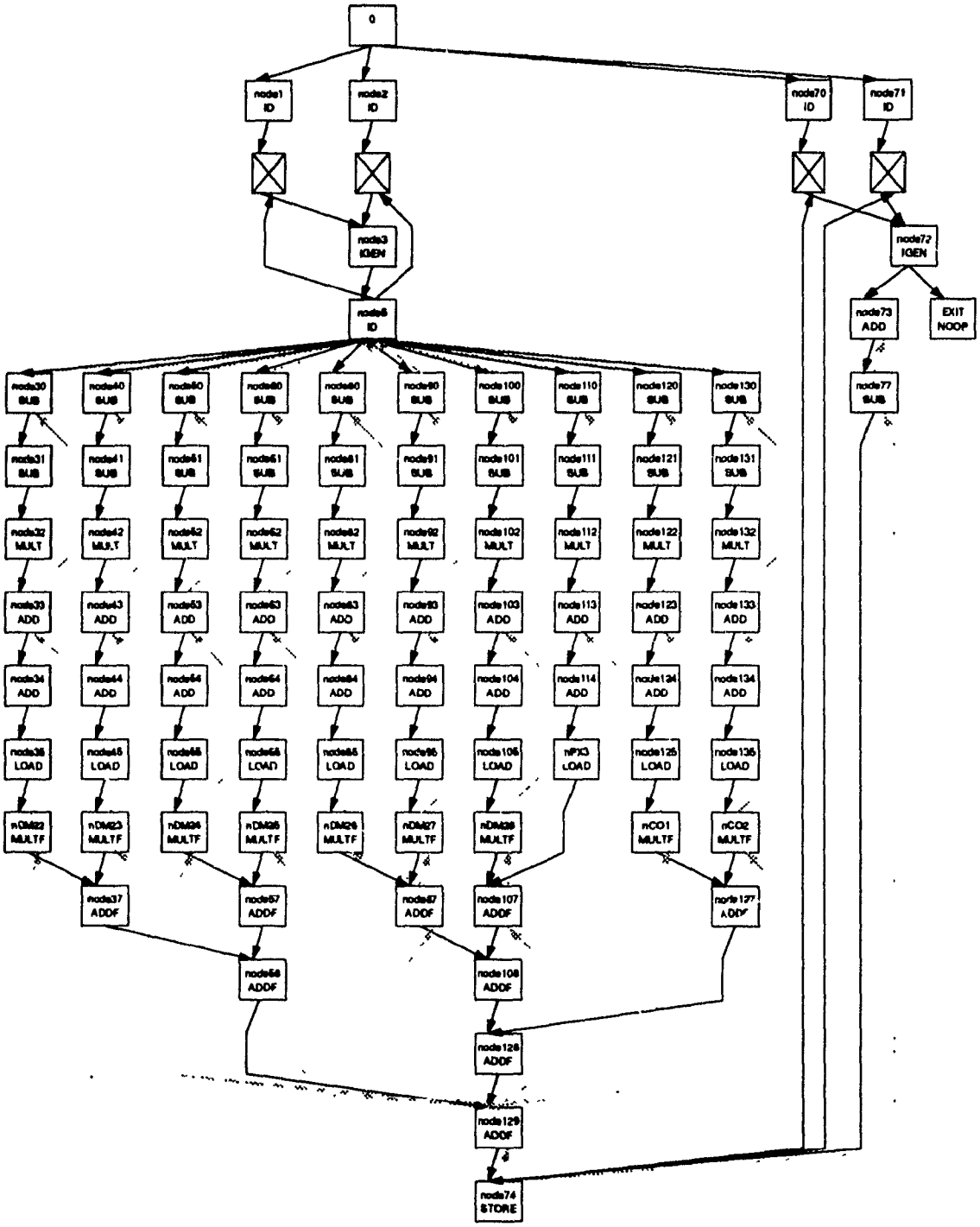


Figure 6.5: Loop 9 under Aggressive Limited Balancing

Chapter 7

Related Work

7.1 Software Pipelining

Software pipelining is a well known technique for exploiting fine-grain parallelism in loops, by reorganizing statements in successive iterations of a loop body so as to execute in a pipelined fashion. The idea originally emerged from the microprogramming community as a means for a pipelined processor to execute vector operations [Kog77]. Since then there have been variations of the technique proposed for loop scheduling [Aik88, Ebc87, EN90, Gao90, Lam89, RG81, SDX86, Tou84]. In this section we survey several typical methods and compare their fundamental ideas. For this discussion, we closely follow the terminology used in [JA90]. We refer to each operation in the original loop body as a *micro-operation* (MO) and the compacted operations as a *micro-instruction* (MI). Accordingly, each MI can contain several MOs after compaction. *Initiation interval* is equivalent to the cycle time concept we defined earlier.

The software pipelining schemes to be discussed include perfect pipelining [Aik88], enhanced pipelining [EN90], URPR algorithm [SDX86], and software pipelining for the Warp [Lam89]. All of these schemes handle loops having loop-carried dependence and conditional constructs. Their major differences lie in their hardware assumptions and their approaches to construct the steady state. They can be divided roughly into two categories: *compact unrolling* and *trial-and-error*.

Compact unrolling unrolls the loop body a number of times and then compacts the unrolled sequence subject to a given compaction algorithm. The repeated pattern spotted in the compacted sequence forms the steady state; its length is the initiation interval. The number of unrollings and the aggressiveness of the compaction algorithm correspond to variations of the earliest firing rule, and hence, affect the optimality of the schedule. Perfect pipelining and enchanced pipelining are examples of compact

unrolling. The trial-and-error methods construct the steady state based upon a series of trials on a range of initiation intervals. The smallest achievable one is taken as the initiation interval. URPR algorithm and Lam's software pipelining scheme are good examples.

7.1.1 Perfect Pipelining

Aiken and Nicolau's work on perfect pipelining [Aik88, AN88] consists of two steps: infinite unrolling and code compaction. A data dependence graph (DDG) expressing the partial order among the MOs presents the ultimate dependence constraint to be followed. Assume that the loop body G is initially expressed in a sequence of MOs, obtained by sorting the DDG topologically and temporarily ignoring loop-carried dependence edges.

Perfect pipelining unrolls G infinitely, i.e., the sequence of MOs is replicated infinitely. Then each MO in the unrolled loop body G' is moved upwards as much as possible with respect to the compaction algorithm, subject to all data dependences and resource constraints. Throughout the process, the sequence of MIs is searched for a repeated pattern. Once a pattern is detected, the prelude sequence is formed directly from the sequence of MIs before the steady state and the postlude sequence is attached accordingly.

As perfect pipelining is applied to a condition-free non-nested loop body without resource restrictions, a time-optimal pipelined schedule is obtained [AN88]. No transformation of the loop based upon the given data dependence can yield a shorter running time. Since each MO is moved to the earliest possible starting position during compaction, the resulting schedule is an earliest firing schedule. Thus, for a machine without resource restrictions, compact unrolling achieves similar results as executing the DDG under the earliest firing rule. The results from our study regarding how to reach steady state is also applicable in this case.

When a conditional statement is considered within a loop body, perfect pipelining finds a repeated pattern on each path regardless of the flow of control. In this case, several predicate MOs can be compacted into an MI. To satisfy the requirement for evaluating multiple predicates, the underlying architecture assumes the function of performing multi-way branching. This feature allows several predicates to be evaluated together within a long instruction word in order to select the subsequent branch point.

7.1.2 Enhanced Pipelining

Assume the loop body G is expressed using a sequence of MOs obtained by topologically sorting the DDG as before. MOs which are at the same depth in the DDG are said to belong to the same *level* and stay consecutively in the sequence. Initially G is unrolled infinitely to form G'. To facilitate explanation, we refer to the MO on each row of G' as MI. The term *window* w_i denotes a region of consecutive MIs in the unrolled sequence G'. Initially, window w_i is set to cover the i-th unrolled loop body in G'.

Ebcioğlu and Nakatani's *enhanced pipelining* is similar to perfect pipelining in the sense that it also performs unrolling and compaction [EN90]. In addition, they also assume a similar multi-way branching capability to handle conditional constructs. However, they speed convergence of steady state by enforcing two rules.

- MI is the basic unit of moving for compaction, i.e., compacted MIs cannot be decomposed.
- Compaction is only applied to MIs within the same window. To be more precise, MIs to be filled at compaction time must be currently at the top *level* of the window. In addition, candidates used to fill the top-level MIs must be chosen from the same window as well. This second rule ultimately constrains the formation of the steady state to contain only one copy of each MO from the original loop body.

Enhanced pipelining applies the same compaction move to each $w_i, \forall i$, subject to data dependence constraints, resource constraints, and the compaction rules just described. After compacting the top-level MIs, all windows are adjusted downward by one level of MIs. This adjustment has the effect of moving the previous top level MIs to the bottom of the window. At this point, compaction is carried out again with the current top-level MIs. If there is no loop-carried dependence, MIs from the bottom level of w_i, which belong originally to the first level of the $i+1$ iteration, move up to the top level of w_i, which corresponds to the second level of iteration i. Thus, software pipelining occurs. Note that every window maintains an identical copy of MIs.

Compaction and window adjustment are repeated until compaction has been tried on each level. The resulting formation of MIs in window w_1 contributes to the steady state. The prelude sequence can be obtained directly from the sequence of MIs before steady state (w_1) and the postlude sequence is similarly attached. Note that each MO from the original loop body appears only once in steady state. The advantage of this approach lies in the acceleration of steady-state formation. Even so, prematurely

tying parallelism together reduces the flexibility of fine-grain scheduling for utilizing available resources.

7.1.3 The URPR Algorithm

URPR (UnRolling, Pipelining, and Rerolling) originally applies only to loops which contain a single basic block [SDX86]. More recently the technique has been extended into what is now called GURPR (Global URPR). GURPR incorporates conditionals, nested loops, and subroutine calls [SDWX87]. For the purpose of this survey, we focus only on URPR, which contains the basis for software pipelining. The URPR algorithm is similar to trial-and-error. First, the initiation interval is computed; the loop body is unrolled and then pipelined with respect to the interval all in one trial.

Initially the loop body G is locally compacted with respect to intra-data dependence and resource constraints, into a sequence of MIs. Once this is accomplished, MOs become indivisible and are manipulated as a single entity. Let G' denote the precompacted loop body and S the schedule to be built incrementally, and let MI_s^t and MI_i^j denote the t-th MI of S and the j-th MI of the i-th unrolled body of G'. The algorithm consists of three stages:

1. **Unrolling:** G' is unrolled $k = \lceil l/d \rceil$ times, where l is the length of G' and d is the initiation interval computed using the maximum inter-body data dependence distance that spans one iteration.[1] Intuitively we need only to unroll the loop bodies k times to uncover the steady states if each initiation is d cycles apart.

2. **Pipelining:** The k loop bodies are pipelined. S is initiated with the first loop body, a copy of G' with l MIs. The remaining loop bodies are added to S one by one, subject to initiation interval and data dependence. If a resource conflict occurs between MI_s^t and MI_i^j while adding the i-th loop body, MI_i^j is delayed to compact with MI_s^{t+1}. If it fails again, a new MI_s containing only MI_i^j is inserted between MI_s^t and MI_s^{t+1}. The reason for this insertion is to keep the distance between MI_i^j and MI_{i-1}^j as close to d as possible so the pipelined steady state is shorter. Compaction is continued with MI_i^{j+1} until all MIs from the k loop bodies are added to S.

3. **Rerolling:** Steady state is formed from a sequence of adjacent MI in S with shortest cycle time. To ensure the steady state consists of an equal number of MOs, redundant MOs are removed from S. URPR further simplifies the

[1] It was later pointed out that the calculation should be done for all dependences [JA90]

selection by restricting the steady state to contain exactly one copy of each MO. Based upon the selection, prelude and postlude sequences are constructed

The GURPR scheme of handling conditional constructs is similar to the global compaction technique called trace scheduling [Fis81]. Separate paths are compacted, pipelined, and rerolled individually. Bookkeeping operations are then added for semantic adjustment.

The URPR algorithm has the advantage of low computational complexity in building a schedule. On the other hand, it prematurely pushes potential parallelism together, losing the flexibility of fine-grain parallelism. In addition, restraining one copy of G' in steady state eliminates the opportunity for forming a denser pipeline. The GURPR algorithm also suffers from the same criticism of trace scheduling. There is no reason why the same path will be repeatedly executed in the loop body.

7.1.4 The Systolic Array Optimizing Compiler

Lam's software pipelining algorithm is tailored to code generation for a VLIW systolic architecture known as Warp [Lam88, Lam89]. For this machine, the DDG is slightly different. Each micro-operation (MO) composes a sequence of indivisible operations; once an MO is initiated the entire sequence must run to completion without interruption. Lam's algorithm is a typical trial-and-error scheme. A range of initiation intervals is first obtained, and then a sequence of trials on the chosen interval are initiated. For each trial, each MO is forced to schedule regularly at the interval.

The starting point of the trial range is computed using resource and precedence constraints. Given an acyclic loop body G and a current trial interval d, same MO from successive iterations are scheduled exactly d steps apart while each MO within a G is list scheduled.

For an MO_i scheduled at time t, resource usage is checked against the u, $u+d$, $u+2d$, ... indivisible operations of the scheduled MO_j having its u-th indivisible operation executed at time t. If a resource conflict occurs, MO_i is delayed one cycle and the resource check repeated. However, if the operation fails to be scheduled within the range $[t, t+d-1]$, the entire schedule is abandoned, and a new trial started using the interval $d+1$.

If there is loop-carried dependence inside G, strongly-connected components are scheduled first. Then each component is reduced into an MO_j by merging resource requirements. The set of nodes in the components become an indivisible sequence of MO_j. Again an acyclic graph is obtained which can be scheduled using the acyclic-graph scheduling scheme.

A distinct characteristic of this approach is its way of scheduling conditionals using so-called hierarchical reduction. Similar to the schedule for strongly-connected components, the inner most sub-branch of a conditional is list scheduled and reduced into an MO by merging resource requirements. This reduction is then repeated for the outer conditional. Because the entire conditional construct is reduced into a single node, the acyclic-graph scheduling scheme is used.

The major difference between Lam's algorithm and the URPR algorithm is that once a resource conflict occurs, the former scheme abandons the entire schedule and starts a new trial over a larger interval whereas URPR does not.

7.1.5 Remarks

The valid-schedule-computation scheme (Lemma 3.1) introduced in Chapter 3.7 provides a typical example of software pipelining [RH80, Rei68]. This approach mathematically computes a valid schedule for a live-bound Petri net. However, the technique is only applicable to ideal machines and machines without resource constraints. One very nice characteristic of this approach is that the resulting schedule enters steady state as soon as every transition is fired once, i.e., after the first iteration.

Among the various software pipelining schemes introduced, perfect pipelining possesses the most similarity to SPS. In fact, SPS was inspired by perfect pipelining. For a loop operated in an ideal machine, Aiken and Nicolau gave an $\mathcal{O}(n^3)$ bound in time steps (or equivalently, $\mathcal{O}(n^2)$ iterations) to find a pattern in the single critical cycle case, and they noted that the least-common-multiple effect incurred by the multiple critical cycles would seldom occur in practical situations [Aik88, AN88]. We could not justify their proofs and so reinvestigated the problem in this thesis.

Under the assumption of an ideal machine model and for a class of loops having only a single critical cycle, steady state appears after $\mathcal{O}(n^3)$ iterations, where n denotes the size of the loop body. For the case of multiple critical cycles, the length of the steady state is directly proportional to a common multiple of critical cycles. We are unaware of any polynomial bound for the length of the prelude sequence in this case; instead, we have derived an approach with polynomial time complexity by fixing an initial condition. Doing so, we were able to achieve a significant improvement in efficiency in finding a schedule, regardless of the number of critical cycles.

7.2 Storage Allocation

Two strategies, based upon software and hardware support respectively, treat storage allocation for software pipelining:

- The software approach solves the register allocation problem by the use of conventional graph coloring techniques [CAC+81, Cha82, Tou84]. Since the number of registers allocated to each variable is unknown until the schedule is computed, the general graph coloring approach can only be applied on the final schedule. It is assumed that a large amount of registers are available initially so the scheduling phase can be handled independently of register constraints. After scheduling, graph coloring is performed globally for register allocation. Spill codes are inserted in the schedule to reuse registers in the absence of unallocated ones.

  Once a schedule is fixed, it is difficult to reduce register usage because the goal is to avoid the insertion of spill code. In Lam's case, the problem becomes harder because of the indivisible operations. The major criticism of the graph coloring approach is the degradation of the performance that results from the insertion of spill code. Spill code unavoidably lengthens the initiation interval of a pipeline. Since spill code needs main memory access, the impact on schedule throughput is more severe.

- The hardware approach solves the register allocation problem using special hardware. The particular type of architecture that uses this concept is called a *polycyclic* architecture [RG81]. Intuitively, each data arc in the DDG is implemented with a FIFO queue. After each operation execution, a result is appended to the output queues. The design significantly simplifies the work of compiler storage allocation. However, the number of required FIFO queues and their associated length vary for different programs, while the amount of hardware resources are limited.

Limited balancing binds scheduling and storage allocation into a unified framework. In limited balancing, special concern is given to the issue of exploitable parallelism under resource constraints. For simple type of machine assumed in this thesis, the amount of exploitable parallelism varies according to both the number of pipelined processors and the number of registers. With respect to these two critical factors, limited balancing restructures the software pipeline to achieve adequate parallelism in the machine. The amount of storages required in SPS is already bounded by the number of nodes in the loop body because the construction of the schedule is based upon a static dataflow graph. Limited balancing offers further opportunity to reduce storage usage based upon exploitable parallelism. The advantages of limited balancing include:

- When enough registers are available, limited balancing reduces register usage without affecting the throughput of the software pipeline.
- When lack of registers, unlike conventional graph coloring schemes, no spill code is required. Instead, a lower balancing ratio is chosen for limited balancing to reduce the requirement further, thus avoiding the possible interruption of memory access.
- For the SPS scheme, limited balancing helps to shorten the time to reach steady state by theoretically placing each node on a critical cycle, satisfying the initial condition given in Theorem 4.7.
- For static dataflow architectures, limited balancing also helps to reduce synchronization costs.

Chapter 8

Conclusion and Future Research

The application of Petri-Net theory to compiler design received attention as early as 1970 [SS70]. Similar work, reported recently, has to do with microprogram optimization of loops on a pipelined architecture, where resource constraints such as registers and functional units are modeled within a unified Petri-net framework [Han89] This work indicates that the search for an optimal schedule has exponential complexity in general. In this thesis, we have introduced a new Petri-net model to study fine-gram loop scheduling. The followings are the results of our research.

- We have shown the development of a Petri-net loop model called an SDSP-PN wherein loops are first translated into a class of static dataflow graphs known as a *static dataflow software pipeline* (SDSP) and then this SDSP is translated into an SDSP-PN. When an SDSP-PN is executed according to the *earliest firing rule*, a steady state appears in the behavior graph within a bounded number of steps. We show that (1) in an SDSP-PN having a single critical cycle, a polynomial bound can be established for the steady state to occur (for all nodes in the loop) under the earliest firing rule. (2) In an SDSP-PN having multiple critical cycles, a polynomial bound can be established for the steady state to occur only for nodes resided on the critical cycles. (3) In addition, we have shown that the impact on the length of the prelude sequence for multiple critical cycles can be circumvented by imposing an initial token-distribution constraint This constraint ultimately accelerates the emergence of steady state, regardless of the number of critical cycles in the loop body. (4) From steady state, a time-optimal schedule for the corresponding loop can be derived.

- We have presented a methodology for integrating resource limitations into our model. Through it we have demonstrated how a timed Petri-net model known as an SDSP-MCP-PN can be constructed to model execution of an SDSP on

architectures having any number of clean execution pipelines.

- Simulation results on a number of Livermore loops, both with and without loop-carried dependences, have demonstrated that steady state for both the SDSP-PN and the SDSP-MCP-PN can be determined at compile-time in $\mathcal{O}(n)$ time, where n is the number of instructions in the loop body. This demonstrates the feasibility of determining steady state at compile time.

- For storage allocation, we have justified limited balancing as a solution. With this method, a scheduled loop can maintain execution speed without using extra storage. Simulation results also verified the correctness of our mathematical guideline for finding a feasible balancing ratio.

Shown below is ongoing research we intend to pursue to solve the problem of fine-grain scheduling and storage optimization:

- Incorporation of conditional constructs. Due to the unpredictable run-time behavior of conditional branches, consideration of conditional branches inside loops is a major obstacle in the design of a compile-time loop scheduling scheme. Preliminary ideas on the implementation of conditional constructs are documented in [GWN91b].

- Extension of our scheduling method using dataflow models other than static dataflow, to study time-optimal scheduling. Two such models are the tagged-token dataflow model [AG78] and the FIFO-queued dataflow model [Kah74]. Both models have eliminated the one-token-per-arc restriction assumed in the static model. The tagged-token model allows a pool of tokens on a single arc and distinguishes tokens by color. For the FIFO model, each arc is a FIFO queue capable of holding multiple tokens.

- Storage optimization. The results from evaluating our model suggest that critical cycles in a program determine the achievable performance of a software pipelined loop. This opens up new opportunities for storage optimization through time-optimal scheduling. For example, storage optimization of various dataflow graph models might be studied with this insight. For the latest results in this area, see [GN91].

- Application to other machine models. The scheduling method in this thesis described might be applied to other machine models to verify scheduling effectiveness.

Appendix A

Example: A-code graph for Loop 3

Loop3 is a Livermore Loop which computes the dot products of two one dimensions array. Its corresponding A-code graphical representation is given in Figure A.1

- Initially, all dotted arcs are assigned a token while the solid ones are empty. All labeled nodes except node 0 are regular actors; each of which represents a single instruction and is executed in the execution pipeline when enabled.

- Node 0 is always recognized by the machine model as the starting node of a program. Its adjacency list points to all starting nodes of the program. As the program started, all nodes on the list are signaled directly

- Each crossed square in the program graph denotes a signal merge structure. Note that it is not a node consisting of an operation. It is merely drawn to show the detail of the signal flow. Its output node receives a signal if either one of its input node sends a signal.

The operations of most nodes are self-explanatory with the description on it. In particular, ID stands for an identity actor, which copies the value from the input register to its result register. IGEN stands for the index generator; it generates a sequence of integer index within the range of two input values. The readers are referred to [GP88, Tio88a, Tio88b] for a more detail explanation of A-code operations. A deeper insight to each portion of the code is given below:

- First four levels of the graph are the loop initiation sequence. All IGENs are loaded with the necessary input values.

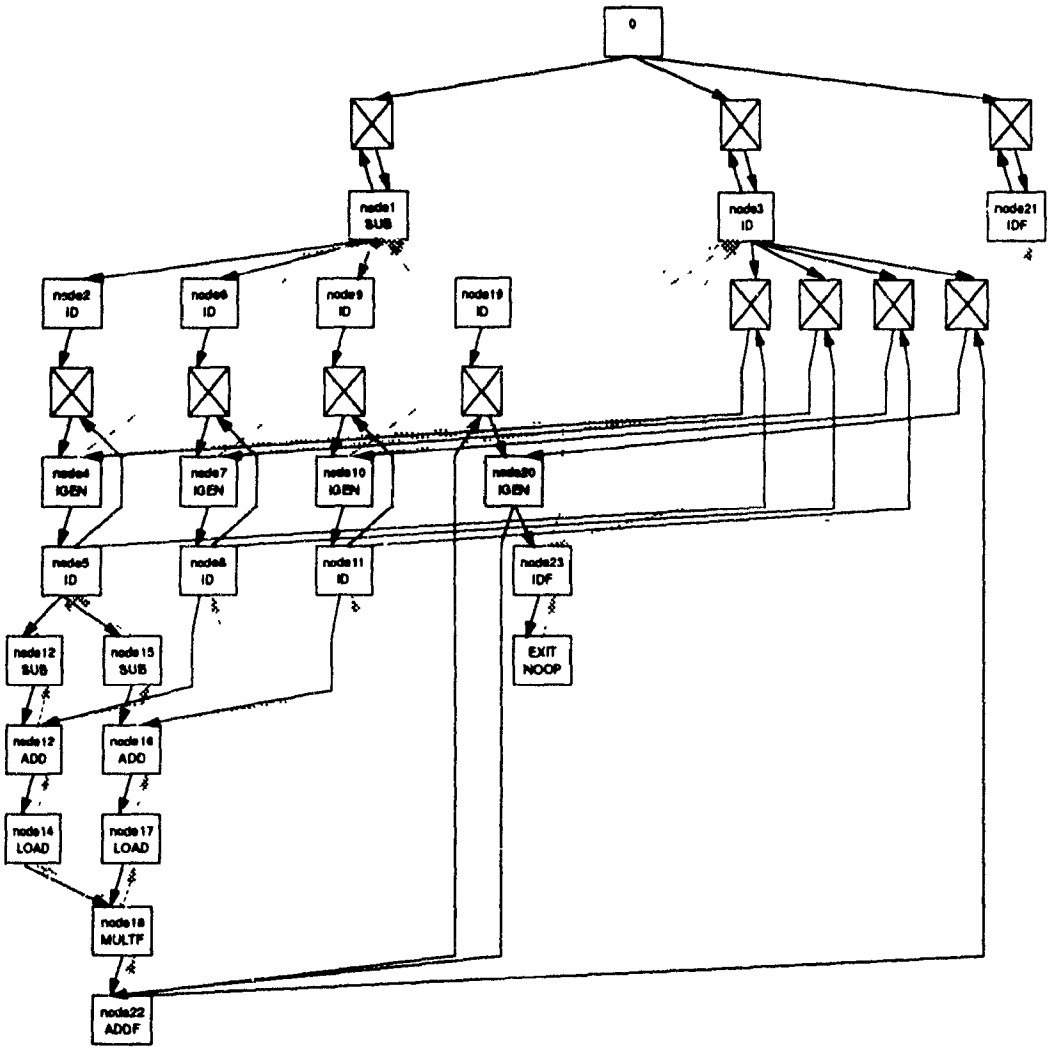


Figure A.1: A-code Graphical View of Loop 3

- Node4 serves the purpose of index generation for the loop body while node20 guides the proper loop termination. Node7 and node10 are used for loop constant propagation.
- Each branch under node5 (i.e., node12 to node14 and node15 to node17) corresponds to the address computation sequence and the element load operation of an input array.
- The actual dot product multiplication and addition are done at node18 and node22.

Bibliography

[AC86] Arvind and D. E. Culler. Dataflow architectures. *Annual Reviews in Computer Science*, 1:225–253, 1986.

[ADNP88] Arvind, M. L. Dertouzos, R. S. Nikhil, and G. M. Papadopoulos. Project dataflow—the Monsoon architecture and the Id programming language. Computation Structures Group Memo 285, Laboratory for Computer Science, MIT, March 1988.

[Aea83] Arvind and et al. The tagged token dataflow architecture (preliminary version). Technical report, Laboratory for Computer Science, MIT, Cambridge, MA., August 1983.

[AG78] Arvind and K P. Gostelow. Some relationships between asynchronous interpreters of a data flow language. In E. J. Neuhold, editor, *Formal Description of Programming Concepts*, pages 95-119. North-Holland, 1978.

[AG82] Arvind and K. P. Gostelow. The U-Interpreter. *IEEE Computer*, 15(2):42–49, February 1982.

[Aik88] A. Aiken. Compaction-based parallelization. (PhD thesis), Technical Report 88-922, Cornell University, 1988.

[AN88] A. Aiken and A. Nicolau. Optimal loop parallelization. In *Proceedings of the 1988 ACM SIGPLAN Conference on Programming Languages Design and Implementation*, June 1988.

[BG89] D. Bernstein and I. Gertner. Scheduling expressions on a pipelined processor with a maximal delay of one cycle. *ACM Transactions on Programming Languages and Systems*, 11(1):57–66, January 1989.

[CAC+81] G. J. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein. Register allocation via coloring. *Computer Languages 6*, pages 47–57, January 1981.

[CCG84] J. Carlier, P. Chretienne, and C. Girault. Modeling scheduling problems with timed Petri nets. In G. Goos and J. Hartmanis, editors, *Advances in Petri Nets, LNCS 340*, pages 62-82. Springer-Verlag, Berlin, Heidelberg, NY, 1984.

[Cha82] G. J. Chaitin. Register allocation and spilling via graph coloring. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pages 98-105, June 1982.

[CHEP71] F. Commoner, A. W. Holt, S. Even, and A. Pnueli. Marked directed graphs. *Journal of Computer and System Sciences*, 5:511–523, 1971.

[Chr84] P. Chretienne. *Les Reseaux de Petri Temporisés (These d'état,* PhD thesis, Institut de programmation, Université P. et M. CURIE, C.N.R.S.-E.R.A. 592, September 1984.

[Chr85] P. Chretienne. Timed event graphs: A complete study of their controlled executions. In *International Workshop on Timed Petri Nets*, pages 47–54, Torino, Italy, July 1985. IEEE Computer Society Press.

[Den84] J. B. Dennis. Data flow models of computation. In *Notes from lectures at the International Summer School on Control Flow and Data flow: Concepts of Distributed Programming.* Springer-Verlag, Marktoberdorf, Germany, 1984.

[Den91] J. B. Dennis. Evolution of the static dataflow architecture. In *Advanced Topics in Dataflow Computing.* Prentice-Hall, 1991.

[DFL72] J. B. Dennis, J B. Fosseen, and J P. Linderman. Data flow schemas. In *International Symposium on Theoretical Programming, LNCS 5*, pages 187–215. Springer-Verlag, Berlin, 1972.

[DG88] J. B. Dennis and G. R. Gao. An efficient pipelined dataflow processor architecture. In *Proceedings of the Supercomputing '88 Conference*, pages 368-373, Florida, November 1988. IEEE Computer Society and ACM SIGARCH.

[Ebc87] K. Ebcioğlu. A compilation technique for software pipelining of loops with conditional jumps. In *Proceedings of the 20th Annual Workshop on Microprogramming*, December 1987.

[EN90] K. Ebcioğlu and T. Nakatani. A new compilation technique for parallelizing loops with unpredictable branches on a vliw architecture. In D Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, pages 213-228. MIT Press, 1990.

[Feo88] J. T. Feo. An analysis of the computational and parallel complexity of the Livermore loops. *Parallel Computer*, 8(7):163 185, July 1988.

[Fis81] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 7(C-30):478 490, July 1981.

[Fis83] J. A. Fisher. Very long instruction word architectures and the ELI-512. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, 1983

[Gao86] G. R. Gao. A maximally pipelined tridiagonal linear equation solver. *Journal of Parallel and Distributed Computing*, 3(2):215 235, June 1986.

[Gao89] G. R. Gao. Aspects of balancing techniques for pipelined data flow code generation. *Journal of Parallel and Distributed Computing*, 6:39 61, 1989.

[Gao90] G. R. Gao. *A Code Mapping Scheme for Dataflow Software Pipelining.* Kluwer Academic Publishers, Boston, December 1990.

[GHW90a] G. R. Gao, H. H. J. Hum, and Y. B. Wong. An efficient scheme for fine-grain software pipelining. In *Proceedings of the CONPAR '90- VAPP IV Conference*, pages 709–720, Zurich, Switzerland, September 1990.

[GHW90b] G. R. Gao, H. H. J. Hum, and Y. B. Wong. Limited balancing an efficient method for dataflow software pipelining. In *Proceedings of the International Symposium on Parallel and Distributed Computing, and Systems*, New York, NY, October 1990.

[GHW90c] G. R. Gao, H. H. J. Hum, and Y. B. Wong. Parallel function invocation in a dynamic argument-fetching dataflow architecture. In *Proceedings of PARBASE '90-International Conference on Databases, Parallel Architectures, and Their Applications*, pages 112 116, Miami Beach, FL, March 7–9 1990. IEEE Computer Society.

[GN91] G. R. Gao and Qi Ning. Loop storage optimization for dataflow machines. In *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*, San Jose, California, August 1991.

[GO90] R. D. Groves and R. Oehler. RISC system/6000 processor architecture. In *IBM RISC System/6000 Technology*. International Business Machines Corp., 1990.

[GP88] G. R. Gao and Z. Paraskevas. Efficient software pipelining in an argument-fetching dataflow architecture. ACAPS Technical Memo 02, School of Computer Science, McGill University, Montreal, March 1988. Presented at the Canadian Conference on Electrical and Computer Engineering, Montreal, September 89.

[GP90] G. R. Gao and Z. Paraskevas. Compiling for dataflow software pipelining. In David Gelernter, Alexandru Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, pages 275 306. The MIT Press, 1990.

[GT89] G. R. Gao and R Tio. Instruction set design of an efficient pipelined dataflow architecture. In *Proceedings of the 22nd International Conference of System Science*, pages 383–393, Hawaii, January 1989. IEEE Computer Society.

[GTH88] G. R. Gao, R. Tio, and H. H. J. Hum. Design of an efficient dataflow architecture without dataflow. In *Proceedings of the International Conference on Fifth-Generation Computers*, pages 861 868, Tokyo, Japan, December 1988.

[GWN91a] G. R. Gao, Y. B. Wong, and Qi Ning. A Petri-Net model for fine-grain loop scheduling. In *Proceedings of the '91 ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 204 218, Toronto, Canada, June 1991.

[GWN91b] G. R. Gao, Y. B. Wong, and Qi Ning. A Petri-Net model for fine-grain loop scheduling. ACAPS Technical Memo 18, School of Computer Science, McGill University, Montreal, January 1991.

[GWN91c] G. R. Gao, Y. B. Wong, and Qi Ning. A Petri-Net model for loop scheduling. In *Proceedings of the 12th International Conference on Application and Theory of Petri Nets*, Gjern, Denmark, June 1991.

[Han89] C. Hanen. Optimizing microprograms for recurrent loops on pipelined architectures using timed petri nets. In G. Rozenberg, editor, *Advances in Petri Nets, LNCS 424*, pages 236–261. Springer-Verlag, 1989.

[HG83] J. Hennessy and T. Gross. Postpass code optimization of pipelined constraints. *ACM Transactions on Programming Languages and Systems*, 5(3):422–448, July 1983.

[HP90] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1990.

[JA90] R. B. Jones and H. A. Allan. Software pipelining: A comparison and improvement. In *Proceedings of the 23th Annual Workshop on Microprogramming and Microarchitecture*, pages 46-56, Orlando, Florida, November 1990.

[Kah74] G. Kahn. The semantics of a simple language for parallel processing. In *Information Processing 74*, pages 471–475, 1974.

[KLL86] S. Y. Kung, S. C. Lo, and P. S. Lewis. Timing analysis and optimization of VLSI data flow arrays. In *Proceedings of the 1986 International Conference on Parallel Processing*, 1986.

[KM89] L. Kohn and N. Margulis. Introducing the Intel i860 64-bit microprocessor. *IEEE Micro*, pages 15–30, August 1989.

[Kog77] P. Kogge. The microprogramming of pipelined processors. In *The 4th Annual Symposium on Computer Architecture*, pages 63–69, March 1977.

[Kog81] P. M. Kogge. *The Architecture of Pipelined Computers*. McGraw-Hill Book Company, New York, 1981.

[Lam88] Monica Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the 1988 ACM SIGPLAN Conference on Programming Languages Design and Implementation*, pages 318–328, Atlanta, GA, June 1988.

[Lam89] M. S. Lam. *A Systolic Array Optimizing Compiler*. Kluwer Academic Publishers, 1989.

[Mag84] J. Magott. Performance evaluation of concurrent systems using Petri nets. *Information Processing Letters, North-Holland*, 18:7–13, January 1984.

[Mea85] J. R. McGraw and et al. SISAL: Streams and iteration in a single assignment language—language reference manual version 1.2. Technical Report M-146, Lawrence Livermore National Laboratory, 1985.

[Mel89] C. Melear. The design of the 88000 RISC family. *IEEE Micro*, pages 26–38, April 1989.

[MK80] T. Murata and J. Y. Koh. Reduction and expansion of live and safe marked graphs. *IEEE Transactions on Circuits and Systems*, 27(1):68–71, January 1980.

[Mur80] T. Murata. Synthesis of decision-free concurrent systems for prescribed resources and performance. *IEEE Transactions on Software Engineering*, 6(6):525–530, November 1980.

[Mur89] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989

[NPA88] A. Nicolau, K. Pingali, and A. Aiken. Fine-grain compilation for pipelined machines. Technical Report TR-88-934, Department of Computer Science, Cornell University, Ithaca, NY, 1988.

[Par88] Z. Paraskevas. Summary of the discussion for the expansion of instruction set of the argument-fetching architecture. ACAPS Design Note 05, School of Computer Science, McGill University, Montreal, November 1988.

[Par90] Z. Paraskevas. Code generation for dataflow software pipelining. Technical Report TR-SOCS-89.9, School of Computer Science, McGill University, Montreal, Jananuary 1990.

[Pet62] C. Petri. *Kommunikation mit Automaten*. PhD thesis, University of Bonn, Bonn, West Germany, 1962.

[Pet81] J. L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1981.

[Ram74] C. Ramchandani. Analysis of asynchronous concurrent systems. Technical Report TR-120, Laboratory for Computer Science, MIT, 1974.

[Rei68] R. Reiter. Scheduling parallel computation. *Journal of ACM*, 15(4):590–599, October 1968.

[RG81] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proceedings of the 14th Annual Workshop on Microprogramming*, pages 183–198, 1981.

[RH80] C. V. Ramamoorthy and G. S. Ho. Performance evaluation of asynchronous concurrent systems using Petri Nets. *IEEE Transactions on Computers*, pages 440–448, September 1980.

[SDWX87] B. Su, S. Ding, J. Wang, and J. Xia. GURPR—a method for global software pipelining. In *Proceedings of the 20th Annual Workshop on Microprogramming*, pages 88–96, Colorado, December 1987.

[SDX86] B. Su, S. Ding, and J. Xia. URPR—a extension of URCR for software pipelining. In *Proceedings of the 19th Annual Workshop on Microprogramming*, pages 94–103, New York, October 1986.

[SS70] R. Shapior and H. Saint. A new approach to optimization of sequencing decisions. *Annual Review in Automatic Programming*, 6:257–288, 1970.

[Tio88a] R. Tio. The A-code assembly language reference manual. ACAPS Design Note 02, School of Computer Science, McGill University, Montreal, July 1988.

[Tio88b] R. Tio. DASM: The A-code data-driven assembler program reference manual. ACAPS Design Note 03, School of Computer Science, McGill University, Montreal, July 1988.

[Tou84] R. F. Touzeau. A FORTRAN compiler for the FPS-164 scientific computer. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 48–57, June 1984.