SCHEDULING AND BEHAVIORAL TRANSFORMATIONS FOR PARALLEL SYSTEMS

Liang-Fang Chao

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE

October 1993

To my parents,

for their unconditional love and support.

Abstract

In a parallel system, either a VLSI architecture in hardware or a parallel program in software, the quality of the final design depends on the ability of a synthesis system to exploit the parallelism hidden in the input description of applications. Since iterative or recursive algorithms are usually the most time-critical parts of an application, the parallelism embedded in the repetitive pattern of an iterative algorithm needs to be explored. This thesis studies techniques and algorithms to expose the parallelism in an iterative algorithm so that the designer can find an implementation achieving a desired execution rate. In particular, the objective is to find an efficient schedule to be executed iteratively.

A form of *data-flow graphs* is used to model the iterative part of an application, e.g. a digital signal filter or the while/for loop of a program. Nodes in the graph represent operations to be performed and edges represent both intra-iteration and inter-iteration precedence relations. A schedule of the operations is executed repeatedly with certain execution rate. Two transformation techniques, *retiming* (pipelining) and *unfolding* (unwinding), are combined to transform a data-flow graph into another data-flow graph while preserving the intended behavior of the application. After the interaction of retiming and unfolding is analyzed, it is shown that the order of retiming and unfolding is immaterial. Algorithms are designed to find a transformation such that there exists a schedule achieving a given execution rate or the optimal execution rate.

This new technique turns out to be very useful, and can be generalized to other problems. For example, the problem of *software pipelining* in parallel compilers is modeled as a special case of our technique. Efficient polynomial-time algorithms are

derived for the scheduling on different parallel models and implementation styles. For uniform nested loops, multi-dimensional retiming and unfolding are defined and studied for nested loop scheduling.

Based on the theoretical results, a novel technique, *rotation*, is designed for loop pipelining under resource constraints. Rotation technique repeatedly transforms a schedule to a more compact schedule. The rotation scheduling gives very good performance from the experiments on many benchmarks.

Acknowledgements

I am greatly indebted to my advisor Andrea LaPaugh. She has provided valuable suggestions and motivations. Her constructive comments have improved the quality of the research and the organization of the thesis. I am appreciative for her patience and understanding.

My readers, Ken Steiglitz and Leslie Hall, have provided helpful comments and suggestions on this dissertation. The amicable discussions with Leslie Hall have made me interested in scheduling theory. Wayne Wolf has brought me into the interesting field of high-level synthesis of VLSI. The brilliant faculty members of the Computer Science department demonstrated the examples of great researchers. I feel fortunate to be able to have contact with them.

I wish to express my gratitude to my officemates (especially Ayellet Tal), my fellow graduate students. They have made my four years at Princeton pleasant and fruitful. I would like to thank Winnie Waring and Sharon Rodgers for their help with administration.

I would also like to express my appreciation to Edwin Sha. He not only collaborated with me on research, but also provided encouragement and inspiring guidance. I enjoyed numerous discussions and even arguments with him.

My sincere thanks go to my parents, Tso-Chi Chao and Yuan-Yu Chu, for their unconditional love and support since the minute I was born. Without their teaching and sacrifice, I would never reach this point.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis studies techniques and algorithms to expose the parallelism in an iterative or recursive algorithm so that the designer can find an implementation achieving the desired execution rate. In a parallel system, either a VLSI chip in hardware or a parallel program in software, the quality of the final design depends on the ability of a synthesis system to exploit the parallelism hidden in the input description of applications. As VLSI technology advances, the density of transistors in a chip increases. Since the system that is implementable by a chip is very complicated, it is difficult to design a VLSI chip manually. The need for computer-aided design tools on functional-level description becomes apparent. On the software side, as parallel machines become prevalent, the applications that make use of parallel machines are versatile. It is necessary to design a parallel compiler to accept either sequential or parallel programming languages and produce control codes for parallel machines. In both of the target implementations, hardware and software, efficient algorithms are needed to synthesize a large application.

For real-time or high-performance computing, a synchronous system or application program is executed repeatedly at a fixed execution rate in order to communicate with other parts of the entire system or to meet certain input/output requirements. Therefore, a high-level synthesis system needs to have to be able to optimize the execution rate of a design. Since iterative or recursive algorithms are usually the most time-critical parts of an application, the parallelism embedded in the repetitive pattern of an iterative algorithm needs to be explored. As the performance of computers becomes faster, we can design a high-quality and complicated synthesis system to explore all alternative designs of an application. Therefore, algorithms for such exploration are needed.

Synthesis systems usually use *data-flow graphs* to represent algorithms. Nodes in the graph represent operations to be performed and edges represent precedence relations. A data flow graph can represent an iterative or recursive algorithm or the for/while loops of a program. This thesis proposes generic techniques for the *scheduling* of such data-flow graphs, when multiple processing elements are available. Scheduling consists of determining the time at which each operation is performed such that the execution rate of the data-flow graph is optimized.

1.1 Scheduling

The scheduling of tasks on parallel machines has been extensively studied since 1970's [15, 3]. For a set of given tasks with computation times, the scheduling problem is to decide the starting time each task to satisfy certain requirements. One type of scheduling problem that has wide applications in computer science is to schedule tasks with precedence constraints so that the completion time of all tasks, called the *schedule length*, is minimized. In a traditional scheduling problem, each task is only executed once.

The precedence constraints are specified by an acyclic graph, where nodes represent tasks, and edges represent precedence constraints. The execution of a task can not start until all the predecessors finishs their computations. The path which has the largest total computation time is called *critical path*. The length of a critical path is a lower bound of the schedule length.

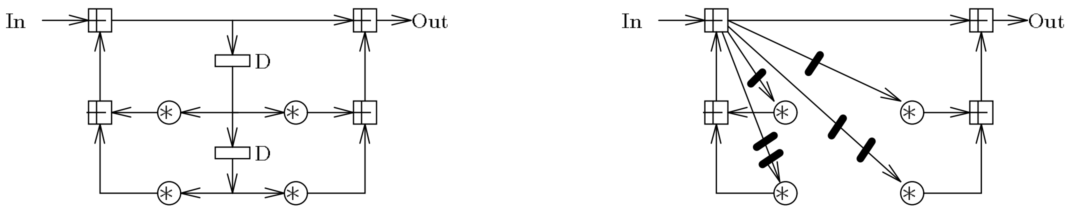
For an unlimited number of processors, the precedence-constrained scheduling problem is easy; the minimum completion time is the length of a critical path. When the number of processors is limited, the problem is NP-complete even if all tasks have the same computation time [29]. A lot of heuristic algorithms have been proposed for this problem.

As-soon-as-possible scheduling, as-late-as-possible scheduling, critical-path scheduling and list scheduling are commonly used heuristics [18]. Among them, *list scheduling* is especially popular and performs very well. These heuristics have been used in many different contexts where the scheduling problem arises, such as compiler design [26], high-level synthesis [56, 27], microcode compilation [18, 25], and real-time task scheduling [72].

The general methodology of *list scheduling* is briefly described below. The variations of list scheduling differ from each other by the way lists are prioritized. A task is *ready* if all its predecessors are scheduled. In list scheduling, a list of ready tasks ordered by a priority function is maintained at each step of the scheduling. The tasks in the list are processed one by one in the order of decreasing priority. The scheduler tries to schedule the task to be processed into the earliest possible time within the current schedule length. If the task can be scheduled, it is removed from the list. After all tasks are processed, if no task is removed from the list, the schedule length is increased to accommodate more tasks; otherwise, a new list of ready tasks is derived. The above process is repeated until the list is empty.

1.2 Background

Although the problem of scheduling arises in various contexts, there is a common platform where similer formulations and techniques are used. The input tasks are described by data/control-flow graphs, the exact form of which differs from application to application, even from system to system. Generally, the nodes in a flow graph represent tasks or operations, and arcs represent data/control dependencies or communications. The operations of the flow graph need to be executed repetitively. Execution of all operations once is referred to *an iteration.* The interval at which the successive inputs are processed is the *iteration period*; in other words, the iteration period is the (average) computation time per iteration. The goal of this scheduling problem is to minimize the iteration period.



(a) The signal flow graph (b) The corresponding data-flow graph

Figure 1.1: The second order IIR filter

Static Scheduling of Signal Processing Algorithms

Signal processing applications are nonterminating in nature. These applications process an infinite stream of inputs and produce an infinite stream of outputs. Many signal processing algorithms with recursion or feedback, especial digital filters, are described by signal flow graphs. An example is shown in Figure 1.1-(a), where the rectangle boxes are called *delays*. The signal flow graph can also be drawn as the data-flow graph in (b) so that each arc has only one head and one tail. Arcs with no delay represent precedence relations *within* an iteration, and arcs with delays represent precedence relations *between* different iterations.

Repetitive schedules that preserve the precedence relations among instances of nodes are sought. A cyclic data-flow graph has a fundamental lower bound on the iteration period, referred to as the *iteration bound* [71]. A multi-processor schedule is said to be *rate-optimal* if the iteration period equals the iteration bound.

As modern high-performance signal processing applications emerge, real-time signal processing systems of very high throughput are in great demand. Such applications include HDTV (high definition television) and multi-media networks. In order to satisfy the throughput requirement, multi-processor systems are adopted to realize data-flow graphs. Therefore, the rate-optimal scheduling of data-flow graphs is being studied by several researchers.

Some research has been done in rate-optimal scheduling of data-flow graphs based on the following assumptions [38, 39, 63, 64, 65, 78, 79]: A large number of processors are available, and fully connected; the communication costs are negligible.

Parhi and Messerschmitt studied the technique of *unfolding* [1] on data-flow graphs, where an instance of the schedule contains several iterations [63, 64, 65]. The number of iterations contained in a schedule instance is referred to an *unfolding factor*. They showed that a rate-optimal schedule can be constructed for any data-flow graph by unfolding. A class of graphs, called *perfect-rate data-flow graphs*, is identified to be able to achieve rate-optimal schedules without unfolding. An algorithm is presented to transform any data-flow graph into a perfect-rate graph with unfolding. Wang and Hu refined the above class into another class of graphs, called *generalized perfect-rate graphs*, which gives the necessary and sufficient conditions for the existence of a rate-optimal implementation [78, 79].

In the above-mentioned works, successive instances of a schedule can be *overlapped* or *pipelined* to improve the iteration period. When unfolding is under consideration, the interval at which successive instances of a schedule are initiated is called the *initiation interval*. The iteration period of a schedule of f iterations is the ratio of the initiation interval and the unfolding factor f.

A technique called *retiming* [53] was first designed to optimize systolic arrays by rearranging delays while preserving the behavior. It is recognized by these researchers that the retiming technique can also reduce the iteration period. However, these two similar techniques, pipelining and retiming, have been applied individually, and the relationship between them is unclear. This thesis will point out that retiming and pipelining are in fact the same thing, and explain the relationship between them.

Jeng and Chen [38] first attempted to apply both *retiming* and unfolding to *unit-time* data-flow graphs, where all operations have unit computation time, to achieve a rate-optimal implementation. No algorithm is presented explicitly in their work. In their later work, an algorithm for finding a rate-optimal implementation is presented in [39] for *general-time* data-flow graphs. However, the time complexity of the algorithm seems to be exponential.

All the algorithms from the above mentioned work involve the generation of *large unfolded graphs*, e.g. perfect-rate graphs, which are several times larger than the

[1] Unfolding is also called *unrolling* or *unwinding* in high-level synthesis or compiler design, where a **for** or **while** loop is concerned.

original data-flow graphs. This thesis will present polynomial-time algorithms to find rate-optimal schedules with implicit unfolding so that rate-optimal schedules can be generated without constructing large unfolded graphs.

An unfolding factor f is referred to as *rate-optimal* if a rate-optimal schedule of f iterations exists. The smaller the unfolding factor, the shorter the resultant rate-optimal schedule. Much effort has been put into reducing the rate-optimal unfolding factor. Parhi and Messerschmitt gave an upper bound on rate-optimal unfolding factors [65]. Wang and Hu proposed another upper bound on rate-optimal unfolding factors under the assumption that the iteration bound is integral. Jeng and Chen [38] claimed an upper bound for unit-time data-flow graphs without proof. They also derived an upper bound on rate-optimal unfolding factors for general-time data-flow graphs in [39]. This thesis derives the *minimum* rate-optimal unfolding factors for unit-time and general-time data-flow graphs. The minimum rate-optimal unfolding factor for unit-time data-flow graphs is smaller than the upper bound claimed in [38]. The minimum rate-optimal unfolding factor for general-time data-flow graphs is larger than the upper bound derived in [39], which implies that their upper bound is wrong.

This thesis will provide rigorous proofs of the minimum rate-optimal unfolding factors under different models in Chapter 5. A unified scheduling algorithm is presented for these models. Besides rate-optimal schedules, for a given unfolding factor we can find the schedule with the minimum iteration period under each of these models.

High-Level Synthesis

A high-level synthesis system usually takes a behavioral description as input to specify the sequence of operations to be performed by custom hardware to be synthesized [5, 56, 6]. This description is normally compiled into an internal data representation such as a data-flow graph to capture all data dependencies and a control-flow graph to capture the control structure [59, 21]. Most systems represent control flow and data flow separately by cyclic control-flow and acyclic data-flow graphs.

Synthesized hardware usually consists of several functional units (such as adders and multipliers), a data path (such as registers and multiplexers), and a control unit to control the data path. The control unit, which is usually synthesized from the control/data flow graph, can be implemented by a finite state machine or a micro-programmed control. The synthesis process is usually broken down into scheduling and synthesis of the data path, called *connectivity binding*. In the literature, a lot of data-path synthesis systems have been implemented, such as [20, 76, 70].

Scheduling algorithms assign operations to *control steps*. Each control step corresponds to one state of the controlling finite state machine. Within a control step, a separate functional unit is allocated to execute each operation assigned to that step. A *schedule* is an ordered set of control steps that covers all operations in a data-flow graph. The schedule length is the number of control steps needed. The data dependencies specified by the data-flow graph must be satisfied by the schedule.

There has been a lot of work on scheduling of acyclic data-flow graphs for high-level synthesis systems. Many variations of the common scheduling algorithms introduced in Section 1.1 are adopted in these systems. Especially, *list scheduling* is used by many systems under different priority functions [32, 57, 61]. Some typical priority functions are as-late-as-possible time, the number of descendants, and the difference between as-soon-as-possible and as-late-as-possible times (mobility) [61]. Several scheduling algorithms are compared in [37].

Force-directed scheduling [67] uses the probability distribution of operations being executed in each time step to balance the amount of hardware required. Force-directed scheduling has been shown to have very good performance. However, it is computationally more expensive than most list scheduling algorithms.

If the acyclic data-flow graph represents a loop body, which needs to be executed several times, many synthesis systems use pipelining to obtain higher performance. With pipelining, a subsequent iteration can start before the current iteration is completely finished as long as the data dependencies are not violated.

Acyclic data-flow graphs cannot represent inter-iteration dependencies, which are crucial for representing recursive algorithms, where data produced is fed back to later iterations. For a complete specification, cyclic data-flow graphs are needed, where edges are added to capture inter-iteration dependencies.

For loops where there are no dependencies between iterations, i.e., acyclic data-flow graphs, each iteration can start execution independently as long as there are resources available. For such loops, the Sehwa pipeline-synthesis system searches the design space for varying cost and performance [66]. The system either maximizes the effective performance while satisfying a cost constraint, or minimizes the total cost while satisfying a minimum performance constraint.

Loop winding [30] is a concept used to modify the acyclic data-flow graph in a preprocessing phase for pipelining. A graph with long critical paths is partitioned into several subgraphs with shorter critical paths. Each subgraph corresponds to a pipeline stage.

A few systems have been designed to pipeline loops with inter-iteration dependencies [31, 33, 50, 68, 77]. ALPS [33] formulates the scheduling problem in an integer linear programming form. The objective of the integer linear programming formulation can be either minimizing the delay for a given number of resources or maximizing the throughput for a set of functional units. Other systems use heuristics for loop pipelining under resource constraints. These systems will be reviewed in Chapter 8 when our scheduling algorithm is presented.

Software Pipelining in Compiler Design

Pipelining several instances of a schedule (program code) as described above is also called *software pipelining*, in contrast to the usual hardware pipeline. The technique of software pipelining is widely applied in various parallel compilers. The pioneering work by Monica Lam [45] performs software pipelining for a general loop program under resource constraints for the WARP computer. The unfolding technique is usually referred to as *unwinding*. The unwinding technique generates code for several iterations of a loop program, instead of performing graph transformations to obtain an unwound dependency graph. For loops without conditional jumps, some algorithms

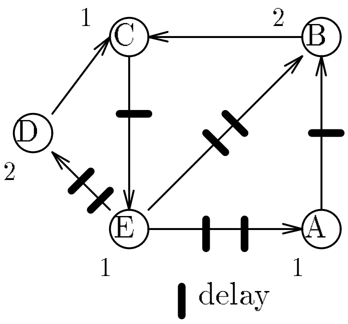


Figure 1.2: Our model of data-flow graphs

have been presented to find a repeating pattern as a rate-optimal static schedule without resource constraints [2, 28]. Aiken and Nicolau's algorithm [2] unwinds the loop incrementally until the resultant repeating pattern achieves rate-optimality. Gao, Wong and Ning use a timed petri-net model to represent inter- and intra-iteration dependencies [28]. Both algorithms can only handle edges with only one delay, and the computational complexity depends on the number of critical cycles in the graph model. Also, both algorithms cannot control the size of a resultant repeating pattern because the length of a repeating pattern is not known until it is found. By a graph-based approach, Iwano and Yeh made a breakthrough and presented a truly polynomial time algorithm [36]. This thesis will present a faster and simpler algorithm for the software-pipelining problem. Our approach, based on retiming and unfolding, is different from the approach of Iwano and Yeh.

1.3 Outline of the Thesis

We use as our model data flow graphs with delays on some edges. This model of data-flow graphs is based on the data-flow graphs used in signal processing, and is different from those used in high-level synthesis systems or compilers. However, this data-flow-graph model can also be used to describe many applications including high-level synthesis and program compilation, which may have different interpretations of computation time and delays. The input description of an iterative algorithm would be transformed into a data-flow graph under our model. This thesis develops techniques and algorithms to improve schedule length based on the manipulation of data-flow graphs with delays. This model of data-flow graphs provides a uniform representation for both data and control flow graphs. Therefore, the algorithms will also treat them uniformly.

The graph in Figure 1.2 is an example of a data-flow graph, where the number attaches to a node is its computation time. The edges with a nonzero number of delays represent inter-iteration precedence relations. The derivation of the data-flow graph from input descriptions of some applications is described in Chapter 2. A *data-flow graph (DFG) G* is called a *unit-time DFG* if the computation time of every node in G is 1.

In the ordinary pipeline model, a new iteration starts execution in every *initiation interval*. The execution periods of several iterations are overlapped. When a DFG is cyclic, the schedule cannot be overlapped arbitrarily. To resolve the difficulty, instead of looking at the original iteration, we construct a schedule which is a new loop body consisting of nodes from different original iterations. In our model, a loop pipeline is composed of a *prologue*, a *repeating schedule*, and an *epilogue*. A repeating schedule, a schedule that is repeatedly executed, forms the new loop body. The length of a repeating schedule corresponds to the initiation interval in the ordinary pipeline model. We would like to construct a repeating schedule of minimum length with or without resource constraints, The transition from the initial state of the old loop body to the new loop body is done by the *prologue*. The prologue and epilogue can be easily obtained once a static schedule is found.

Chapter 2 defines the data-flow graph model, and introduces the concept of retiming [53], which is used to interpret our loop pipeline model. Under this model, as long as a *retiming function* is found, we can generate a pipeline schedule based on this function, with or without resource constraints. Retiming on general-time DFGs only produces a small class of pipeline schedules. We consider a form of retiming on general-time DFGs *with splitable nodes* so that all pipeline schedules can be represented by this form of retiming.

In the first part of the thesis, we combine two techniques, *retiming* and *unfolding*, to obtain rate-optimal static schedules. This combination technique turns out to be very simple and efficient, and has great potential to be generalized to other problems.

An important theorem, which shows that the order of applying retiming and unfolding does not affect the performance obtained, is proved in Chapter 3. Based on this theorem, we take the approach of applying retiming first, then unfolding, to save computational cost.

Retiming algorithms with implicit unfolding are designed to find schedules of minimum length without resource constraints. Chapter 4 gives a retiming algorithm for unit-time DFGs, and applies the algorithm to the problem of software pipelining for general-time DFGs. Chapter 5 unifies scheduling problems for general-time DFGs under various models without computing retiming explicitly. These pipeline schedules corresponds to retiming on general-time DFGs with splitable nodes. This chapter presents a single scheduling algorithm, which can generate *static schedules* under two timing models (integral versus fractional) and two processor models (pipelined versus non-pipelined) without resource constraints. The schedules required by the software-pipelining problem belong to the integral timing model and pipelined-processor model. Therefore, the scheduling algorithm in this chapter can generate schedules directly for the software-pipelining problem. Chapter 6 presents retiming algorithms for general-time DFGs without node splitting to obtain a small class of schedules.

The data-flow graph usually models the innermost loop of a nested loop. For nested loops, Chapter 7 defines *multi-dimensional data-flow graphs* to model the uniform multi-dimensional precedence relations. *Multi-dimensional retiming and unfolding* techniques are defined for such DFG. Integer linear programming forms are derived for pure retiming and its combination with unfolding.

The scheduling of cyclic DFGs with resource constraints is more difficult than scheduling acyclic DFGs with resource constraints or scheduling cyclic DFGs without resource constraints. The loops are usually pipelined in order to increase the execution rate, that is the execution periods of several iterations are overlapped. To pipeline loops with acyclic DFGs, theoretically the performance can be made arbitrarily good. However, when pipelining cyclic DFGs, the cycles confine the depth and freedom of loop pipelining. In contrast to acyclic DFGs, cycles in a DFG provide bounds on the improvement we can achieve by pipelining.

Resource-constrained scheduling algorithms for acyclic data-flow graphs, as introduced in Section 1.1, can be applied to the prologue, repeating schedule, and epilogue after the retiming function is found. However, since the retiming function is found without the knowledge of resource constraints, the obtained schedule may not be satisfactory under certain resource constraints. A novel technique, called *rotation*, is proposed so that it is easier to explore schedules under resource constraints. Chapter 9 derives some theoretical properties, and shows the sufficiency of the rotation technique. Chapter 8 presents a practical algorithm for resource-constrained scheduling for multi-processor systems or data-path designs. This algorithm performs retiming and scheduling when resource constraints are considered. The performance of this algorithm is demonstrated by very good experimental results.

Finally, some concluding remarks are made, and pointers to future work are presented.

1.4 Contributions

The thesis presents the following new results, where $|V|$ is the number of nodes in a DFG and $|E|$ the number of edges. The first half of the results do not concern resource constraints, but the rotation scheduling does.

- The concept of retiming and that of pipelining are equivalent. (Chapter 2)
- The ordering of the applications of retiming and unfolding is immaterial. (Chapter 3)
- Several retiming algorithms with implicit unfolding.
  - Unit-time DFGs. (Chapter 4)
    - To achieve rate-optimality with time complexity $O(|V|\,|E|)$.
  - General-time DFGs – for a given unfolding factor f: (Chapter 6)
    - To achieve a given cycle period with time complexity $O(f|V|\,|E|)$.
    - To find the minimum cycle period with time complexity $O(f^2|V|\,|E| + f^2|V|^2 \log(f\,|V|))$.

- Scheduling general-time DFGs under two timing models (integral versus fractional) and two processor models (pipelined versus non-pipelined). (Chapter 5)
  - An $O((|E| + f)\,|V|)$-time unified (pipeline) scheduling algorithm for all four models.
  - The minimum rate-optimal unfolding factors for all four models.
  - The problem of software pipelining is under the integral timing model and pipelined processor model. Thus, a static schedule for software pipelining can be generated in time $O((|E| + f)\,|V|)$.
- Multi-dimensional data-flow graph for nested loops. (Chapter 7)
  - Multi-dimensional retiming and unfolding
  - An integer linear programming form for multi-dimensional retiming.
  - An integer linear programming form for multi-dimensional retiming and unfolding.
- Scheduling under resource constraints: rotation scheduling.
  - The rotation technique.
  - A (pipeline) scheduling algorithm under resource constraints. (Chapter 8)
  - Theoretical properties of the rotation technique. (Chapter 9).

Chapters 3 to 7 are joint works with Edwin Sha. Chapter 8 is a joint work with Andrea LaPaugh and Edwin Sha. Chapter 9 is a joint work with Andrea LaPaugh. Some of the work in this thesis has been published in [7, 8, 9, 10, 11]. The following grants have supported the research: DARPA/ONR contract N00014-88-K-0459 and MIP90-23542.

Chapter 2

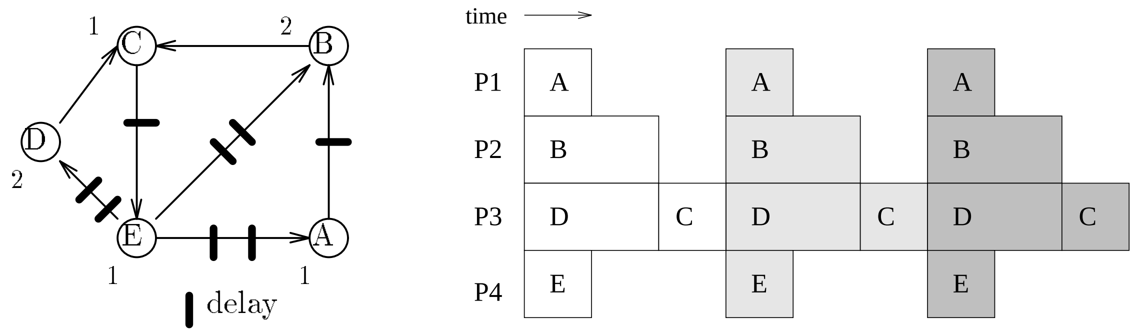
Foundations of Loop Pipelining

2.1 Introduction

This chapter first introduces the data-flow graphs used in this thesis to model iterative algorithms, and then establish the relationship of loop pipelining and some attributes of data-flow graphs. Two design styles to implement a pipeline schedule are introduced in the last section.

In the ordinary pipeline model, a new iteration starts execution every n cycles, where n is called the *initiation interval*. The execution periods of several iterations are overlapped. All iterations of an acyclic data flow graph can be executed simultaneously after proper transformation. Therefore, where there is no resource constraint, the execution rate of an acyclic data-flow graph can be made arbitrarily good. When a data-flow graph is cyclic, the schedule cannot be overlapped arbitrarily. It is not trivial to generate a schedule for a loop pipeline when the data-flow graph is cyclic even without resource constraints.

To resolve the difficulty, instead of looking at the original iteration, we construct the static schedule, which is a new loop body consisting of nodes from different original iterations. In our model, a loop pipeline is composed of a *prologue*, a *repeating schedule*, and an *epilogue*. A repeating schedule, a schedule that is repeatedly executed, forms the new loop body. The length of a repeating schedule corresponds to the initiation interval in the ordinary pipeline model. We would like to construct



(a) DFG G (b) A static schedule

Figure 2.1: A data-flow graph G and its static schedule

a repeating schedule with minimum length. The transition from the initial state of the old loop body to the new loop body is done by the *prologue*. The prologue and epilogue can be easily obtained once a static schedule is found. Under our pipeline model, we can focus on the construction of short repeating schedules.

The *retiming* technique [53] is a convenient and effective tool for the optimization of synchronous systems, due to its nice properties and algorithms. The technique was first designed to optimize systolic arrays, and then used to reduce the clock cycle time of sequential circuits [52, 51]. The technique is also widely used for other circuit-level optimization [22, 55]. For digital-signal-processing synthesis for multi-processor systems, retiming and pipelining are both used to improve performance [12, 23]. However, the relationship between them and the effect of their combination have not been clear. This thesis uses the retiming technique to model the pipelining of iterative algorithms or loops. In Sections 2.5 and 2.6, we will demonstrate that retiming and loop pipelining are essentially the same concept. In fact, we can use retiming to obtain the new loop body under our loop pipeline model.

2.2 Data-Flow Graphs

Definition 2.1 *A data-flow graph (DFG) $G = (V, E, d, t)$ is a node-weighted and edge-weighted directed graph, where V is the set of nodes, $E \subseteq V \times V$ is the set of edges, d is a function from E to the nonnegative integers, and t is a function from V*

```
for i = 1 to n do
  A[i] = E[i-2] + 9;
  B[i] = A[i-1] * E[i-2];
  D[i] = E[i-2] * 30;
  C[i] = B[i] + D[i];
  E[i] = C[i-1] + 5;
end
```

Figure 2.2: A loop program

to the positive integers.

Set V represents the set of computation nodes, and the value $t(v)$ of a node v where $v \in V$ is the computation time of v. Set E represents precedence relations among computation nodes, and $d(e)$ of an edge e equals the number of delays (delay count) on edge e. Let $e = (u, v)$. The data used as input to node v is generated by node u $d(e)$ iterations before. For the graph to be a meaningful data-flow graph, the delay count of any cycle should be nonzero. A data flow graph is called a *unit-time* DFG if the computation times of all nodes are identical; otherwise, it is called a *general-time* DFG. For example, see Figure 2.1-(a), where the numbers attached to nodes are the corresponding computation times. The data-flow model is widely used in many fields, for example, in circuitry[53] and in program descriptions [19, 43, 65, 81], which may give different interpretations of functions d and t in DFG.

For the sake of convenience, we use the following notation. The notation $u \xrightarrow{e} v$ means that e is an edge from node u to node v. The notation $u \overset{p}{\leadsto} v$ means that p is a path from node u to v. The delay count of a path $p = v_0 \xrightarrow{e_0} v_1 \xrightarrow{e_1} v_2 \ldots \xrightarrow{e_{k-1}} v_k$ is $d(p) = \sum_{i=0}^{k-1} d(e_i)$. The total computation time of a path p is $t(p) = \sum_{i=0}^{k} t(v_i)$. Notice that the end-points of a path are included.

We define one *iteration* to be an execution of each node in V exactly once. Inter-iteration data dependencies are represented by weighted edges. An edge e from u to v with delay count $d(e)$ means that the computation of node v at iteration j depends on the computation of node u at iteration $j - d(e)$. The set of edges without delay composes a *directed acyclic graph (DAG)*, which represents data dependencies within the same iteration.

The *cycle period* of a graph can characterize the computation time of one schedule instance without resource constraints.

Definition 2.2 *The* **cycle period** *of a DFG G, denoted by $\mathcal{C}(G)$, is defined as the computation time of the longest path without delay, i.e.*

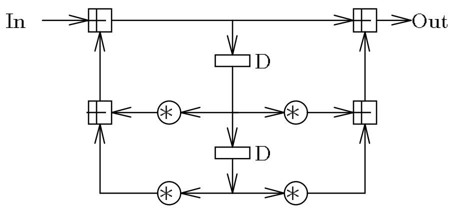
$$\mathcal{C}(G) = \max\{t(p) \mid p \textit{ is a path in } G \textit{ with } d(p) = 0\}.$$

For example, the cycle period of the DFG in Figure 2.1-(a) is 3 from the longest path, which is from node B to C. The cycle period of a DFG corresponds to the minimum schedule length of one iteration when there are no resource constraints. A schedule for G is shown in Figure 2.1-(b), which takes 3 time units (equaling the cycle period of G) to complete an iteration. The circles following node B and D represent that the corresponding processors are still busy. Since such a schedule can be repeatedly executed for every iteration, it is called *a static schedule*. An instance of the static schedule starts execution in every cycle period. In a static schedule, each instance of a node in the DFG is assigned *statically* to the same time step and the same processor for each instance of the schedule. Without loss of generality, it is assumed that the cycle period of a static schedule is non-zero.

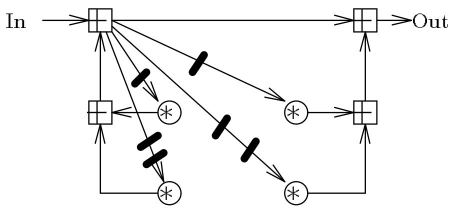
2.2.1 Applications of Data Flow Graphs

Data flow graphs can be interpreted as the loop body of a **for** loop in an imperative program. For example, The DFG in Figure 2.1 represents the loop program in Figure 2.2. Each iteration corresponds to one execution of the loop body.

Data flow graphs are also a natural representation of digital filters, which are usually represented as *signal flow graphs*. A signal flow graph of an IIR filter is shown in Figure 2.3-(a), where the rectangle boxes are called *delays*. The signal flow graph can also be drawn as the data-flow graph in (b) so that each arc has only one head and one tail.



(a) The signal flow graph



(b) The corresponding data flow graph

Figure 2.3: The second order IIR filter

We use the differential equation solver in [67] as an example of using data-flow graphs to model while loops. The behavioral description is in Figure 2.4-(a). We assume that an adder performs additions, subtractions, and comparisons. For the DFG in Figure 2.4-(b), we use boxes to represent adder nodes and circles to represent multiplier nodes. The DAG part, represented by thick arcs in Figure 2.4-(b), is constructed from the loop body and while-loop test. The outputs of Nodes 8, 9, and 6 are the values of $x1$, $y1$, $u1$, respectively. Node 10 corresponds to the condition test $(x < a)$ of the while-loop control, which should precede the loop body of every iteration. Thus, an edge without delay is added from Node 10 to every other node without incoming zero-delay edges. Under this formulation, the control flows and data flows are modeled uniformly, and treated uniformly by our algorithms. The dependencies across iterations are represented by edges with delays to form a cyclic DFG. The cyclic DFG is constructed as follows. From the statement $u = u1$, we know the value $u1$ produced in iteration i is used as value u in iteration $i + 1$. We add an edge with one delay from the node producing $u1$, Node 6, to the nodes using u as an input, Nodes 0 and 5. Similar edges are added from node producing $x1$ to the nodes using x, and from node producing $y1$ to the nodes using y.

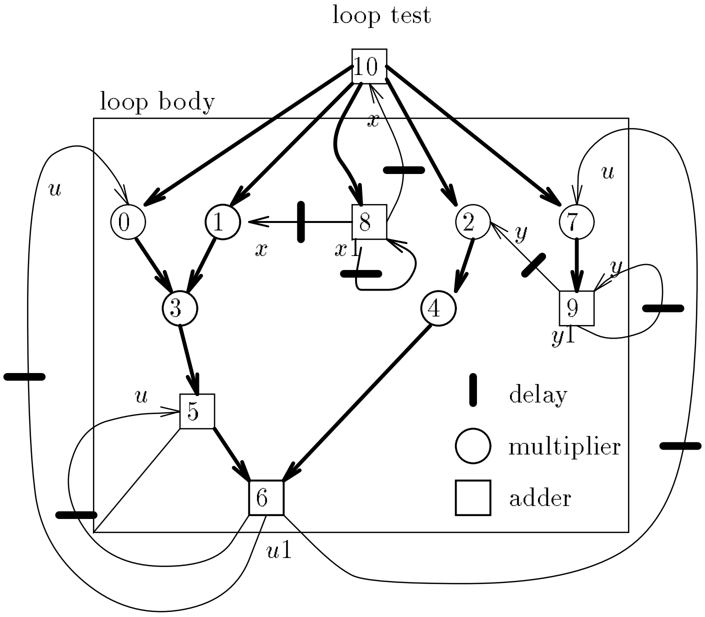
2.2.2 Iteration Period and Iteration Bound

In order to measure the average execution rate, the *iteration period* is defined as the average computation time it takes to execute one iteration. Each cycle in the DFG imposes a lower bound on the iteration period. For a cycle of one delay, the iteration period is at least the total computation time of the cycle. Consider the cycle DCE

To solve $y'' + 3xy' + 3y = 0$.

```
while (x < a) do
    x1 =  x + dx;
    u1 =  u - (3 * x * u * dx)
          -(3 * y * dx);
    y1 =  y + u * dx;
    x = x1;   u = u1;   y = y1;
end
```

(a) The behavioral description



(b) The cyclic data flow graph

Figure 2.4: The differential equation solver

of 3 delays in Figure 2.1. A data item which is emitted from node D takes at least 4 time units to flow through node C, E, and D back to the outgoing edge of node D. During this flow, 3 iterations are computed since the data passes through 3 delays. Therefore, the computation time of 3 iterations is at least 4 time units. The iteration period is at least 4/3. Each cycle gives a lower bound on the iteration period.

This lower bound derived from all cycles of a DFG G is called the *iteration bound* [71].

Definition 2.3 *The* **iteration bound** *of a DFG G, denoted by $\mathcal{B}(G)$, is defined to be the maximum time-to-delay ratio of all cycles, i.e.*

$$\mathcal{B}(G) = \max_{\forall\, cycle\ l\ in\ G} \frac{T(l)}{D(l)},$$

where $T(l)$ is the sum of computation time in cycle l, and $D(l)$ is the sum of delay counts in cycle l.

A cycle which gives the iteration bound is called a *critical cycle,* l_{cr}. For example, the $\mathcal{B}(G)$ in Figure 2.1-(a) is 4/3. A schedule is *rate-optimal* if the iteration period of

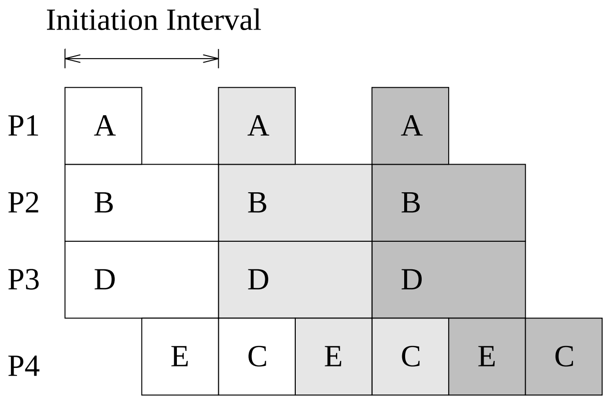


Figure 2.5: A schedule pipelined with initiation interval 2

this schedule equals its iteration bound. When there is no resource constraint and a sufficiently large number of iterations are executed together, there is always a static schedule that can achieve the rate-optimality. Chapter 4 and Chapter 5 will discuss the derivations of rate-optimal schedules for unit-time DFGs and general-time DFGs, respectively.

The value $\mathcal{B}(G)$ can be found in time $O(|V||E|\log|V|)$, when the total number of delays and total computation time are upper bounded by $O(|V|^k)$, where k is a constant [41, 36]. For a unit-time DFG, it takes only time $O(|V||E|)$ to compute the bound $\mathcal{B}(G)$.

2.3 Difficulties of Ordinary Pipeline Model

A schedule is *pipelined* if the subsequent instance of the schedule is *initiated* before the previous instance is finished. The interval at which new instances of a schedule are initiated is called *initiation interval*. For example, the static schedule in Figure 2.1 is initiated at an interval of 3. This schedule cannot be pipelined due to inter-iteration dependencies.

When there are no inter-iteration dependencies in a DFG, all the iterations can be performed simultaneously and an arbitrarily small iteration period can be achieved. But, for a pipeline schedule of a DFG with delays, the inter-iteration dependencies

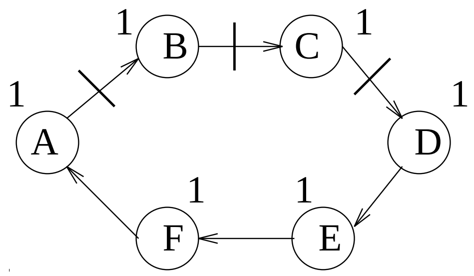


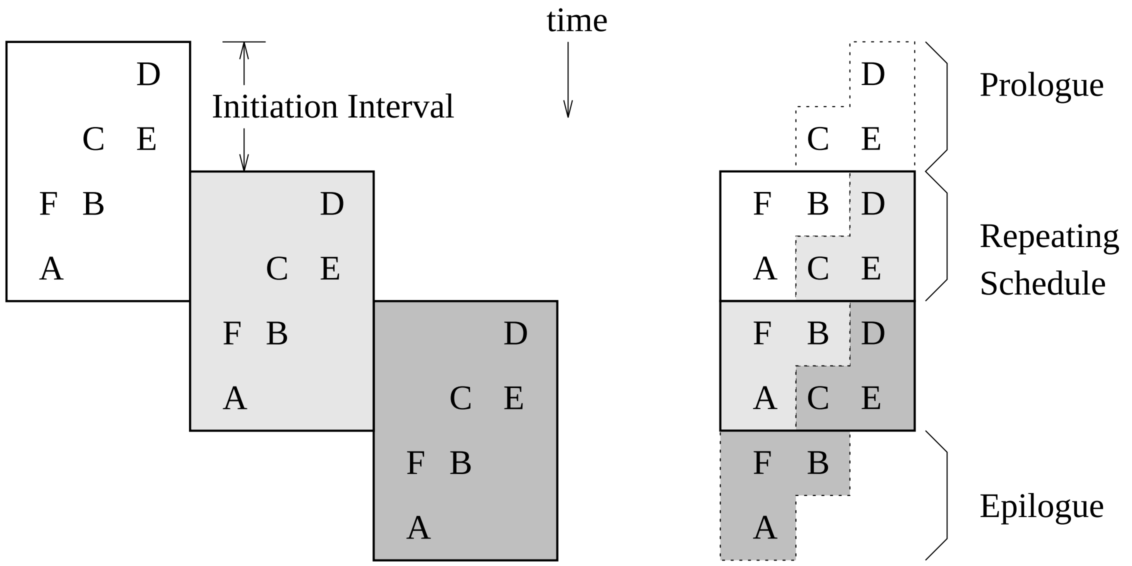
Figure 2.6: A simple DFG

have to be satisfied among overlapped instances of the schedule. For the above-mentioned example, Node C in the first instance must precede Node E in the second instance because the edge from C to E has one delay. Thus, the schedule cannot have an initiation interval of 2 although another schedule of the DFG can achieve such an initiation interval, as shown in Figure 2.5. For a given static schedule, it is not easy to decide the minimum initiation interval allowable by the schedule. For a given initiation interval, it is not easy to find a schedule to achieve that execution rate, either.

For a pipeline with initiation interval n, nodes from time steps separated by n steps will be in execution at the same time in the pipeline. When resource constraints are considered, the difficulty is that these nodes cannot occupy the same resource. For example, Nodes F, B and D are executed simultaneously in the steady state of the pipeline. Thus, in the creation of a pipeline schedule, the designer needs to worry about resource constraints and precedence relations over several iterations. These difficulties make it much harder to generate pipeline schedules for DFGs with inter-iteration precedence relations.

This thesis organizes loop pipelining in a different way such that the above two difficulties can be relieved and the pipeline scheduling of a cyclic DFG is transformed into (nonpipeline) scheduling of a DAG. Instead of looking at the original iteration, we construct another schedule which is a new loop body consisting of nodes from different original iterations.

Let's use the unit-time DFG of a single cycle in Figure 2.6 to illustrate the idea. Figure 2.7-(a) shows a schedule that can be pipelined at an initiation interval of 2 time



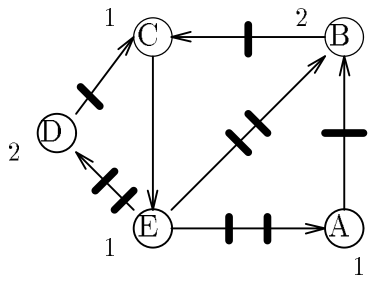
(a) Ordinary model (b) Our model

Figure 2.7: Models of loop pipelines

units. In an alternative organization, in Part (b), we focus on the actual repeating pattern of the pipeline schedule, which is depicted as the rectangle boxes.

In our model, a loop pipeline consists of three parts: *prologue, repeating schedule, epilogue*. The repeating schedule consisting of nodes from different iterations forms a new loop body. The length of the repeating schedule reflects the execution rate of the pipeline in a steady state. The prologue and epilogue are the sequence of operations executed before and after several instances of the repeating schedule. In other words, the prologue brings the computation to the starting state of the repeating schedule; the epilogue carries the terminating state out from the last instance of the repeating schedule.

We will use the so-called *retiming technique* to interpret the loop pipeline under our model such that the prologue, new loop body, and epilogue can be derived from a *retiming function*. After the retiming technique is introduced in the next section, we explain the relationship between loop pipelining and retiming in Section 2.5.



$r(A) = r(B) = r(D) = r(E) = 0$
$r(C) = -1$

Figure 2.8: A retimed DFG G_r

2.4 Retiming Technique and Retimed Graph

The *retiming* technique [53] has been effectively used to obtain the minimum cycle period for a DFG by rearranging the delays. The data-flow graph can be used to model synchronous circuitry, where nodes represent combinational circuit elements, edges represent connectivity, and delays correspond to registers. The clock cycle of the synchronous circuitry equals the cycle period of the DFG. The retiming technique tries to evenly distribute registers such that the retimed circuitry has a smaller clock cycle period. In this thesis, we use data-flow graphs to model behavioral loop descriptions and use the retiming technique to model (software) loop pipelines.

The technique of retiming moves delays around in the following way: a delay is drawn from *each* of the incoming edges of v, and then a delay is pushed to *each* of the outgoing edges of v, or vice versa. Note that the retiming technique still preserves the data dependencies of the original DFG. Only initial assignments may need to be modified after a retiming. The modification of initial assignments is carried out by a *prologue* of statements.

A retiming r of a DFG G is a function from V to the integers. The value $r(v)$ is the number of delays drawn from each of the incoming edges of node v and pushed to each of the outgoing edges. From the DFG in Figure 2.1 to the DFG of Figure 2.8, a delay is pushed through node C in the backward direction. Therefore, the retiming function of node C is -1, and thus in the schedule the i-th copy of node C will be computed with the $(i+1)$-th copies of other nodes.

Let $G_r = (V, E, d_r, t)$ be the DFG retimed by a retiming r of G, where d_r is retimed delay counts of edges described in Property 2.1. A retiming is *legal* if the retimed delay count d_r is nonnegative for every edge in E. For an edge $u \xrightarrow{e} v$, the number of delays increased equals the number of delays pushed into the edge through node u minus the number of delays pushed out of the edge through node v, which is $r(u) - r(v)$. The following properties of d_r are easily obtained :

Property 2.1 *[53] Let G be a DFG and r a retiming.*

(a) *r is a legal retiming if $d_r(e) \geq 0$ for every $e \in E$.*

(b) *For any edge $u \xrightarrow{e} v$, we have $d_r(e) = d(e) + r(u) - r(v)$.*

(c) *For any path $u \overset{p}{\rightsquigarrow} v$, we have $d_r(p) = d(p) + r(u) - r(v)$.*

(d) *For a cycle l, we have $d_r(l) = d(l)$.*

Since the retiming operation preserves the number of delays in a cycle, it also preserves the iteration bound $\mathcal{B}(G)$ of a DFG. Retiming can improve the cycle period of a DFG, but cannot always achieve rate-optimality since the cycle period of a DFG is integral and the iteration bound may be fractional. Even if the iteration bound is integral, sometimes retiming cannot help due to the large computation time of certain node. Consider a cycle consisting of two nodes of computation times 49 and 1. If the number of delays in the cycle is 2, retiming can never achieve the iteration bound, 25. However, retiming can achieve rate-optimality for a unit-time DFG with integral iteration bound. This is a consequence of a theorem shown in Chapter 4.

For a given DFG G, Leiserson and Saxe presented an $O(|V|\,|E| \log |V|)$-time algorithm to find a legal retiming function r such that the cycle period $\mathcal{C}(G_r)$ of the retimed graph G_r is minimized [53].

Note that our definition of retiming functions is slightly different from that by Leiserson and Saxe. In our definition, $r(v)$ is positive if delays are pushed along the direction of edges. We think this definition is more natural, especially in the scheduling problem.

One of the advantages of the retiming technique is that nodes in a cycle and nodes not in any cycle are treated in the same way. From the definition of retiming, an arbitrary amount of delays can be pushed through a node with no incoming edges

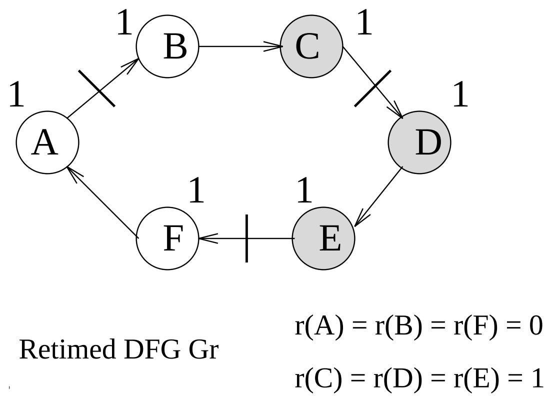


Figure 2.9: Delays are evenly distributed in G_r

to its outgoing edges or pushed through a node with no outgoing edges to its incoming edges. Therefore, an acyclic DFG with delays can be retimed into an acyclic DFG without delays. After attaching the proper prologue and epilogue, an acyclic DFG with delays can also achieve arbitrarily good performance as a DFG without delays.

2.5 Pipelining and Retiming

In this section, we first explain the effect of retiming on pipeline scheduling of loops, and then show how to measure the sizes of prologues and epilogues. To simplify description, we consider a *normalized* retiming r with the property that $\min_v r(v) = 0$. Any retiming function r can be normalized by subtracting $\min_v r(v)$ from $r(v)$ for every v in V. This section is concerned with unit-time DFGs only; the next section will talk about pipelining of general-time DFGs.

Let's use the unit-time DFG G of a single cycle in Figure 2.6 to have a clearer picture on the effect of retiming. We can retime the DFG by evenly distributing the delays over the cycle such that the cycle period of the retimed graph is minimized, as shown in Figure 2.9. Notice that the repeating schedule in Figure 2.7-(b) is a *static schedule* of the retimed DFG G_r, i.e. a schedule of the DAG of G_r.

In fact, every retiming function corresponds to a set of loop pipelines with the same set of nodes in prologue, repeating schedule and epilogue, respectively. When a delay is pushed from the incoming edges of node D to its outgoing edges, every copy of node D is shifted up by one iteration, and the first copy of node D is shifted out of the first iteration into the prologue. The technique of retiming regroups operations in a loop into new iterations, where each operation is executed once in each new iteration. The node v in the new iteration i was actually in the old iteration $i + r(v)$. The cycle period of the retimed graph equals the schedule length of the new loop body without resource constraints.

The *prologue* of a schedule is a sequence of operations which brings the entry status of the original loop to the entry status of the new loop. After the new loop executes the last copy of its loop body, the schedule enters the *epilogue*, which proceeds to finish the computation of the original loop. Whenever a delay is pushed through a node v, a copy of node v is pushed into the prologue. In general, the number of copies of node v appearing in the prologue is the value $r(v)$. Because r is normalized, there exists a node u such that the prologue does not contain any copy of node u. This condition distinguishes the first instance of the static schedule from the prologue since a static schedule contains one copy of every node. An analogous condition distinguishes the last instance of the static schedule from the epilogue. Therefore, the number of copies of node v appeared in the epilogue is $\max_u r(u) - r(v)$. The pipeline is not executed at full speed during the prologue and the epilogue. Hence, the user may wish to measure their size.

We can measure the size of prologue and epilogue from the retiming function, as described below. Let $\mathcal{P}(r)$ be $\max_u r(u)$ for a normalized retiming r. We know that in the prologue there are nodes from the first $\mathcal{P}(r)$ iterations and in the epilogue there are nodes from the last $\mathcal{P}(r)$ iterations. The number of iterations executed at full speed by the static schedule is $n - \mathcal{P}(r)$, where n is the total number of iterations executed. Actually, the union of prologue and epilogue consists of $\mathcal{P}(r)$ iterations of the original loop body. Therefore, from a given retiming, we not only obtain the entry state and the length of the static schedule, but also the length of the prologue and the epilogue.

For the retimed DFG G_r of Figure 2.8, we first normalize the retiming function into $r(A) = r(B) = r(D) = r(E) = 1$, and $r(C) = 0$. Figure 2.5 is a pipeline schedule corresponding to the retiming function r. From the original program in Figure 2.2

```
A[1] = E[3] + 9;   /* Prologue */
B[1] = A[0] * E[-1];
D[1] = E[-1] * 30;
E[1] = C[-1] + 5;
for i = 1 to n-1 do   /* Repeating Schedule */
    A[i+1] = E[i-1] + 9;
    B[i+1] = A[i] * E[i-1];
    D[i+1] = E[i-1] * 30;
    C[i] = B[i] + D[i];
    E[i+1] = C[i] + 5;
end
C[n] = B[n] + D[n];   /* Epilogue */
```

Figure 2.10: The retimed program corresponding to G_r

and the normalized retiming function, we obtain the corresponding retimed program in Figure 2.10. The execution of nodes A, B, D and E brings us to the repeating schedule, and after the last copy of the repeating schedule is executed, the program enters the epilogue, which consists of node C. In the retimed program, there is one iteration in the prologue and epilogue, and the new loop body is executed $n-1$ times. By simple modifications of our retiming algorithms, we can control the size of prologue and epilogue while achieving a given cycle period, as explained in Section 4.3 for unit-time DFGs and at the end of Section 6.1 for general-time DFGs.

The schedule length of the new loop body without resource constraints equals the cycle period of the retimed graph. In order to find a pipeline schedule under resource constraints, we can first find a retiming such that the cycle period is minimized, and then schedule the DAG of the retimed graph under resource constraints by any scheduling algorithm for DAGs.

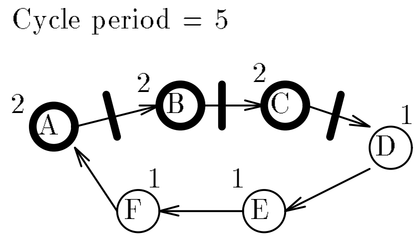
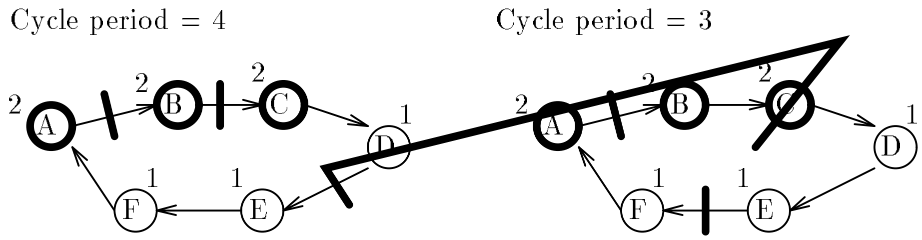


Figure 2.11: A general-time DFG



(a) a retimed DFG (b) a retimed DFG with splitable nodes

Figure 2.12: Retimed general-time DFGs with versus without splitable nodes

2.6 General-Time Data-Flow Graphs with Splitable Nodes

The pipelining of schedules for a general-time DFG does not directly correspond to a retiming on the general-time DFG. We need to model pipeline schedules of a general-time DFG with retiming on the general-time DFG *with splitable nodes*. Each retiming of a general-time DFG still corresponds to a set of pipeline schedules. However, not every pipeline schedule can be represented by a retiming on the corresponding general-time DFG. The general-time DFG with splitable nodes can be considered as a unit-time DFG by replacing each general-time node as a sequence of unit-time nodes.

Consider the general-time DFG G in Figure 2.11. A retiming on G as shown in Figure 2.12-(a) can reduce the cycle period from 5 to 4. A pipeline schedule is shown in Figure 2.13. Since the repeating schedule is obtained from the DAG of the retimed general-time DFG, the execution period of any node does not span across the boundary of the repeating schedule. However, some pipeline schedules do need repeating schedules with the above-mentioned property, such as the pipeline schedule in Figure 2.14. Such pipeline schedules correspond to retimings on general-time DFGs

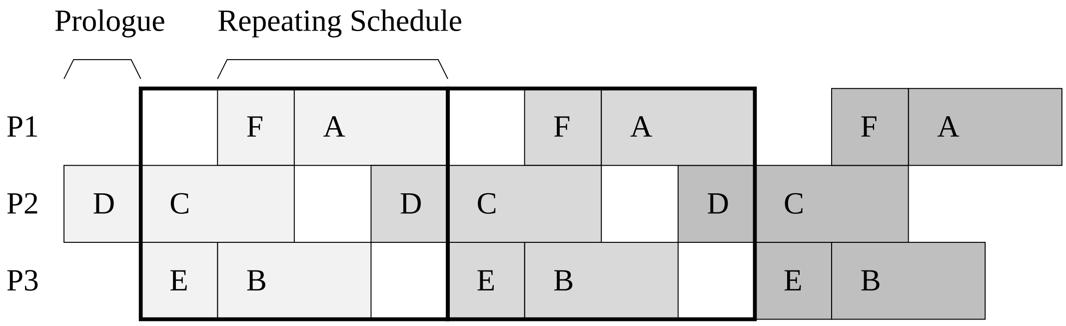


Figure 2.13: A repeating schedule without splitable nodes

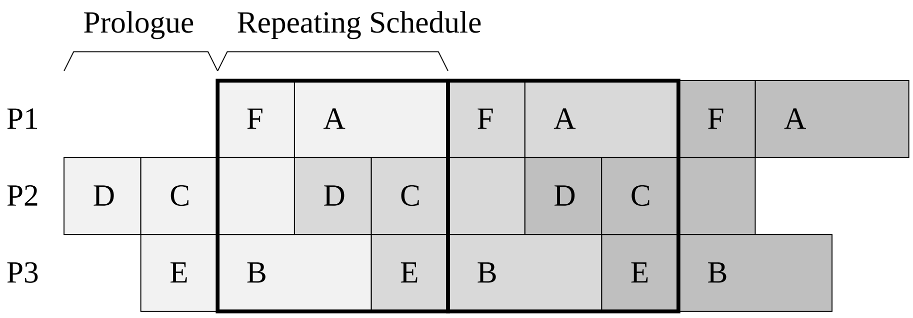


Figure 2.14: A repeating schedule with splitable nodes

with splitable nodes. We can retime the DFG in Figure 2.11 into the one with splitable nodes in Figure 2.12-(b), where node C is split into two one-unit parts. The repeating schedule of Figure 2.14 corresponds to the DAG part of the retimed DFG in part (b). Such splitting of node C in the retimed graph corresponds to a repeating schedule where the execution period of node C starts in one instance of the repeating schedule and ends in the next instance. Since a delay is pushed half-way through node C, the prologue consists of half of node C.

Chapter 4 is concerned with unit-time DFGs. Based on the concept for unit-time DFGs, Chapter 5 produces pipeline schedules for the general-time DFGs by allowing node splitting. The problem of software pipelining is solved under this model. When node splitting is not allowed, the execution period of any node cannot run across the boundary of the repeating schedule. Although the repeating schedules without node splitting are more restrictive, in some applications for multi-processor systems such a repeating schedule can be used as an atomic task. It is also necessary to consider

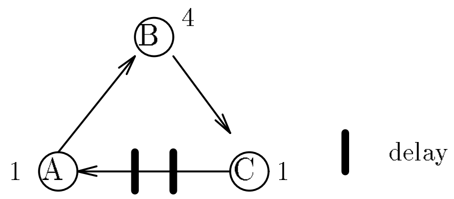


Figure 2.15: An exemplary DFG for implementation styles

the case when node splitting is not allowed because each node in the data-flow graph will be realized by actual hardware. If the hardware that realizes the split node is an off-the-shelf component, we cannot split such a node. Chapter 6 will present retiming algorithms with implicit unfolding for general-time DFGs.

2.7 Pipelined versus Non-Pipelined Design Styles

A static schedule is to be executed repeatedly with the same relative starting time and the same processor assignment for every node. Depending on cycle periods, some schedules need to be implemented by pipelined processors; some may be implemented by non-pipelined processors. For the discussion in this section, it is assumed that a static schedule contains only one iteration of nodes. When the unfolding technique is also applied, a static schedule contains several iterations of nodes. We consider the copies of the same node in an instance of a static schedule to be different nodes. Then, the pipelined and non-pipelined designs are accordingly defined.

For the implementation of static schedules, we consider the following two design styles. Consider the DFG in Figure 2.15.

Pipelined implementation There is no restriction on the scheduling of copies of the same node besides precedence relations. If the second copy of a node starts its execution on the same processor as the first copy before the first copy is finished, we can use pipelined hardware to implement this processor. This model is the same as the concept of software pipelining for parallel compilers [2]. (See Figure 2.16)

Non-pipelined implementation Since the hardware or processors are non-pipelined, the next copy of a node cannot start execution before the previous copy has

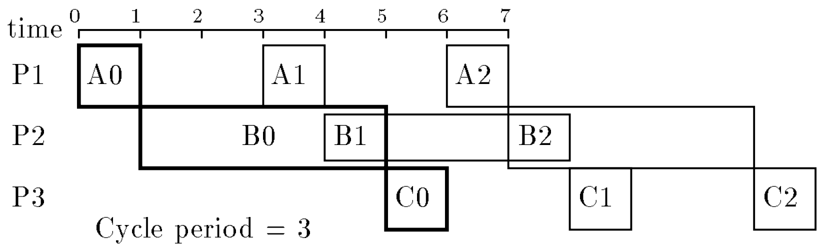


Figure 2.16: Pipelined implementation

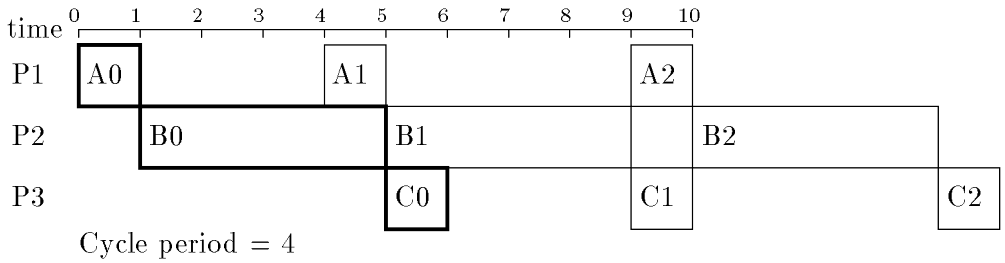


Figure 2.17: Non-pipelined implementation

finished execution. Thus, this constraint causes an implicit precedence relation between these two copies. For every iteration i, the i-th copy of a node has an implicit precedence relation with the $(i+1)$-th copy because they are assigned to the same processor. (See Figure 2.17)

The contrasting properties of pipelined and non-pipelined designs are discussed in Chapter 5. Besides Section 4.4 and Chapter 5, the thesis assumes a non-pipelined design.

Chapter 3

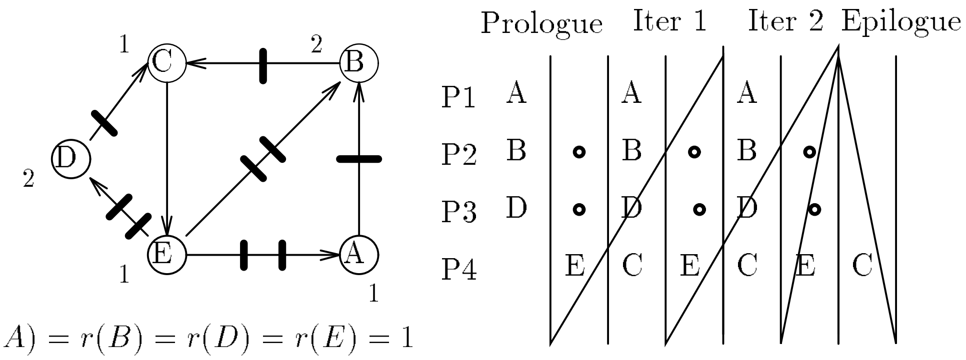
Retiming and Unfolding

3.1 Introduction

Scheduling loops efficiently and optimally is an important problem in VLSI high level synthesis and compilers for parallel machines like very-long-instruction-word (VLIW) or data-flow machines. For example, given a signal flow graph of any digital filter (it may have many cycles), we would like to know how to obtain a rate-optimal schedule such that the resultant synthesized hardware can achieve the highest pipeline rate. We combine two techniques, *retiming* and *unfolding*, to obtain a rate-optimal static schedule without resource constraints. This combined technique turns out to be simple and efficient, and has great potential to be generalized to other problems.

In Chapter 2, we showed that retiming is an equivalent form of loop pipelining, which can improve the cycle period of a DFG. However, retiming cannot always achieve the iteration bound. Another technique called *unfolding* can achieve the iteration bound as long as the so-called *unfolding factor* is large enough. This technique is also called *unrolling* or *unwinding*, and widely used in compiler design [2, 1], high-level synthesis for VLSI design [4, 66], and digital-signal-processing synthesis [12, 23].

A schedule of unfolding factor f can be obtained by unfolding G f times. That is, a total of f iterations are scheduled together, and the schedule is repeated every f iterations. Such a schedule is called *fully-static* from the viewpoint of unfolded graphs in [65], also called *globally-static* from the view point of the original graph [38].



$r(A) = r(B) = r(D) = r(E) = 1$
$r(C) = 0$

(a) a retimed DFG G_r (b) a static schedule

Figure 3.1: A retimed DFG

The DFG G of Figure 2.1 and the retimed DFG of Figure 3.1 will be used as an example in this chapter. The DFG in Figure 3.1-(a) is scheduled globally-statically with every two iterations as in Figure 3.2-(b).

Parhi and Messerschmitt first studied the properties of the unfolded graph and proposed an algorithm to generate the unfolded graph [65]. Complicated formulas were used to describe the unfolded graph. We will derive simpler and easy-to-handle formulas to characterize the relationship between the original DFG and the unfolded DFG. These formulas facilitate the derivation of theorems involving the unfolding technique. Besides a simpler algorithm for the generation of an unfolded graph, we also derive simple techniques to access the unfolded graph without constructing it explicitly.

Unfolding can reduce the cycle period of a DFG down to the iteration period bound to produce a rate-optimal implementation. However, the unfolding process is both time-consuming and space-consuming. In the worst case, the unfolding factor f and the unfolded graph can be very large in order to achieve the target throughput. For example, the unfolding factor can be the least common multiple of all cycle delay counts in order to achieve rate-optimality [65]. If retiming is also applied, the unfolding factor and the cost of implementation can be reduced.

We use the techniques of retiming and unfolding to modify a DFG in order to achieve a better iteration period. This retimed and unfolded DFG can later be input to a resource allocator to achieve resource requirements. One open question was how to efficiently combine these two techniques to minimize the iteration period and unfolding factor. In this chapter, we study some fundamental theorems regarding this combination. Retiming algorithms for achieving the minimum iteration periods are presented in Chapter 4 and Chapter 6 for unit-time DFGs and general-time DFGs, respectively.

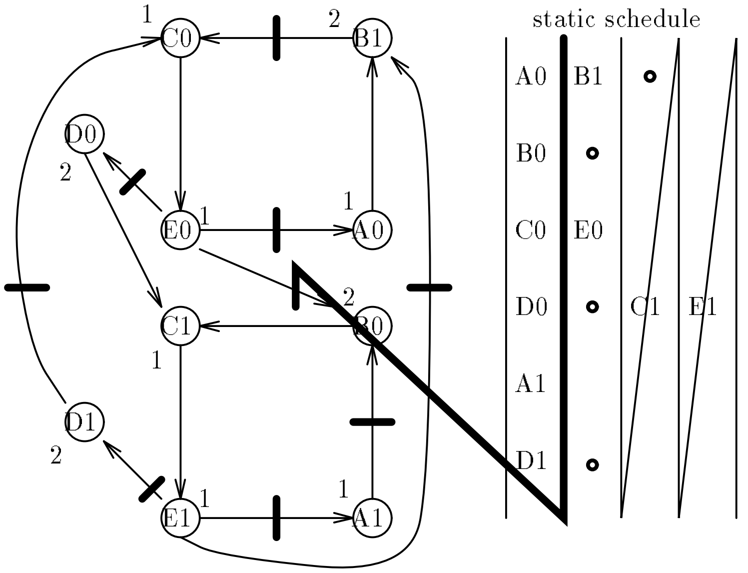
One obvious way to find the minimum iteration period is to unfold the original DFG first, and then do retiming. Instead, we show that we can perform retiming directly on the *original* DFG, and obtain the same minimum iteration period without working on a large unfolded DFG. This convenience and counter-intuitive property is shown in Section 3.5 by proving that the order of retiming and unfolding does not matter for obtaining the minimum iteration period.

3.2 The Unfolding Technique and Unfolded Graphs

This section will characterize the unfolded graph and present an algorithm to generate it. The formulas derived here enable us to study the combination of retiming and unfolding.

We introduce some modular operations to describe properties of the unfolded graphs. Let $\mathbf{Z}$ be the set of integers and $[0, f)$ be the set of integers 0, 1, 2, ..., $f-1$, for a positive integer f. For $a, b \in \mathbf{Z}$, the notation $a \equiv_f b$ means that there exists $n \in \mathbf{Z}$ such that $a - b = n \cdot f$. The operation $a \,\%\, f$ produces a congruent integer within $[0, f)$.

Let f be a positive integer. An unfolded graph with unfolding factor f, denoted G_f, consists of f copies of node set V and represents the same precedence relations in G by delay counts on edges. We say the unfolded DFG $G_f = (V_f, E_f, d_f, t_f)$ is a DFG obtained by unfolding G f times. Set V_f is the union of $V^0, V^1, \ldots, V^{f-1}$. Let u^i be the node u in V^i and the computation time $t_f(u^i) = t(u)$. Figure 3.3 gives the procedure for constructing E_f and d_f. For example, the DFG in Figure 3.2-(a) is the unfolded graph with factor two for the DFG G_r in Figure 3.1-(a). The unfolded graph gives a more global view of the data dependencies with a manageable graph



(a) $G_{r,f}$ with $f = 2$ (b) iteration period = 4/2 =2

Figure 3.2: An unfolded retimed DFG $G_{r,f}$ and a globally-static schedule

size.

We use the subscript f to represent the correspondences between G and G_f. For a node v (resp. edge e) in G, v_f (resp. e_f) represents any copy of v (resp. e) in G_f. For a path p_f in G_f, there is a unique path p in G corresponding to p_f.

One *cycle* in G_f consists of all computation nodes in V_f. The period during which all computations in a cycle are executed is called the *cycle period*. The cycle period $\mathcal{C}(G_f)$ of G_f equals $\max\{t_f(p_f) \mid p_f \text{ is a path in } G_f \text{ with } d_f(p_f) = 0\}$. During a cycle period of G_f, f iterations of G are executed. Thus, the *iteration period* of G_f is equal to $\mathcal{C}(G_f)/f$; in other words, the average computation time for each iteration in G. For the original DFG G, the iteration period is equal to $\mathcal{C}(G)$. There is an algorithm to find $\mathcal{C}(G)$ for a DFG in time $O(|E|)$ [53].

All the dependencies within a cycle are considered intra-cycle dependencies and are represented as edges without delay. The dependency which is from the k-th cycle to the $(k+n)$-th cycle is represented by an edge with n delays. Since f iterations compose one cycle, every f delays in G are represented by one delay in G_f. Therefore, for every $0 \le i, j < f$, an edge e_f in E_f from u^i to v^j with $d_f(e_f)$ delays means that there is a data dependency in G from node u of the i-th iteration to the node v of the

```
for every edge e = (u, v) in E do begin
    δ ⟵ d(e) % f;     ρ ⟵ ⌊d(e)/f⌋;
    for i = 0 to f − δ − 1 do begin
        Add edge e_f = (u^i, v^{i+δ}) to E_f;
        d_f(e_f) ⟵ ρ;
    end
    for i = f − δ to f − 1 do begin
        Add edge e_f = (u^i, v^{i+δ−f}) to E_f;
        d_f(e_f) ⟵ ρ + 1;
    end
end
```

Figure 3.3: The procedure for constructing E_f and d_f

$(d_f(e_f) \cdot f + j)$-th iteration. The data dependency is represented by an edge e from u to v with $d_f(e_f) \cdot f + j - i$ delays in DFG G. Therefore, we have $d(e) = d_f(e_f) \cdot f + j - i$. Every edge $u \xrightarrow{e} v$ has f copies in G_f, which is from u^i to $v^{(d(e)+i)\%f}$ for every $0 \leq i < f$. The value of $d_f(e_f)$ can be written in terms of the floor and ceiling operations on $d(e)$, as shown in Property 3.1-(b) (which follows).

We give some intuition here from simple cases. Consider an edge $u \xrightarrow{e} v$ and its i-th copy incident to u^i. If $d(e) = 0$, there is an edge going from u^i to v^i with no delay. If $i + d(e) < f$, we shift the index of node v by the amount of $d(e)$, so there will be an edge from u^i to $v^{i+d(e)}$. When $f \leq i + d(e) < 2\,f$, the index of node v is shifted past $f-1$ and circularly back to 0, so the index of v is $(i + d(e))\,\%\,f$, and one delay is added to the corresponding edge. In general, the delay number on an edge of the unfolded graph is the number of circular shifts from v^{f-1} to v^0, the index of v is the residue of the circular shifts by the amount $d(e)$ starting from the index i.

Let e be the edge from u to v in G, and e^i from u^i to $v^{(d(e)+i)\%f}$ be the i-th copy of e. By the formula $d_f(e^i) = (d(e) - j + i)/f$, the summation of $d_f(e^i)$ for every e^i is

$$\sum_{i=0}^{f-1} d_f(e^i) \;=\; d(e) + \frac{1}{f}\Big(\sum_{i=0}^{f-1} i - \sum_{i=0}^{f-1}((d(e)+i)\,\%\,f)\Big)$$

$$= \quad d(e) + \frac{1}{f}(\sum_{i=0}^{f-1} i - \sum_{i=0}^{f-1} i) \; = \; d(e).$$

Therefore, the total number of delays in graph G_f is the same as that in G. To sum up, we have the following properties for the unfolded graph.

Property 3.1 *Let u and v be nodes in G and $u \xrightarrow{e} v$.*

(a) *For any $0 \leq i < f$ and $0 \leq j < f$, there is an edge $u^i \xrightarrow{e_f} v^j$ in G_f if and only if*

$$d(e) = d_f(e_f) \cdot f + j - i.$$

(b) *For any $0 \leq i < f$, $0 \leq j < f$ and $j - i \equiv_f d(e)$, there is an edge $u^i \xrightarrow{e_f} v^j$ in G_f, and the delay count*

$$d_f(e_f) = \begin{cases} \left\lfloor \frac{d(e)}{f} \right\rfloor & \text{if } j \geq i, \\ \\ \left\lceil \frac{d(e)}{f} \right\rceil & \text{otherwise.} \end{cases}$$

(c) *The f copies of edge e in G_f are the set of edges $u^i \xrightarrow{e^i} v^{(i+d(e))\% f}$ for every $0 \leq i < f$.*

(d) *The total number of delays of the f copies of edge e is $d(e)$, i.e. $\sum_{i=0}^{f-1} d_f(e^i) = d(e)$.*

These properties can be easily extended to paths by substituting p for e. And, we know that each path in G has exactly f copies in G_f. These properties will be useful in deriving other properties and in designing the algorithms which we present in this thesis.

The unfolding transformation does not change the infinite graph generated; it only changes the level of abstractions used to describe the infinite graph. Therefore, the iteration bound $\mathcal{B}(G)$ is still a lower bound of the iteration period of the unfolded graph. An unfolding factor f is referred to as *rate-optimal* if there is a (pipeline) schedule of f iterations with cycle period c such that $c/f = \mathcal{B}(G)$. Parhi and Messerschmitt [65] showed that if the unfolding factor is the least common multiple of the delay counts of all cycles, a rate-optimal schedule can be achieved. The *minimum* rate-optimal unfolding factor for a unit-time DFG is obtained in Chapter 4. For general-time DFGs, Chapter 5 will derive the *minimum* rate-optimal unfolding factors for various models.

3.3 Simulation of Unfolded Graphs

When the DFG is represented as adjacency-lists, f times as much space and time are needed for the unfolded DFG; when the adjacency matrix is used, f^2 times as much space and time are needed. Since the unfolded DFGs are well-structured, we can derive the properties of the unfolded graphs without actually constructing them.

First, we derive the characterization of the cycle period $\mathcal{C}(G_f)$ in terms of the original DFG G. Then, we present simple techniques to access an unfolded DFG from the original DFG G. That is, we can decide the adjacency relation between nodes and the number of delays on an edge by examining the original DFG directly. Finally, we present an algorithm to compute the cycle period $\mathcal{C}(G_f)$ without constructing G_f.

The cycle period $\mathcal{C}(G)$ can alternatively be expressed as the following.

Lemma 3.3.1 [53] *Let G be a DFG, c a cycle period, and f an unfolding factor. The cycle period $\mathcal{C}(G) \leq c$ if and only if for every path p in G if $d(p) = 0$, then $t(p) \leq c$.*

This characterization was used to derive retiming algorithms in [53].

The following lemma expresses the cycle period of the unfolded graph in terms of the *original DFG*, instead of the unfolded graph. This characterization will be used to derive retiming algorithms with *implicit unfolding*. We show that the cycle period in the unfolded graph G_f is the maximum total computation time among all paths where the total delay count is less than f in the original graph G.

Lemma 3.3.2 *Let G be a DFG, c a cycle period, and f an unfolding factor.*

(a) *$\mathcal{C}(G_f)$ is equal to $\max\{t(p) \mid p$ is a path in G with $d(p) < f\}$.*

(b) *$\mathcal{C}(G_f) \leq c$ if and only if for every path p in G if $t(p) > c$ then $d(p) \geq f$.*

Proof: Let node u^i in G_f be the ith copy of the node u in G. Thus, we know

$$\mathcal{C}(G_f) \;=\; \max\{t_f(p_f) \mid p_f \text{ is a path in } G_f \text{ with } d_f(p_f) = 0\}$$

Procedure `Cycle_Period_0`(G_f)
Input : the actual unfolded DFG G_f.
Output : Cycle period $\mathcal{C}(G_f)$.
Step 0. $L(v_f) \longleftarrow 0$ for every v_f in V_f.
  Queue $\longleftarrow \emptyset$;
Step 1. /* Compute $N(v_f)$: the number of outgoing zero-delay edges of each node */
  for every node v_f in V_f **do**
    $N(v_f) \longleftarrow 0$;
  for every edge $e_f = (v_f, w_f)$ in E_f **do**
    if $d_f(e_f) = 0$ **then** $N(v_f) \longleftarrow N(v_f) + 1$;
Step 2. /* Queue $\longleftarrow$ the set of sink nodes in the DAG */
  for every node v_f in V_f **do**
    if $N(v_f) = 0$ **then** Push v_f to Queue;
Step 3. **while** Queue $\neq \emptyset$ **do begin**
    Pop v_f from Queue;
    $L(v_f) \longleftarrow t_f(v_f) + \max\{L(w_f) \mid v_f \xrightarrow{e_f} w_f \text{ with } d_f(e_f) = 0\}$;
    for every edge $e_f = (u_f, v_f)$ in E_f **do begin**
      $N(u_f) \longleftarrow N(u_f) - 1$;
      if $N(u_f) = 0$ **then** Push u_f to Queue;
    end ;
  end ;
Step 4. $\mathcal{C}(G_f) \longleftarrow \max\{L(u_f) \mid \forall u_f \in V_f\}$.

Figure 3.4: An algorithm to compute the cycle period of G_f

$$= \max\{t_f(p^i) \mid p^i \text{ is a path from } u^i \text{ to } v^j \text{ with } d_f(p^i) = 0$$
$$\text{for every } u^i \text{ and } v^j \text{ in } V_f \text{ where } 0 \leq i, j < f\}.$$

Let p be the path from u to v in G corresponding to the path p^i in G_f. Since we know $d(p) = d_f(p^i) \cdot f + j - i$, we conclude $d(p)$ is equal to $j - i$. From the facts $-f < j - i < f$ and $d(p) \geq 0$, we have $0 \leq d(p) < f$. Thus, Part (a) is proved due to $t(p) = t_f(p_f)$. Part (b) is easily derived from Part (a). □

The cycle period $\mathcal{C}(G)$ is the length of the longest zero-delay path in G. Intuitively, if we call each zero-delay path a segment, we can say that $\mathcal{C}(G_f)$ is the length of the longest f consecutive segments in G.

Instead of actually unfolding the DFG, which consumes both time and space, we simulate the unfolded DFG on the original DFG G, and design a procedure `Cycle_Period`$(G,\ f)$ to compute the cycle period of G_f, $\mathcal{C}(G_f)$, in $O(f\,|E|)$ time. This procedure will be used in Section 6.3 to design an efficient retiming algorithm.

First, we write the algorithm to be simulated in Figure 3.4 to compute the cycle period $\mathcal{C}(G_f)$ with the actual unfolded graph G_f as an input. Let $L(v_f)$ be the longest zero-delay path from node v_f to any node in G_f. Leiserson and Saxe [53] presented an algorithm to compute $\mathcal{C}(G_f)$ in $O(f\,|E|)$ time, which performs a topological sort of the DAG of G_f, and then computes the values $L(v_f)$ in the topological order of nodes. In Figure 3.4, the procedure computes $\mathcal{C}(G_f)$ of G_f by slightly modifying their algorithm so that the topological sort of the DAG of G_f is performed simultaneously with the computation of the function L. Procedure Cycle_Period_0(G_f) can compute $\mathcal{C}(G_f)$ in $O(f\,|E|)$ time on the actual unfolded graph G_f.

The following lemma shows that we can decide whether an *incoming* edge or an *outgoing* edge of v^j in E_f is an edge without any delay by simply checking the sign of the value $j - d(e)$ or $i + d(e) - f$, respectively.

Lemma 3.3.3 *Let v^j be a node in G_f, where $0 \le j < f$.*

(**a**) *There is an edge $u \xrightarrow{e} v$ in G and $j - d(e) \ge 0$ if and only if there is an edge $u^{j-d(e)} \xrightarrow{e_f} v^j$ in G_f and $d_f(e_f) = 0$.*

(**b**) *There is an edge $v \xrightarrow{e} w$ in G and $j + d(e) < f$, if and only if there is an edge $v^j \xrightarrow{e_f} w^{j+d(e)}$ in G_f and $d_f(e_f) = 0$.*

Proof: This lemma can be easily derived from Property 3.1. □

The procedure `Cycle_Period`, as shown in Figure 3.5, uses Lemma 3.3.3 to access the unfolded graph G_f to compute $L(v_f)$. Although the asymptotical time complexity of Procedure Cycle_Period$(G,\ f)$ is the same as the original algorithm by Leiserson and Saxe. The procedure saves a programmer the trouble of constructing an unfolded graph stored in linked-list form, instead of matrix form, in order to achieve the $O(f\,|E|)$-time complexity.

Procedure `Cycle_Period`(G,f)
Input : DFG G, unfolding factor f.
Output : Cycle period $\mathcal{C}(G_f)$.
Step 0. $L(v^j) \longleftarrow 0$ for every $v \in V$ and $0 \le j < f$;
  `Queue` $\longleftarrow \emptyset$;
Step 1. /* Compute $N(v_f)$: the number of outgoing zero-delay edges */
  for every v^j where $v \in V$ and $0 \le j < f$ **do**
    $N(v^j) \longleftarrow 0$;
  for every edge $e = (v, w)$ in E **do**
    for every j where $0 \le j < f - d(e)$ **do** /* $d_f(e_f) = 0$ */
      $N(v^j) \longleftarrow N(v^j) + 1$;
Step 2. /* `Queue` $\longleftarrow$ the set of sink nodes in the DAG of G_f */
  for every v^j where $v \in V$ and $0 \le j < f$ **do**
    if $N(v^j) = 0$ **then** Push v^j to `Queue`;
Step 3. **while** `Queue` $\neq \emptyset$ **do begin**
    Pop v^j from `Queue`;
    $L(v^j) \longleftarrow t(v) + \max\{L(w^{j+d(e)}) \mid v \xrightarrow{e} w \text{ with } j + d(e) < f\}$;
    for every edge $u \xrightarrow{e} v$ in E **do**
      if $j - d(e) \ge 0$ **then begin** /* $d_f(e_f) = 0$ */
        $i \longleftarrow j - d(e)$; $N(u^i) \longleftarrow N(u^i) - 1$;
        if $N(u^i) = 0$ **then** Push u^i to `Queue`;
      end ;
  end ;
Step 3. $\mathcal{C}(G_f) \longleftarrow \max\{L(v^j) \mid \forall v \in V \text{ and } 0 \le j < f\}$.

Figure 3.5: An algorithm to compute the cycle period of G_f with implicit unfolding

3.4 Program Generation

This section describes the generation of the unfolded retimed program corresponding to the example DFG $G_{r',f}$ of Figure 3.9. The generation of retimed programs is described in Section 2.5. The corresponding retimed program of $G_{r'}$ is shown in Figure 3.6. The number of iterations in the prologue and epilogue is $\mathcal{P}(r)$, which is $\max_v r(v) - \min_v r(v)$.

The program for the unfolded retimed DFG $G_{r',f}$ is shown in Figure 3.7. The retimed loop body of $G_{r'}$ is unfolded by the unfolding factor 2 to form the loop body of $G_{r'f}$, which corresponds to the static schedule in Figure 3.9. After retiming by r', the number of iterations to be executed as the static schedule is $\mathsf{n} - \mathcal{P}(r)$, where n is the total number of iterations to be executed in the original loop program. If $\mathsf{n} - \mathcal{P}(r)$ is divisible by f, there is no epilogue due to the unfolding. Otherwise, the number of iterations executed as the unfolded static schedule is $\left\lfloor \frac{\mathsf{n} - \mathcal{P}(r')}{f} \right\rfloor$, and there will be an epilogue consisting of $\mathsf{n} - \mathcal{P}(r') - \left\lfloor \frac{\mathsf{n} - \mathcal{P}(r')}{f} \right\rfloor$ iterations of retimed loop body in $G_{r'}$.

3.5 The Combination of Retiming and Unfolding

For a DFG G, let $G_{r,f}$ be the DFG obtained by unfolding the retimed DFG G_r with factor f, and G_{f,r_f} be the DFG obtained by retiming the unfolded DFG G_f with function r_f, which is a retiming from V_f to $\mathbf{Z}$. Since both retiming and unfolding do not change the iteration bound of a DFG, the iteration period $\mathcal{B}(G)$ is still a lower bound on the iteration periods for both $G_{r,f}$ and G_{f,r_f}.

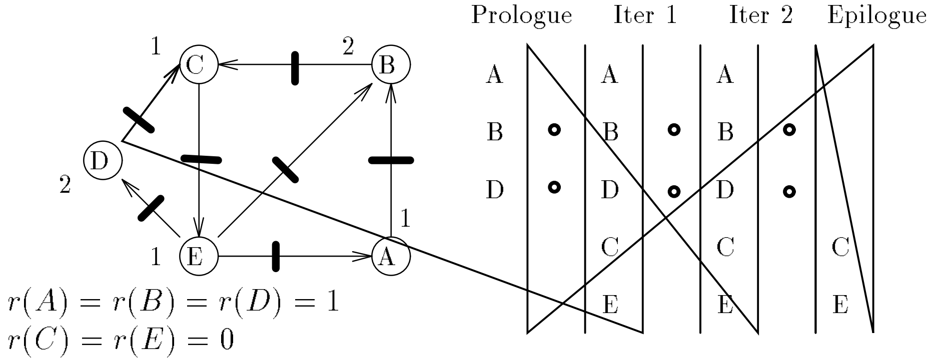
We define the minimum cycle period, denoted by $MCP(f)$, under an unfolding factor f as $\min_{r_f} \mathcal{C}(G_{f,r_f})$, which is the minimum cycle period achieved by unfolding G with factor f. In this section, we show that the order of retiming and unfolding is not important because $MCP(f) = \min_r \mathcal{C}(G_{r,f})$, which is the minimum cycle period achieved by retiming first, and then unfolding. In Chapter 6, an algorithm is designed to find a retiming r such that DFG $G_{r,f}$ has the minimum cycle period $MCP(f)$ under factor f. One of our goals is to find an unfolding factor such that the iteration period $MCP(f)/f$, denoted by $MIP(f)$, is the minimum.

```
/* Prologue */
A[1] = E[3] + 9;
B[1] = A[0] * E[-1];
D[1]= E[-1] * 30;
/* Repeating Schedule */
for i = 1 to n-1 do
    A[i+1] = E[i-1] + 9;
    B[i+1] = A[i] * E[i-1];
    D[i+1] = E[i-1] * 30;
    C[i] = B[i] + D[i];
    E[i] = C[i-1] + 5;
end
/* Epilogue */
C[n] = B[n] + D[n];
E[n] = C[n-1] + 5;
```

Figure 3.6: The retimed program corresponding to $G_{r'}$

```
/* Prologue from retiming */
A[1] = E[3] + 9;
B[1] = A[0] * E[-1];
D[1]= E[-1] * 30;
/* Repeating Schedule */
for i = 1 to f · ⌊(n-1)/f⌋ by 2 do
    A[i+1] = E[i-1] + 9;
    B[i+1] = A[i] * E[i-1];
    D[i+1] = E[i-1] * 30;
    C[i] = B[i] + D[i];
    E[i] = C[i-1] + 5;
    A[i+2] = E[i] + 9;
    B[i+2] = A[i+1] * E[i];
    D[i+2] = E[i] * 30;
    C[i+1] = B[i+1] + D[i+1];
    E[i+1] = C[i] + 5;
end
/* Epilogue from unfolding */
for i = f · ⌊(n-1)/f⌋ + 1 to n - 1 do
    A[i+1] = E[i-1] + 9;
    B[i+1] = A[i] * E[i-1];
    D[i+1] = E[i-1] * 30;
    C[i] = B[i] + D[i];
    E[i] = C[i-1] + 5;
end
/* Epilogue from retiming */
C[n] = B[n] + D[n];
E[n] = C[n-1] + 5;
```

Figure 3.7: The unfolded retimed program corresponding to $G_{r',f}$



(a) DFG $G_{r'}$ retimed by r' (b) iteration period = 2

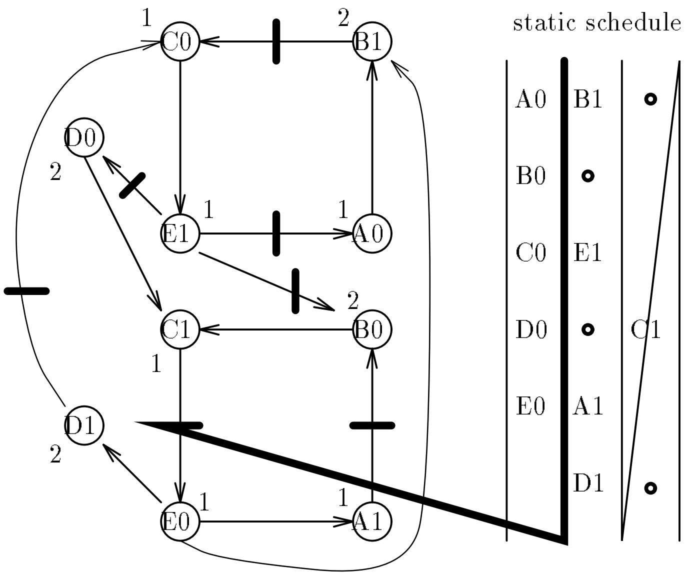
Figure 3.8: A DFG retimed by another retiming r'

Figure 3.1 is a retimed graph G_r with iteration period 2. Without unfolding, the retiming r achieves $MCP(1)$. Figure 3.2-(a) is a result of unfolding DFG G_r by a factor of 2. The cycle period of $G_{r,f}$ is 4 and its iteration period is still 2. The retiming r cannot achieve $MCP(2)$ even though it achieves $MCP(1)$. The retiming r', shown in Figure 3.8, can achieve $MCP(1)$ and $MCP(2)$. The unfolding of $G_{r'}$ by factor 2 produces a static schedule with cycle period 3 in Figure 3.9. This example shows that a retiming r achieving $MCP(1)$ may not achieve $MCP(2)$. Therefore, it is not obvious how to design efficient algorithms to find a retiming achieving $MCP(f)$ without physically unfolding a DFG. We will design algorithms to find retiming functions for given unfolding factors in Chapter 4 and Chapter 6.

Next, the relationship between G_{f,r_f} and $G_{r,f}$ for a fixed unfolding factor is explored. Intuitively, it seems that the DFG G_{f,r_f} provides finer retimings on the unfolded node set and gives better flexibility of retiming. However, we show that the order of retiming and unfolding is immaterial. More precisely, we prove the equality $MCP(f) = \min_r \mathcal{C}(G_{r,f}) = \min_{r_f} \mathcal{C}(G_{f,r_f})$.

The following lemma says that any cycle period which is obtained by retiming the unfolded graph G_f can be achieved by retiming on the original graph G directly, while Lemma 3.5.3 proves the counterpart.

Lemma 3.5.1 *Let G be a DFG, c a cycle period and f an unfolding factor. For any legal retiming r_f on the unfolded graph G_f such that $\mathcal{C}(G_{f,r_f}) \leq c$, there exists a legal retiming r on the original graph G such that $\mathcal{C}(G_{r,f}) \leq c$.*



(a) DFG $G_{r',f}$ with $f = 2$ (b) iteration period = 3/2 = 1.5

Figure 3.9: An unfolded retimed DFG $G_{r',f}$

Proof: Assume that r_f is a legal retiming from V_f to $\mathbf{Z}$ such that $\mathcal{C}(G_{f,r_f}) \leq c$. Let r be a retiming from V to $\mathbf{Z}$ and choose $r(u) = \sum_{i=0}^{f-1} r_f(u^i)$ for every $u \in V$, where u^i is the i-th copy of node u in V_f. We show that r is a legal retiming and $\mathcal{C}(G_{r,f}) \leq c$.

Since r_f is a legal retiming, it is easy to show that r is also a legal retiming. We know that r_f is a legal retiming, so $r_f(v_f) - r_f(u_f) \leq d_f(e_f)$ for every $u_f \xrightarrow{e_f} v_f$. From Property 3.1-(c), the edge $u \xrightarrow{e} v$ in G has f copies in G_f, which are $u^i \xrightarrow{e^i} v^{(i+d(e))\%f}$ where $0 \leq i < f$. Since we know $\sum_{j=0}^{f-1} r_f(v^j) = \sum_{i=0}^{f-1} r_f(v^{(i+d(e))\%f})$, we can derive

$$\begin{aligned} r(v) - r(u) &= \sum_{j=0}^{f-1} r_f(v^j) - \sum_{i=0}^{f-1} r_f(u^i) = \sum_{i=0}^{f-1} (r_f(v^{(i+d(e))\%f}) - r_f(u^i)) \\ &\leq \sum_{i=0}^{f-1} d_f(e^i) = d(e). \end{aligned}$$

The last equation comes from Property 3.1-(d).

Now, we prove that $\mathcal{C}(G_{r,f}) \leq c$. We will derive the properties of a path p in G from the properties of a corresponding path p_f in G_f. Since $\mathcal{C}(G_{f,r_f}) \leq c$, from Lemma 3.3.1, we know for every $p_f \in G_{f,r_f}$ if $t_f(p_f) > c$ then $d_{f,r_f}(p_f) \geq 1$. Since $d_{f,r_f}(p_f) = d_f(p_f) + r_f(u_f) - r_f(v_f)$ for $u_f \overset{p_f}{\rightsquigarrow} v_f$, we know if $t_f(p_f) > c$ then

$r_f(v_f) - r_f(u_f) \leq d_f(p_f) - 1$. Consider a path p^i from u^i to v^j with $t_f(p^i) > c$ for any $0 \leq i < f$. The path p from u to v in G is the corresponding path of p^i, and we have $j = (i + d(p))\%f$. Since $t_f(p^i) > c$, we have $r_f(v^{(i+d(p))\%f}) - r_f(u^i) \leq d_f(p^i) - 1$ for every $0 \leq i < f$. By summing up these f inequalities, we have

$$\begin{aligned} r(v) - r(u) &= \sum_{i=0}^{f-1}(r_f(v^{(i+d(p))\,\%\,f}) - r_f(u^i)) \\ &\leq \sum_{i=0}^{f-1}(d_f(p^i) - 1) \\ &= \sum_{i=0}^{f-1} d_f(p^i) - f = d(p) - f. \end{aligned}$$

The last equality comes from extending Property 3.1-(d) to paths. Therefore, we know for every path p in G if $t(p) > c$ then $r(v) - r(u) \leq d(p) - f$. Thus, from Lemma 3.3.2, r is a legal retiming of G such that $\mathcal{C}(G_{r,f}) \leq c$. □

The following lemma proves that G_f and $G_{r,f}$ are structurally isomorphic, which means that there is a 1-1 correspondence among edges and nodes of the two graphs, and the mapping is a circular shift of every copy of node v in V by the amount $r(v)$. Actually, this lemma gives a way of constructing DFG $G_{r,f}$ directly from G with given r and f, instead of constructing G_r first and then unfolding it.

We consider that G_f and $G_{r,f}$ have the same node set, denoted V_f, but different edge sets, denoted E_f and $E_{r,f}$ respectively, and different delay functions, denoted d_f and $d_{r,f}$ respectively.

Lemma 3.5.2 *Let $G = (V, E, d, t)$ be a DFG, r a retiming on G and f an unfolding factor. The unfolded graph $G_f = (V_f, E_f, d_f, t_f)$ and the unfolded retimed graph $G_{r,f} = (V_f, E_{r,f}, d_{r,f}, t_f)$ have the following relation: there is an edge from u^i to v^j in G_f iff there is an edge from $u^{(i-r(u))\%f}$ to $v^{(j-r(v))\%f}$ in $G_{r,f}$.*

Proof: Consider an edge $e_{r,f}$ from $u^{(i-r(u))\%f}$ to $v^{(j-r(v))\%f}$ in $G_{r,f}$. From Property 3.1, this edge corresponds to edge $u \xrightarrow{e} v$ in G_r, iff $d_r(e) \equiv_f ((j-r(v))\%f)-((i-r(u))\%f)$. Since $d_r(e) = d(e) + r(u) - r(v)$, the modular equation is equivalent to $d(e) \equiv_f j - i$, which means that there is an edge from u^i to v^j. Thus, the if part is derived, and the only if part can be similarly proved. □

The above lemma also holds for the corresponding paths in G_f and $G_{r,f}$. For the structurally equivalent graphs G_{f,r_f} and $G_{r,f}$, we show that there is certain correspondence between the two retimings r_f and r.

Lemma 3.5.3 *Let G be a DFG, c a cycle period and f an unfolding factor. For any legal retiming r on the DFG G such that $\mathcal{C}(G_{r,f}) \le c$, there exists a legal retiming r_f on the unfolded graph G_f such that $\mathcal{C}(G_{f,r_f}) \le c$.*

Proof: Let r be a retiming from V to $\mathbf{Z}$ such that $\mathcal{C}(G_{r,f}) \le c$. From Lemma 3.5.2, we know that G_f and $G_{r,f}$ are structurally equivalent, and there is an edge from u^i to v^j in G_f iff there is an edge from $u^{(i-r(u))\%f}$ to $v^{(j-r(v))\%f}$ in $G_{r,f}$.

We want to prove that there exists a retiming r_f such that G_{f,r_f} is equivalent to $G_{r,f}$, that is, each pair of corresponding edges have the same delay count. Let e_f and $e_{r,f}$ be a pair of corresponding edges for $u^i \xrightarrow{e_f} v^j$ and $u^{(i-r(u))\%f} \xrightarrow{e_{r,f}} v^{(j-r(v))\%f}$. We want to find a retiming r_f to satisfy the equations $d_{f,r_f}(e_f) = d_{r,f}(e_{r,f})$ for every e_f in G_f. Since $d_{f,r_f}(e_f) = d_f(e_f) + r_f(u^i) - r_f(v^j)$, we derive $r_f(u^i) - r_f(v^j) = d_{r,f}(e_{r,f}) - d_f(e_f)$. Thus, if the following linear system has an integer solution, a retiming r_f is found. For every $u^i \xrightarrow{e_f} v^j$ in G_f and $u^{(i-r(u))\%f} \xrightarrow{e_{r,f}} v^{(j-r(v))\%f}$ in $G_{r,f}$,

$$r_f(u^i) - r_f(v^j) = d_{r,f}(e_{r,f}) - d_f(e_f),$$

where $r_f(v_f)$ for $v_f \in G_f$ are unknown variables and $d_{r,f}(e_{r,f})$ and $d_f(e_f)$ are constants. We show that the linear system is consistent and has an integer solution r_f. That is, the value of $r_f(u^i) - r_f(v^j)$ has a consistent value from any linear combination of equations in this system.

We represent this linear system as a graph $H = (V_f, E_f \cup \bar{E}_f, w)$ where $\bar{E}_f$ is the set of reversed edges $\{(v^j, u^i) \mid \text{ for every } (u^i, v^j) \in E_f\}$ and w is a weight function from $E_f \cup \bar{E}_f$ to the integers. For an edge e_f in E_f, the weight equals the righthand side of the corresponding equality, i.e. $d_{r,f}(e_{r,f}) - d_f(e_f)$. The weight of the reversed edge equals the negative of $w(e_f)$. The weight of a path p in H, denoted $w(\mathsf{p})$, equals the total weight of edges in this path. Thus, any linear combination of the equations such that the left-hand side equals $r_f(u^i) - r_f(v^j)$ corresponds to a path p from u^i to v^j in H and the righthand side equals the weight of the path p, i.e. $w(\mathsf{p})$. Notice that a path p in H corresponds to a path in G_f without regarding the directions of arcs; when an edge in G_f is traveled in the reverse direction, the coefficient of that equation in the linear combination is -1. Thus, the linear system is consistent if for every u^i and v^j in V_f all paths p from u^i to v^j in H have the same weight.

We prove that the value of $w(\mathsf{p})$ where $u^i \overset{\mathsf{p}}{\leadsto} v^j$ depends only on nodes u^i and v^j. Consider an edge $u^i \xrightarrow{e_f} v^j$ in E_f and the corresponding edge $u^{(i-r(u))\%f} \xrightarrow{e_{r,f}} v^{(j-r(v))\%f}$ in $G_{r,f}$. From Property 3.1, we know that

(1) $d(e) = d_f(e_f) \cdot f + (j - i)$

(2) $d_r(e) = d(e) + r(u) - r(v)$

(3) $d_r(e) = d_{r,f}(e_{r,f}) \cdot f + ((i - r(u)) \,\%\, f) - ((j - r(v)) \,\%\, f)$.

By replacing $d(e)$ and $d_r(e)$ in (2) by (1) and (3), we have

$$d_{r,f}(e_{r,f}) - d_f(e_f) = \frac{1}{f}(r(u) - i - ((i - r(u)) \,\%\, f)) - \frac{1}{f}(r(v) - j - ((j - r(v)) \,\%\, f)).$$

Since the values for the intermediate nodes are canceled out, for any path $u^i \overset{\mathsf{p}}{\leadsto} v^j$ in H we have

$$w(\mathsf{p}) = \frac{1}{f}(r(u) - i - ((i - r(u)) \,\%\, f)) - \frac{1}{f}(r(v) - j - ((j - r(v)) \,\%\, f)).$$

Thus, the linear system is consistent and has an integer solution.

Since G_{f,r_f} is equivalent to $G_{r,f}$, the retiming r_f is certainly a legal retiming. The lemma is proved. $\Box$

The following theorem is derived from Lemma 3.5.1 and Lemma 3.5.3.

Theorem 3.1 *Let G be a DFG and f a fixed unfolding factor.*

(a) *There is a legal retiming r on the original graph G such that $\mathcal{C}(G_{r,f}) \leq c$ if and only if there is a legal retiming r_f on the unfolded graph G_f such that $\mathcal{C}(G_{f,r_f}) \leq c$.*

(b) $MCP(f) = \min_r \mathcal{C}(G_{r,f}) = \min_{r_f} \mathcal{C}(G_{f,r_f})$.

It would seem that retiming on the unfolded graph would tend to have a better iteration period because finer retiming functions can be found. However, this theorem tells us that, for a fixed unfolding factor, the minimum iteration period can be found no matter which transformation, unfolding or retiming, is performed first. Obviously, a retiming on the original graph saves time and space.

Chapter 4

Unit-Time Data-Flow Graphs

4.1 Introduction

In order to explore different implementations of an iterative algorithm, efficient transformations preserving the same behavior are important. *Retiming* and *unfolding* are two common and useful transformations that preserve the behavior of a data flow graph [12, 77]. They are time- and space- consuming operations, especially for large data flow graphs (DFG) or bit-level DFG's [23]. In this chapter, we present efficient algorithms to combine retiming and unfolding for *unit-time DFGs*, where all operations have the same computation time. The next chapter will talk about general-time-DFG scheduling by using the concepts used for unit-time DFGs here.

A schedule is *rate-optimal* if its iteration period equals the iteration bound, as defined in Subsection 2.2.2. We say that an unfolding factor f is *rate-optimal* if the DFG unfolded f times can achieve the iteration bound with retiming.

Since the unfolded graph is f times larger, we would like to find the *minimum* rate-optimal unfolding factor to achieve rate-optimality. Assuming that enough hardware is available, Parhi and Messerschmitt [63, 64] show that the rate-optimal unfolding factor is upper bounded by the least common multiple of the delay counts of all cycles in DFG. Jeng and Chen [38] claimed that for unit-time DFGs an unfolding factor $D(l_{cr})$ can be used to obtain a rate-optimal schedule without showing how to retime the DFG, where $D(l_{cr})$ is the number of delays of a critical cycle, as defined in

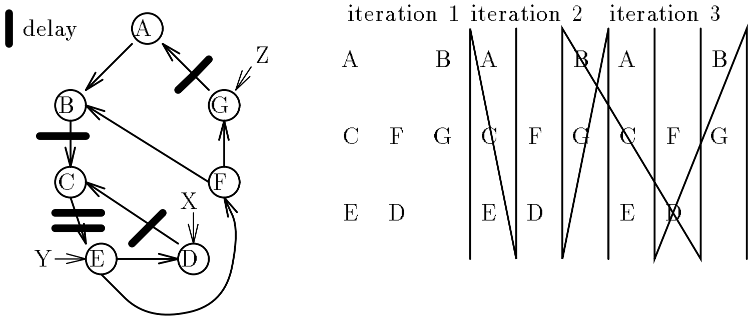
Subsection 2.2.2. We will prove their claim, and show that the minimum rate-optimal unfolding factor may be even smaller for some DFGs.

Since the necessary memory and controller for a schedule is proportional to the size of unfolding factor [38], reducing the unfolding factor is an important problem. It is possible that the unfolding factor for obtaining the rate-optimal schedule is too large to satisfy the memory and controller requirements of a design. Thus, it is important for designers to have a simple criterion such that for any given unfolding factor f, the minimum iteration period is obtained.

Since from Chapter 3, the order of retiming and unfolding is not important, we will use the approach of retiming first, then unfolding. In this chapter, we will show the following results. It is always true that there exists a retiming such that a rate-optimal schedule with unfolding factor $D(l_{cr})$ can be obtained. There is an efficient algorithm for finding such a retiming to achieve a rate-optimal schedule in time $O(|V||E|)$ without dealing with large unfolded graphs. Moreover, the unfolding factor $D(l_{cr})$ can be further reduced, and we prove that the minimum unfolding factor for rate-optimal scheduling is $D(l_{cr})/\gcd(T(l_{cr}), D(l_{cr}))$.

We have a simple criterion for any given unfolding factor f: there is a retiming such that a retimed and unfolded graph can achieve cycle period c as long as $\frac{c}{f} \geq \mathcal{B}(G)$. For any given unfolding factor f, we can always find a retiming with the minimum feasible iteration period $\frac{\lceil f \cdot B(G) \rceil}{f}$, and $\frac{\lceil f \cdot B(G) \rceil}{f} - B(G) \leq \frac{1}{f}$. This retiming algorithm is the same as the previous algorithm.

Software pipelining has been used for compilation for parallel machines [45]. In section 4.4, we extend the results for unit-time DFGs to an efficient solution for the problem of software pipelining for loops without conditional jumps [2, 28, 81]. The previously best-known algorithm is designed by Iwano and Yeh [36], which runs in time $O(|V|^3|E|)$ by a graph-based approach. They also found the minimum rate-optimal unfolding factor and the criteria for the existence of a static schedule. Our approach, based on retiming and unfolding, is completely independent from their approach. We will present a faster algorithm because unfolding is performed implicitly. We model software pipelining of general-time DFGs as unit-time DFGs. Thus, the



(a) a unit-time DFG G (b) a static schedule with iteration period = 3

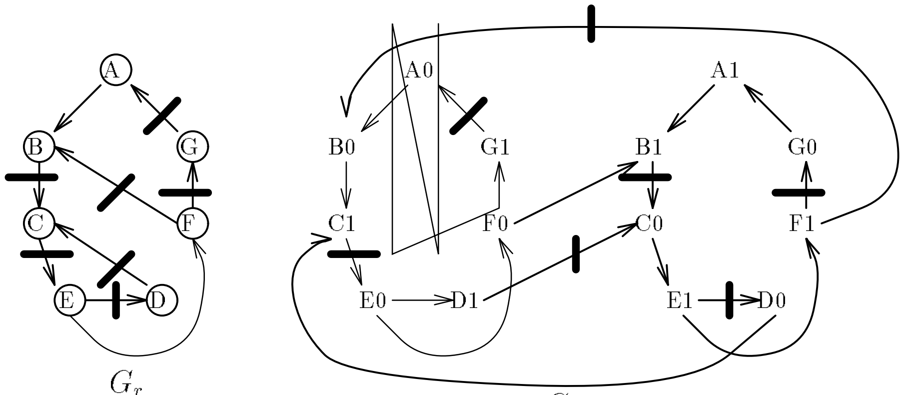
Figure 4.1: A data-flow graph and its static schedule

result about the minimum rate-optimal unfolding factor and the criteria for the existence of a static schedule that we derive for unit-time DFGs can be applied directly to software pipelining of general-time DFGs. (They are the same as those derived by [36]) After finding the bound $\mathcal{B}(G)$ in time $O(|V||E|\log|V|)$, we can find a retiming to achieve the rate-optimal schedule in time $O(|V||E|)$. A rate-optimal schedule can be generated from the retiming in time $O(f|E|)$.

4.2 Retiming Algorithm with Implicit Unfolding

We use a simple example to explain the usages of our algorithms and theorems. Given the following program, we can build a DFG as in Figure 4.1, where `X`, `Y` and `Z` are input arrays, and `A` to `G` are arrays to be computed.

```
for i = 1 to n do
  A[i] = G[i-1] + 10;
  C[i] = B[i-1] + D[i-1];
  E[i] = C[i-2] + Y[i];
  D[i] = E[i] + X[i];
  F[i] = E[i] + 5;
  B[i] = A[i] + F[i];
  G[i] = F[i] + Z[i];
end
```



G_r
$r(E) = r(F) = 1$

$G_{r,f}$

Figure 4.2: The retimed DFG G_r and the unfolded graph with $f = 2$

If we don't perform retiming and unfolding, the iteration period of the corresponding static schedule for G is 3. However, in time $O(|V|\,|E|)$ we know that the optimal iteration period is only 3/2, which is obtained from the critical cycle composed of nodes `A, B, C, E, F`, and `G`. In this chapter, we will show that the minimum rate-optimal unfolding factor should be 2.

Then by using our retiming algorithm in time $O(|V||E|)$, we can get a legal retiming to achieve this optimal iteration period. The DFG G_r in Figure 4.2-(a) shows the rate-optimal arrangement of delays, and the corresponding unfolded graph $G_{r,f}$ is in (b). A rate-optimal schedule is shown in Figure 4.3.

Let $G - x = (V,\ E,\ d',\ t)$ be the graph modified from $G = (V,\ E,\ d,\ t)$ where $d'(e) = d(e) - x$ for every edge e in E. Note that $G - x$ may have non-integral, even negative delay counts if x is not an integer.

Leiserson and Saxe gave a characterization for the cycle period after retiming for a unit-time DFG [53]. An algorithm was also proposed to compute a retiming r such that $\mathcal{C}(G_r) \leq c$.

Lemma 4.2.1 *[53] Let $G = (V,\ E)$ be a unit-time DFG, and c an positive integer. There is a legal retiming r on G such that $\mathcal{C}(G_r) \leq c$ if and only if $G - 1/c$ contains no cycle having negative delay counts.*

The iteration bound $\mathcal{B}(G)$ is introduced in Chapter 2 as a lower bound of the

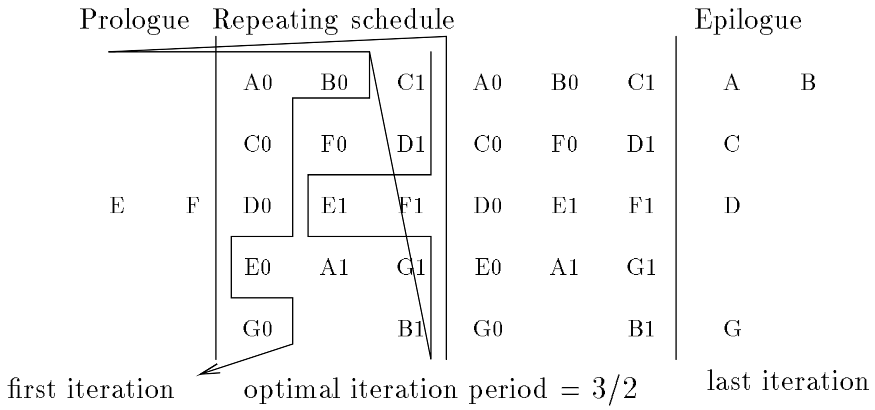


Figure 4.3: An optimal static schedule

iteration period of a data flow graph. Even though the retiming and unfolding techniques are combined, the iteration bound is still a lower bound of the iteration period. We first show the relation of $G - f/c$ to the lower bound $\mathcal{B}(G)$, and then show how the graph $G - f/c$ is used to characterize the cycle period of unit-time DFGs under retiming and unfolding.

Lemma 4.2.2 *Let G be a unit-time DFG and f and c positive integers. The graph $G - f/c$ contains no cycles having negative delay counts if and only if $c/f \geq \mathcal{B}(G)$.*

Proof: Assume that $G - f/c$ contains no cycles having negative delay counts. Let $D'(l)$ be the delay count of a cycle l in $G - f/c$. Since the number of edges in cycle l equals $T(l)$ in a unit-time DFG, we know $D'(l) = D(l) - (f/c) \cdot T(l) \geq 0$. Therefore, we have $c/f \geq T(l)/D(l)$ for every cycle l in G. The if part can be proved similarly. $\square$

Similar to the characterization of cycle period for unit-time DFGs under retiming in Lemma 4.2.1, we give a characterization of cycle period under both retiming and unfolding. A retiming algorithm with implicit unfolding is obtained.

Lemma 4.2.3 *Let $G = (V, E)$ be a unit-time DFG, c a positive integer, and f an unfolding factor. There is a legal retiming r on G_f such that $\mathcal{C}(G_{r,f}) \leq c$ if and only if $G - f/c$ contains no cycles having negative delay counts.*

Proof: The only if part can be easily proved by contradiction. If there is a cycle with negative delay, from Lemma 4.2.2 we know c/f is smaller than the lower bound

$\mathcal{B}(G)$. Thus, no retiming r exists such that $\mathcal{C}(G_{r,f}) \leq c$.

For the if part, we assume that there is no cycle with negative delay count. We construct a retiming for the DFG as follows. We first add a new node v_0 in the graph $G - f/c$, where v_0 is connected to all the nodes with zero length, and compute all the shortest paths from v_0 to other nodes, by using $d'(p)$ as the length of a path p. Let $Sh(v)$ be the length of the shortest path from v_0 to v. We choose the retiming r of node v to be the ceiling of $Sh(v)$ for every v in V.

It is easy to prove that r is a legal retiming and $\mathcal{C}(G_{r,f}) \leq c$, by a similar proof to that of Theorem 11 in [53]. The retiming r is legal if $d_r(e) = d(e) + r(u) - r(v) \geq 0$ for every edge $u \xrightarrow{e} v$. Since $Sh(v)$ is the length of a shortest path from v_0 to v for every node v, we have

$$Sh(v) \leq Sh(u) + d'(e) = Sh(u) + d(e) - f/c.$$

Taking ceilings of both sides gives

$$\begin{aligned} r(v) &\leq d(e) + \lceil Sh(u) - f/c \rceil \\ &\leq d(e) + \lceil Sh(u) \rceil \\ &\leq d(e) + r(u). \end{aligned}$$

Thus, $d_r(e)$ is no less than zero for every edge e.

Next, we show that the cycle period $(G_{r,f}) \leq c$. From Lemma 3.3.2, we need to show that $d_r(p) \geq f$ for any path $u \overset{p}{\leadsto} v$ containing c or more edges. From the shortest path function, we have $Sh(v) \leq Sh(u) + d'(p)$. Since p contains c or more edges, we know $d'(p) \leq d(p) - c \cdot (f/c) = d(p) - f$. Thus, we derive

$$\begin{aligned} Sh(v) &\leq Sh(u) + d(p) - f \\ r(v) &\leq d(p) - f + \lceil Sh(u) \rceil \\ &\leq d(p) - f + r(u). \end{aligned}$$

Therefore, we have $d_r(p) = d(p) + r(u) - r(v) \geq f$, and the cycle period $(G_{r,f})$ is no more than c. $\square$

From this lemma, we have a retiming algorithm to find a retiming r such that $\mathcal{C}(G_{r,f}) \leq c$. The algorithm is shown in Figure 4.4, where the graph $G - f/c$ is not constructed explicitly. We adopt the single-source shortest path algorithm introduced in Tarjan's book [75] to find the shortest paths $Sh(v)$ for all nodes v. If there is a negative cycle, the algorithm will report it, and if not, we return the retiming function $r(v)$ to be the ceiling of $Sh(v)$. The time complexity of this retiming algorithm is $O(|V| \cdot |E|)$. The static schedule with cycle period c can be easily obtained by unfolding the retimed graph G_r in time $O(f|E|)$.

Note that the retiming function found by the above algorithm is non-positive since there is at least one path of length zero from v_0 to every other node. Also, we know that $\max_v r(v)$ is zero. If a non-negative retiming is required, one can normalize the retiming function by adding the amount $-\min_v r(v)$. We can also design another algorithm to find a non-negative retiming function by adding a zero-length edge from every node to v_0, and computing the longest path from every node to v_0.

The following theorem provides a simple existence criterion, $c/f \geq \mathcal{B}(G)$, to check if under a given unfolding factor f, a given cycle period c can be achieved.

Theorem 4.1 *Let G be a unit-time DFG and f and c positive integers. The following statements are equivalent.*

1. *$c/f \geq \mathcal{B}(G)$.*
2. *$G - f/c$ does not contain any cycle having negative delay count.*
3. *there exists a legal retiming r on G such that, after G is retimed by r and unfolded by f, the cycle period $\mathcal{C}(G_{r,f}) \leq c$.*
4. *there exists a legal retiming r_f on G_f such that, after G_f is retimed by r_f, the cycle period $\mathcal{C}(G_{f,r_f}) \leq c$.*

Proof: From lemma 4.2.2, we know the first and the second statements are the same. The equivalence of the second and the third ones is proved from Lemma 4.2.3. The equivalence of the third and the last statements is proved in Theorem 3.1. Thus, all the four statements are equivalent. □

For a pair of given integers f and c, if there exists a legal retiming r such that $\mathcal{C}(G_{r,f}) \leq c$, we say the pair is feasible. From Theorem 4.1, the designer only needs

Retiming Algorithm
Input : a DFG G, an unfolding factor f and a cycle period c.
Output : A retiming r such that $\mathcal{C}(G_{r,f}) \leq c$.
begin /* $Sh(v)$ represents the length of the shortest path
 from v_0 to v. Q is a first-in first-out queue. *pass* is used
 to prevent the algorithm being trapped in negative cycles. */
  for every node $v \in V$ **do begin**
    $Sh(v) \longleftarrow 0$; PUSH v to Q;
  end
  $pass \longleftarrow 0$;
  $last \longleftarrow$ the last element in Q;
  while Q is not empty **and** $pass < |V|$ **do begin**
    Pop v from Q;
    for every edge $e = (v, w) \in E$ **do begin**
      $Sh(w) \longleftarrow \min\{Sh(v) + d(e) - f/c, Sh(w)\}$
      if w is not in Q **then** PUSH w to Q;
    end
    if $v = last$ **then begin**
      $pass \longleftarrow pass + 1$;
      $last \longleftarrow$ the last element in Q;
    end
  end
  if Q is empty **then**
    for every $v \in V$ **do** $r(v) \longleftarrow \lceil Sh(v) \rceil$
  else "There is a negative cycle in the graph $G - f/c$."
end

Figure 4.4: Retiming algorithm for unit-time DFGs

to check the inequality $c/f \geq \mathcal{B}(G)$ in order to decide the feasibility of a pair of f and c, instead of checking whether there is any negative cycle in $G - f/c$. Thus, a lot of running time is spared.

From the above theorem, we know that every unfolding factor f which is a multiple of $D(l_{cr})$ is a rate-optimal unfolding factor. Actually we can find even a smaller f as a rate-optimal unfolding factor. The following theorem shows that the minimum rate-optimal unfolding factor is the denominator in the irreducible form of $\mathcal{B}(G)$.

Theorem 4.2 *The minimum rate-optimal unfolding factor is* $D(l_{cr})/\gcd(T(l_{cr}), D(l_{cr}))$, *where* l_{cr} *is the critical cycle and* gcd *is the greatest common divisor.*

Proof: Let $\hat{f}$ be $D(l_{cr})/\gcd(T(l_{cr}), D(l_{cr}))$ and let $\hat{c}$ be $T(l_{cr})/\gcd(T(l_{cr}), D(l_{cr}))$. From Theorem 4.1, we know that the pair $(\hat{f}, \hat{c})$ is feasible and the value $\hat{f}$ is a rate-optimal unfolding factor, which means $\hat{c}/\hat{f} = \mathcal{B}(G)$. Assume that f is another rate-optimal unfolding factor. Let c be the cycle period such that the pair (f, c) is feasible and $c/f = \mathcal{B}(G)$. Then, c/f and $\hat{c}/\hat{f}$ both are a reduced form of $T(l_{cr})/D(l_{cr})$. Because gcd is the greatest common divisor of $T(l_{cr})$ and $D(l_{cr})$, $\hat{c}/\hat{f}$ must be the most reduced form. Thus, we have $f \geq \hat{f}$, and the value $\hat{f}$ must be the minimum rate-optimal unfolding factor. □

For example, if $T(l_{cr}) = 60$ and $D(l_{cr}) = 45$, our result shows that a rate-optimal unfolding factor can be 3 instead of 45, and that this value 3 is the minimum value.

Assume that we would like to find a rate-optimal schedule. We can first find the iteration bound $\mathcal{B}(G)$ in its irreducible form, denoted $\hat{c}/\hat{f}$. By a dynamic programming technique the iteration bound for unit-time DFG can be found in $O(|V||E|)$ [41]. The value $\hat{f}$ is the minimum rate-optimal unfolding factor. Then, we can use the retiming algorithm to find a retiming r such that $\mathcal{C}(G_{r,\hat{f}}) \leq \hat{c}$ in time $O(|V||E|)$. The rate-optimal schedule can be derived from $G_{r,\hat{f}}$.

In some cases, this minimum rate-optimal unfolding factor will still be too large for the design requirements, for example, when $D(l_{cr})$ is a large prime number. The designer may want to trade off between an unfolding factor and the corresponding minimum iteration period under all the possible retimings. For an unfolding factor

f, the minimum feasible c from the inequality $c/f \geq \mathcal{B}(G)$ is

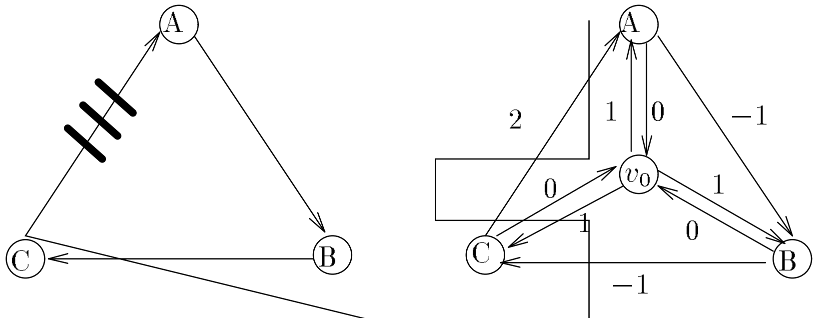
$$MCP(f) = \lceil f \cdot \mathcal{B}(G) \rceil.$$

Thus, all the pairs of f and $MCP(f)$ are easily generated, and the designer may choose the suitable pair according to the design requirements. A legal retiming for such a chosen pair can be found merely in time $O(|V||E|)$. The maximum difference between the iteration period of the chosen pair and the iteration bound $\mathcal{B}(G)$ is less than $1/f$.

4.3 Length of Prologue and Epilogue

The disadvantage of using the technique of retiming is the introduction of the prologue and the epilogue sections. This penalty can be neglected if the number of iterations is large, e.g. in digital signal processing applications. However, when the number of iterations is small, we would like to restrict the length of the prologue and epilogue. This section provides techniques to restrict the length of the prologue and epilogue in the retiming algorithm.

From Section 2.5, we know the number of iterations in the prologue and epilogue is $\mathcal{P}(r)$, which is $\max_v r(v)$ for a normalized retiming r. We can control the length of the prologue and epilogue by adding constraints for $\mathcal{P}(r)$ during the optimization process. For a given prologue length p, the addition of the constraints $r(u) \geq 0$ and $r(u) \leq \mathsf{p}$ for every node u in V will find a normalized retiming such that the length of the prologue is $\leq \mathsf{p}$. The retiming algorithm for unit-time DFGs in Section 4.2 is modified as follows. We use the added new node v_0 as a reference point: $r(v_0) - r(v) \leq 0$ and $r(v) - r(v_0) \leq \mathsf{p}$. In the graph $G - f/c$ plus node v_0, we replace the zero-length edge e from v_0 to every node u in V with length $d'(e) = \mathsf{p}$, and add an edge e from every node u to v_0 with length $d'(e) = 0$. If there is a negative cycle in this modified $G - f/c$, no retiming can satisfy the prologue requirements and the cycle period c simultaneously. Otherwise, we set the retiming function to be $r(u) = \lceil Sh(u) \rceil$ for every u in V, where $Sh(u)$ is the length of the shortest path from v_0 to u for every u in $V \cup \{v_0\}$. Thus, the retiming function obtained as above



(a) a DFG (b) the constructed graph

Figure 4.5: An example to control the length of the prologue

is non-negative and satisfies $\mathcal{C}(G_{r,f}) \leq c$ and $\mathcal{P}(r) \leq \mathsf{p}$. The retiming algorithm for general-time DFGs in Section 6.1 can be similarly modified.

Consider the DFG in Figure 4.5-(a). The constructed graph for the DFG is shown in Part (b) with cycle period $c = 1$, unfolding factor $f = 1$ and prologue length $\mathsf{p} = 1$. There is a negative cycle: A-B-C-v_0. Thus, there does not exist a schedule with cycle period 1 and prologue length 1.

4.4 Software Pipelining

One of the important applications of loop scheduling is the compilation for architectures with multi-functional units, e.g. VLIW machines or data-path designs of a VLSI architecture. Since each instruction needs a different computation time, a loop in a program has to be modeled as a general-time DFG. In this section, we look at the so-called *software pipelining* problem in compiler design [45] or *functional pipelining* problem in data-path synthesis [30] for loops without conditionals.

An instruction is *issued* in the first time unit of its period of execution. Consider a precedence constraint from node B to node C, where the computation time of B is 4. Node C should be issued 4 time units after node B is issued. In software pipelining, a static schedule is composed of instruction issues, each of which takes one time unit. Therefore, the schedules using pipelined processors are considered as described in Section 2.7. It is likely that an instruction which is issued in a cycle may not be completed within the same period and its execution may actually span several cycle periods. In order to model this additional flexibility, we allow a node to be divided into parts with integral computation times, each of which is executed within a different cycle period of the static schedule. Thus, the divided node is itself pipelined. As in ordinary DFGs, every delay arrangement (retiming) corresponds to a static schedule.

Such scheduling problems without resource constraints can be solved by an efficient retiming algorithm as unit-time DFGs in time $O(|V|\,|E|)$, although the problem instance is actually a general-time DFG. In Chapter 5, another algorithm of the same complexity is presented to generate a schedule directly without explicit retiming.

In this section, we model a loop without conditional jumps as a general-time DFG G. As introduced in Section 2.6, pipelined schedules corresponds to retiming on general-time DFGs with splitable nodes. We use the retiming algorithm for unit-time DFGs to solve the software pipelining problem by representing each general-time node as a sequence of unit-time nodes. The time complexity of this approach depends on the computation time of nodes. To improve the time complexity, we present an $O(|V|\,|E|)$-time retiming algorithm for a given unfolding factor f and cycle period c by working on the general-time DFG with nodes divisible into integral parts. The rate-optimal static schedule can be found in time $O(|V|\,|E| \log |V| + f|E|)$.

To model a general-time DFG as a unit-time DFG, we represent a computation node v with time $t(v)$ by a chain of $t(v)$ unit-time subnodes connected with zero-delay edges. All the incoming edges of v come into the first subnode, and all the outgoing edges of v go out from the last subnode in the chain. The first subnode, called the *issue node*, represents the issue of node v; the remaining subnodes are used only to preserve the separation constraints and do not take part in the static schedule of software pipelining. The general-time DFG G is thus transformed into a unit-time DFG $\hat{G}$. Note that the iteration bound is the same for both G and $\hat{G}$.

We can perform the retiming algorithm in Section 4.2 on $\hat{G}$ to find a static schedule with a given f and c. All the results in Section 4.2 can be applied. Thus, we know $MCP(f) = \lceil f \cdot \mathcal{B}(G) \rceil$, and the rate-optimal schedule can be found. However, the number of subnodes in $\hat{G}$ equals the total computation time of all nodes in G. The time complexity is proportional to the total computation time of G, which may not

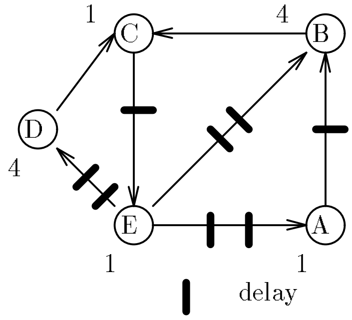


Figure 4.6: A DFG G

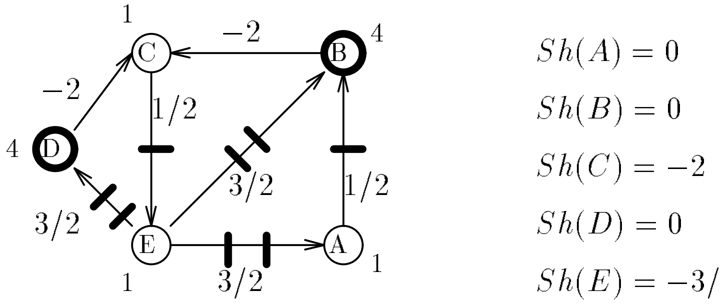


Figure 4.7: The graph G'' with $d''(e) = d(e) - t(u) \cdot f/c$ for every edge e

be polynomial in $|V|$ and $|E|$.

In this section, we present a retiming algorithm which simulates the retiming algorithm on the unit-time DFG $\hat{G}$, but only keeps information for the issue nodes, which compose the static schedule.

The retiming function of a unit-time DFG is computed from the shortest paths of all nodes from node v_0. And, the actual schedule is obtained from the DAG of the retimed graph. For a general-time DFG modeled as a unit-time one, the shortest paths of all subnodes of a general-time node v can be computed from the issue node, since there is only one entry and one exit subnode for node v. Therefore, we can compute the shortest paths for the issue nodes only, and still maintain enough information to generate the actual schedule.

We modify G into another graph $G'' = (V, E, d'', t)$ with the modified delay function $d''(e) = d(e) - t(u) \cdot f/c$ for an edge e from u to v. The retiming algorithm is then performed on G'' instead of $\hat{G} - f/c$. The shortest path from v_0 to node v in G'' equals the shortest path from v_0 to the issue node of v in the unit-time DFG $\hat{G}$, denoted by $Sh(v)$.

Consider the DFG in Figure 4.6 as an example, where nodes B and D have computation time 4. The iteration bound $\mathcal{B}(G)$ of this DFG is 2 with the critical cycles CED and CEB. Figure 4.7 shows the the corresponding modified graph, and the number beside an edge e shows the value of modified delay function $d''(e)$. From the results in Section 4.2 for unit-time DFGs, we know the minimum rate-optimal unfolding factor is 1, and we should be able to obtain a static schedule of length 2.

We first compute all the $Sh(v)$ for every v in V, as shown in Figure 4.7. The retiming function r for each issue node v is $r(v) = \lceil Sh(v) \rceil$. We can generate the static schedules from the values $Sh(v)$, as described below.

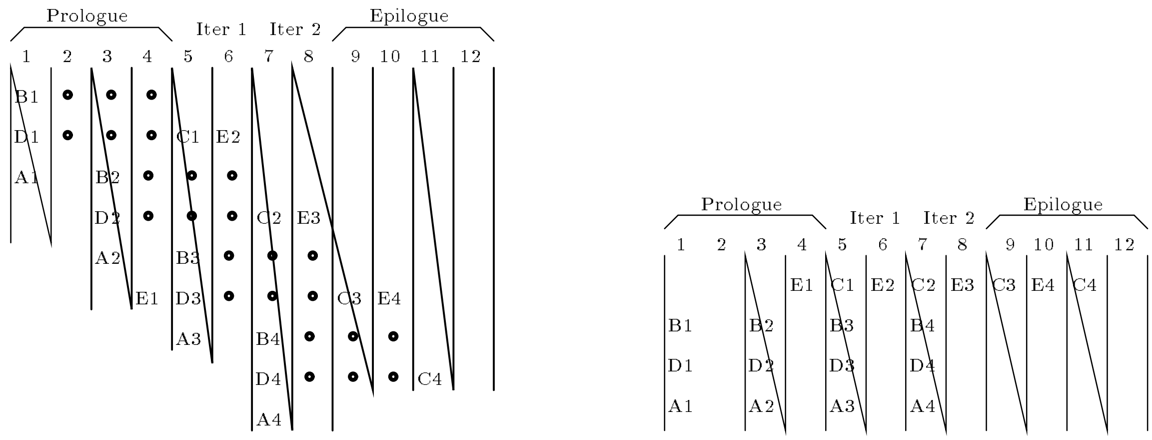
In DFG G_r, A node v is divided by delays if the number of delays pushed into node v (the sequence of subnodes of v) is more than the number of delays pushed out of node v. Without loss of generality, we assume that each node is divided into parts of lengths $head(v)$, $\lceil c/f \rceil$, $\ldots$, $\lfloor c/f \rfloor$, and $tail(v)$ in the DFG G_r where $0 \leq head(v) < c$ and $0 \leq tail(v) < c$. For example, a node B of computation time 4 can be divided into 3 parts of times 1, 2, and 1 by 2 delays, where $head(v) = tail(v) = 1$.

The static schedule of instruction issues is determined from the acyclic graph consisting of zero-delay edges. The static schedule of a DFG with divisible nodes can be determined from the functions $tail(v)$ and $head(v)$ for every v. The value $tail(v)$ is used to decide the earliest starting time of the nodes with incoming edges from node v; the value $head(v)$ is used to decide the latest finishing time of the nodes with outgoing edges to node v. Therefore, if $head(v)$ and $tail(v)$ are known, the static schedule can be obtained.

We can derive the functions *head* and *tail* from the values $Sh(v)$.

Lemma 4.4.1 *For every node v in V, we have*
(a) $head(v) = \left\lceil \dfrac{Sh(v) - \lfloor Sh(v) \rfloor}{f/c} \right\rceil$.
(b) $tail(v) = \left\lfloor \dfrac{\lceil Sh'(v) \rceil - Sh'(v)}{f/c} \right\rfloor$, *where* $Sh'(v) = Sh(v) - t(v) \cdot f/c$.

Proof: Assume that v has subnodes v_1, v_2, $\ldots$, $v_{t(v)}$ in the unit-time DFG $\hat{G}$. While we trace forward from the issue node through the subnodes, the length $Sh(v_i)$ is



(a) the time table of instructions (b) A static schedule of instruction issues

Figure 4.8: The schedule for software pipelining

decreased by an amount of f/c for each increment in i from $Sh(v_1) = Sh(v)$. The value $head(v)$ is the largest i such that $\lceil Sh(v_i) \rceil = \lceil Sh(v) \rceil$ for every subnode v_i. Once the total amount decreased is larger than $Sh(v) - \lfloor Sh(v) \rfloor$, we found the subnode with index $head(v) + 1$. Therefore, we have $head(v)$ equals $\left\lceil \frac{Sh(v) - \lfloor Sh(v) \rfloor}{f/c} \right\rceil$.

The values of $tail(v)$ can be similarly derived. We imagine that there is a pseudo node, denoted by v_{tl}, as the tail of node v, which has 0 computation time. Then, $Sh(v_{tl} = Sh(v) - t(v) \cdot f/c$. While we are tracing back from v_t through subnodes, the length of shortest paths $Sh(v_i)$ are increasing, each time by an amount of f/c. Once the total amount increased is larger than $\lceil Sh(v_{tl}) \rceil - Sh(v_{tl})$, we found the subnode with index $tail(v) - 1$. Therefore, we have $tail(v)$ equals $\left\lfloor \frac{\lceil Sh(v_{tl}) \rceil - Sh(v_{tl})}{f/c} \right\rfloor$. $\square$

Note that if $head(v) \geq c$ or $tail(v) \geq c$, the node is not divided by any delay. For example, when $Sh(B) = 0$, and $t(B) = 4$ $(f = 1, c = 2)$, we have $head(B) = 0$ and $tail(B) = 0$, which means that a delay is on each of the incoming edges and each of the outgoing edges of B. Since $t(B) = 4$, we know that node B is partitioned into 2 parts of length c, which is 2. Consider another example where $Sh(B) = -1/2$, and $t(B) = 4$ $(f = 1, c = 2)$. We have $head(B) = 1$ and $tail(B) = 1$, which means that node B is partitioned into 3 parts of lengths 1, 2, and 1.

Consider the example in Figure 4.6. After retiming, the minimum cycle period of the DFG is 4, which is far from the lower bound, 2. However, if we treat each node

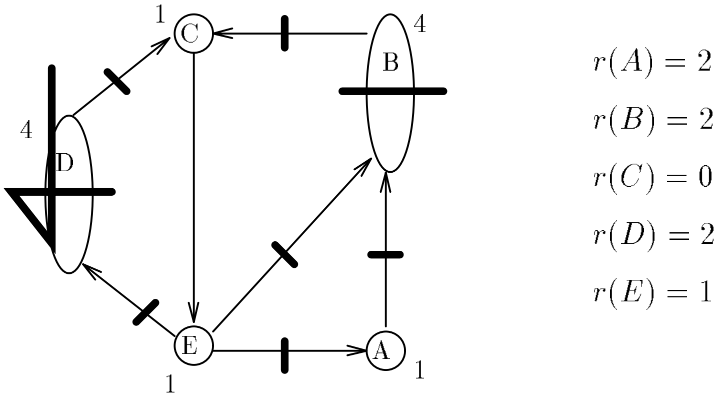


Figure 4.9: The retimed DFG with split nodes

as divisible nodes, and apply the above algorithm, we obtain a static schedule with cycle period 2, as shown in Figure 4.8. Part (a) shows the actual execution periods of nodes; Part (b) shows only the instruction issues. The normalized retiming function r and the corresponding retimed graph are shown in Figure 4.9. For the prologue, as mentioned in Section 2.5, there are $r(v)$ copies of node v issued for every node v. The static schedule can be obtained from the retiming by using the functions $head(v)$ and $tail(v)$. If the general-time DFG in the example is modeled as a unit-time DFG, the retiming algorithm on the unit-time DFG will obtain the same delay configuration as that shown in Figure 4.9. But, the time complexity of such a retiming algorithm depends on the magnitude of the computation time of each node.

Since we work on the DFG G'', the time complexity of the retiming algorithm for software pipelining is still $O(|V|\,|E|)$. It takes time $O(|V|\,|E| \log |V|)$ to compute the iteration bound $\mathcal{B}(G)$ for general-time DFGs [36, 41], so the retiming function achieving the iteration bound for a loop program can be found in time $O(|V|\,|E| \log |V|)$.

4.5 Conclusion

Although unit-time DFGs seem to be a very simple model, it turns out that unit-time DFGs are useful for solving the problem of software pipelining in compiler design for parallel machines. From the nice properties of unit-time DFGs, we are able to obtain a very efficient algorithm to find retimings and unfolding factors for rate-optimal schedules. Also, very simple formulas are derived for a given unfolding factor to predict the quality of schedules of the unfolding factor.

By considering general-time DFGs with splitable nodes as unit-time DFGs, we are able to generate schedules with *integral* starting times. The next chapter will discuss the generalization that a node can be split into *fractional* parts. Such model corresponds to schedules with *fractional* starting times.

Chapter 5

Various Models

5.1 Introduction

The last chapter is concerned with retiming and unfolding on unit-time DFGs. Pipelined schedules for general-time DFGs are produced by allowing node splitting. Based on the same concept, pipelined schedules with fractional starting times can be generated by allowing a general-time node to be split into fractional parts. Since such a fractional retiming is hard to interpret, this chapter develops an algorithm to generate integral or fractional schedules without generating retiming functions as an intermediate step.

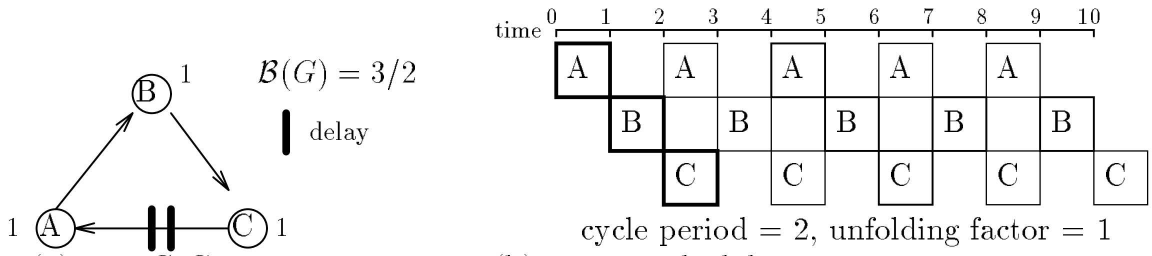
There has been much work on finding the rate-optimal static schedule with the minimum unfolding factor under different assumptions [38, 78, 65]. This chapter tries to clarify this problem by using two kinds of timing models and two types of implementations. We derive the minimum rate-optimal unfolding factor for each of the four combinations, and design an efficient unified polynomial-time algorithm. Moreover, we will prove general theorems for static scheduling where the rate-optimal scheduling is just a special case. From our results we know the relationship between an unfolding factor and its corresponding minimum iteration period for static scheduling and how to get such a schedule.

Two timing models are considered. If a scheduled computation can start at any rational number, we say this model is a *fractional model*; otherwise is an *integral-grid* *model.* Basically the fractional model is much simpler than the integral-grid model, because it does not have the integral constraints. Therefore we will first focus on the integral-grid model, and then we can easily apply our results to the fractional model. Two design styles are studied: *pipelined* and *non-pipelined* processors, as introduced in Section 2.7. The software-pipelining problem solved in Section 4.4 uses pipelined design and the integral model. We show how to find a rate-optimal static schedule with the minimum unfolding factor under pipelined implementation and non-pipelined implementation for the two timing models.

The next section introduces the two timing models and some properties of the two design styles. An efficient polynomial-time algorithm is presented in Section 5.3, which can be applied to all four combinations of models. This algorithm is particularly efficient because it works on the original DFG instead of the large unfolded DFG. In Section 5.4 we show our results on the integral models for pipelined and non-pipelined designs.

For the pipelined design and integral timing model, the results from Chapter 4 directly lead to the minimum rate-optimal unfolding factors. By using the techniques developed for the pipelined design, we are able to prove that the minimum rate-optimal unfolding factor for non-pipelined design is $\left\lceil \frac{\max_v t(v)}{\sigma} \right\rceil \cdot \rho$, where $\max_v t(v)$ is the maximum computation time over all nodes, and σ and ρ are the numerator and the denominator of the irreducible form of the iteration bound $B(G)$. From this result, we conclude that the minimum unfolding factor shown in [39] is incorrect. Section 5.5 describes the results on the fractional model. We show the minimum rate-optimal unfolding factor is just 1 for pipelined design and is ρ for non-pipelined design. In addition to providing minimum rate-optimal unfolding factors and algorithms, the contribution of this work is to give a general technique which can be generalized to other scheduling problems under different design situations, and can be easily combined with other structural transformation techniques.

Many results for unit-time DFGs also hold for scheduling general-time DFGs under the integral timing model and pipelined implementation. For clarity, we recast them in this chapter so that the reader can compare them with the results of other models.



(a) a DFG G (b) a static schedule

Figure 5.1: A simple DFG

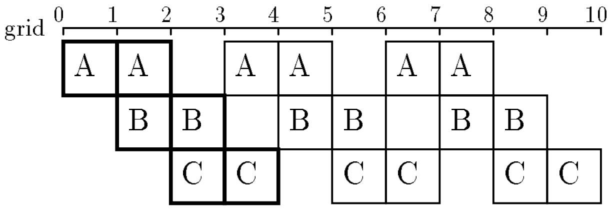
5.2 Various Models and Design Styles

Consider a data-flow graph $G = (V, E, d, t)$ as defined in Definition 2.1. A *schedule* is represented by a function $S : V \times N \longrightarrow R$. The starting time of node v in the i-th iteration is $S(v, i)$. A schedule is *legal* if for every edge $u \stackrel{e}{\longrightarrow} v$ and iteration i, we have $S(u, i) + t(u) \leq S(v, i + d(e))$. A schedule is said to have an *unfolding factor* f and a *cycle period* c if $S(v, i + f) = S(v, i) + c$ for every v in V and iteration i. Thus, such a schedule can be represented by the partial schedule of the first f iterations. A new instance of this partial schedule of f iterations can be initiated for every interval of length c to form a legal complete schedule. Moreover, if in every instantiation, an operation of the partial schedule is assigned to the same processor, this type of schedule is called a *static schedule* or a *processor-static schedule*.

If a schedule does not have a repeating pattern, there does not exist an unfolding factor and a cycle period for this schedule. We only consider schedules with a repeating pattern. Note that in this chapter we organize the repeating pattern of a schedule as the ordinary model, as depicted in Figure 2.7. There is no distinction of the prologue and epilogue.

Under different timing or architecture models, there are several ways of implementing a static schedule. We classify the implementation styles below, and present one unified algorithm to solve the scheduling problems under different models in Section 5.3. The minimum rate-optimal unfolding factors under different models are also characterized in Sections 5.4 and 5.5.

The measurement of computation time and cycle period is in terms of a pre-defined *time unit*, which may be a machine cycle or a clock cycle under different models.



cycle period = 3, unfolding factor = 2

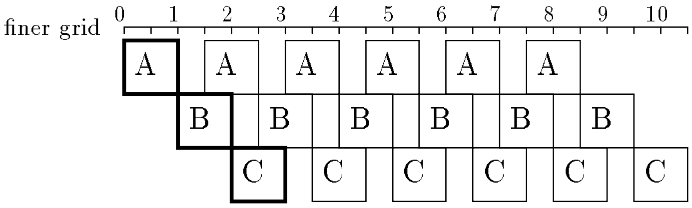
Figure 5.2: Integral-grid model

Without loss of generality, we assume that every node has an integral computation time. There are two models for the timing style of scheduling.

Integral-grid model We imagine there is an integral grid (time unit) for the schedule. An operation can only be issued in the beginning of a time unit. Under this model, a schedule has an integral cycle period.

Fractional Model (Gridless Model) The starting time of a node can be any fractional number, i.e. the schedule can have an infinitely fine grid, and an operation can be issued at any time. Under this model, a schedule may have a fractional cycle period. Actually, we will show that a grid of size $1/\rho$ gives schedules as good as gridless schedules, where ρ is the denominator of $\mathcal{B}(G)$ in its irreducible form.

Consider the DFG in Figure 5.1 with iteration bound $\mathcal{B}(G) = 3/2$. Figures 5.2 and 5.3 show a schedule under the integral-grid model and a schedule under the fractional model, respectively, where the first instances of the static schedules are highlighted by boxes with thicker boundaries. These two static schedules have the same iteration period, 3/2. In order to achieve rate-optimality, the static schedule under the integral-grid model consists of 2 iterations, while that under the fractional model contains one iteration. Usually, a static schedule is stored as a program code where each instruction consists a set of nodes to be issued. An instruction is executed at each clock tick. The costs of storing a static schedule can be measured by the number of instructions in the program code. Under the fractional model as shown, the clock ticks at every 1/2 time unit. Thus, the cost of storing the integral-grid and the fractional models are the same.



cycle period = 1.5, unfolding factor = 1

Figure 5.3: Fractional Grid Model

From Chapter 2, we know that a pipelined schedule of a general-time DFG corresponds to a retiming on the general-time DFG where nodes can be split into integral parts. Under the fractional timing model, a pipelined schedule of a general-time DFG also corresponds to a retiming on the general-time DFG where nodes can be split into fractional parts. For example, in Figure 5.3 the repeating part of the schedule, marked by the box with dashed boundary, corresponds to a retiming on the DFG in Figure 5.1-(a) such that node B is split into two half-unit parts.

Under every timing model, there are the two design styles, pipelined and non-pipelined design, as described in Section 2.7. For a static schedule with the unfolding factor f under the non-pipelined design, the $(i+f)$-th copy of a node cannot start execution before the i-th copy has finished execution. Thus, a schedule with unfolding factor f can be implemented by using non-pipelined processors if for every v in V and positive integer i, we have $S(v, i+f) \geq S(v, i) + t(v)$. While, for the pipelined design there is no such restriction.

Next, we derive the condition for a static schedule to be implemented under non-pipelined design. All the arguments are true for both the integral and fractional timing models. A new copy of a node assigned to the same processor starts execution every c time units in a static schedule of cycle period c. Thus, the cycle period must be larger than the computation time of a node. In order to improve the execution rate, one can execute several copies of a node with long computation time simultaneously. Hence, the schedule will have a larger cycle period, but the iteration period can be reduced. The following shows that as long as the cycle period of a schedule is is no less than $\max_v t(v)$, it can be implemented by a static schedule under the non-pipelined

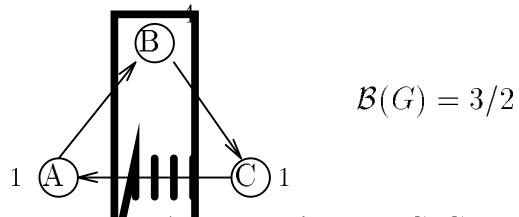


Figure 5.4: An exemplary DFG G

design.

Lemma 5.2.1 *Given a schedule S with cycle period $c \geq \max_v t(v)$, this schedule can be realized as a static schedule under the non-pipelined design.*

Proof: Assume that S has unfolding factor f. We can simply assign every copy of every node to a distinct processor. Since $t(v) \leq c$ for every v, we have $S(v, i+f) = S(v,i) + c \geq S(v,i) + t(v)$. Therefore, S is a static schedule under the non-pipelined design. $\Box$

Actually, the inequality $c \geq \max_v t(v)$ is also a necessary condition for a static schedule to be implemented under non-pipelined design.

Lemma 5.2.2 *Any static schedule S under non-pipelined design has a cycle period c where $c \geq \max_v t(v)$.*

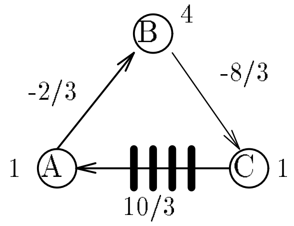
Proof: Assume that S has unfolding factor f. For every node v and iteration i, we have $S(v, i+f) \geq S(v,i) + t(v)$. Since S is repeated for every period of c, we know $S(v, i+f) = S(v,i) + c$. Thus, c must be no less than the computation time of a node. Therefore, we have $c \geq \max_v t(v)$. $\Box$

The above lemmas give the necessary and sufficient condition for a static schedule to be implemented under the non-pipelined design.

Theorem 5.1 *Let G be a DFG, and S be a legal schedule with cycle period c and unfolding factor f. The static schedule S can be implemented under the non-pipelined design if and only if $c \geq \max_v t(v)$.*

Proof: This theorem follows from Lemma 5.2.1 and Lemma 5.2.2. $\Box$

Thus, there are four combinations of implementations of static schedules. The following example, which is used throughout the rest of this chapter, shows that the



$$sh(A) = 0$$
$$sh(B) = -2/3$$
$$sh(C) = -10/3$$

Figure 5.5: The scheduling graph G^s with $c/f = 3/2$

minimum rate-optimal unfolding factor obtained may be different under these four combinations. The exemplary DFG is shown in Figure 5.4, and its iteration bound $\mathcal{B}(G)$ is 3/2. The minimum rate-optimal unfolding factors derived from theorems in this chapter for these four combinations are as follows. The derivations of unfolding factors and the corresponding schedules will be explained in this chapter.

Min Rate-optimal Unfolding Factor	Timing Models	
	Fractional	Integral
Pipelined Design	1	2
Nonpipelined Design	3	4

5.3 Unified Scheduling Algorithm

We present an algorithm to find a schedule for given cycle period c and unfolding factor f under the fractional and integral models. Throughout this section, no constraint is put on the cycle period c and the maximum computation time $\max_v t(v)$. Therefore, the pipelined design is considered. Nevertheless, the same algorithm can be used for the non-pipelined design as long as Theorem 5.1 is satisfied.

For a DFG $G = (V, E, d, t)$ and given c and f, we define a modified graph with different weights on edges: scheduling graph $G^s = (V, E, w)$ where $w(e) = d(e) - t(u) \cdot f/c$ for every edge $u \xrightarrow{e} v$ in E. This modified graph was used in Section 4.4 to compute legal retimings when modeling general-time DFGs as unit-time ones. Here, we use the modified graph to compute legal schedules for different models.

The next lemma relates the lower bound $\mathcal{B}(G)$ to the existence of negative-weight cycles in G^s.

Lemma 5.3.1 *Let G be a general-time DFG, c a (fractional) cycle period, and f an unfolding factor. The scheduling graph G^s contains no cycles having negative weights if and only if $c/f \geq \mathcal{B}(G)$.*

Proof: Although the cycle period c can be fractional and G is a general-time DFG, the proof of this lemma is analogous to that of Lemma 4.2.2 for a unit-time DFG. $\square$

Assume that there is no negative-weight cycle in the scheduling graph G^s. We add a node v_0 and directed edges from v_0 to every other node with zero weight in G^s. Let $sh(v)$ be the length of the shortest path from v_0 to v in the scheduling graph G^s. The values of $sh(v)$ can be computed by any single-source shortest path algorithm. It is easy to observe that for every node v we have $sh(v) \leq 0$, and there exists a node u in V such that $sh(u) = 0$.

The following theorem obtains legal fractional and integral schedules from the scheduling graph.

Theorem 5.2 *Let G be a general-time DFG, c a cycle period, f an unfolding factor. Assuming that there is no negative-weight cycle in the corresponding scheduling graph G^s, let $sh(v)$ be the length of the shortest path from v_0 to v.*

(a) *$S^f(v,i) = -sh(v) \cdot c/f + i \cdot c/f$ for every v and i is a legal schedule under the fractional model.*

(b) *$S^i(v,i) = \lceil -sh(v) \cdot c/f + i \cdot c/f \rceil$ for every v and i is a legal schedule under the integral grid model.*

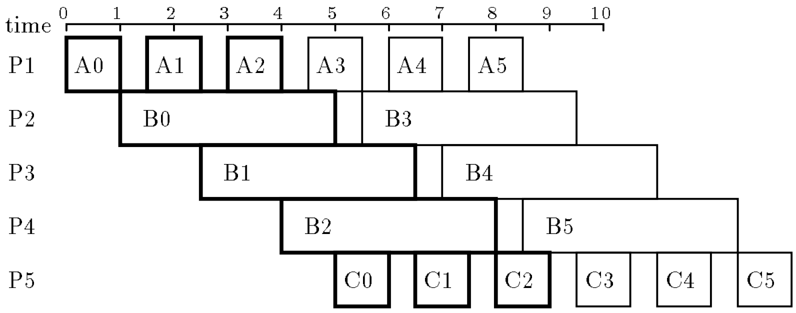
Proof: Consider any edge $u \xrightarrow{e} v$ and iteration i. We claim that $S^f(u,i) + t(u) \leq S^f(v, i+d(e))$. For every edge $u \xrightarrow{e} v$ in G^s, we have $sh(u) + d(e) - t(u) \cdot f/c \geq sh(v)$. The following is derived.

$$\begin{aligned} -sh(u) \cdot c/f + t(u) &\leq -sh(v) \cdot c/f + d(e) \cdot c/f \\ -sh(u) \cdot c/f + i \cdot c/f + t(u) &\leq -sh(v) \cdot c/f + i \cdot c/f + d(e) \cdot c/f \end{aligned}$$

Thus, we know $S^f(u,i) + t(u) \leq S^f(v, i + d(e))$, and S^f is a legal schedule.

Since $t(u)$ is integral, we have

$$\lceil -sh(u) \cdot c/f + i \cdot c/f \rceil + t(u) \leq \lceil -sh(v) \cdot c/f + i \cdot c/f + d(e) \cdot c/f \rceil .$$



cycle period = 4.5, unfolding factor = 3

Figure 5.6: Fractional model and nonpipelined implementation

Therefore, we know $S^i(u,i) + t(u) \leq S^i(v, i + d(e))$, and S^i is a legal schedule. $\Box$

Note that another integral schedule can be obtained by substituting the *floor* operation for the *ceiling* operation. Since there exists a node with $sh(v) = 0$, the schedule S^f and S^i both start from time 0. Consider the DFG in Figure 5.4 as an example. The iteration bound $\mathcal{B}(G)$ of this DFG is 3/2. The scheduling graph G^s is depicted in Figure 5.5 [1], where the number beside an edge e is weight $w(e)$. We first compute all the $sh(v)$ for every v in V, as shown in Figure 5.5. For $c = 4.5$ and $f = 3$, we obtain the fractional schedule in Figure 5.6, where $S^f(A,0) = 0$, $S^f(B,0) = 1$, and $S^f(C,0) = 5$. And, $S^f(A,1) = 0 + c/f = 1.5$, $S^f(B,1) = 2.5$, $S^f(C,1) = 6.5$.

The following lemma shows that the schedules S^f and S^i have unfolding factor f and cycle period c.

Lemma 5.3.2 *The schedules S^f and S^i have the following properties: For every node v and every positive integer i,*

(a) $S^f(v, i + f) = S^f(v, i) + c.$

(b) $S^i(v, i + f) = S^i(v, i) + c.$

Proof: For the fractional schedule S^f, we have

$$\begin{aligned} S^f(v, i+f) &= -sh(v) \cdot c/f + (i + f) \cdot c/f \\ &= -sh(v) \cdot c/f + i \cdot c/f + c \\ &= S^f(v,i) + c. \end{aligned}$$

[1]Notice that the scheduling graph G^s is the same for finding schedules of the same rate, i.e. the same c/f ratio. All rate-optimal schedules found in this chapter for the exemplary DFG will use the same scheduling graph in Figure 5.5.

For the integral schedule S^i, we have

$$\begin{aligned}
S^i(v, i+f) &= \lceil -sh(v) \cdot c/f + (i+f) \cdot c/f \rceil \\
&= \lceil -sh(v) \cdot c/f + i \cdot c/f + c \rceil \\
&= \lceil -sh(v) \cdot c/f + i \cdot c/f \rceil + c \\
&= S^i(v, i) + c.
\end{aligned}$$

□

The entire scheduling algorithm is shown in Figure 5.7. The single-source shortest path algorithm introduced in [75] is adopted. If there is a cycle with negative weight in graph G^s, the algorithm will detect it; otherwise, the algorithm will compute the shortest paths.

It takes time $O(|V|\,|E|)$ to compute $sh(v)$, and time $O(f\,|V|)$ to generate a schedule with unfolding factor f. The time complexity of the scheduling algorithm is $O((|E| + f)\,|V|)$. This algorithm is particularly efficient because our scheduling algorithm for finding schedules with unfolding factor f works on the scheduling graph G^s, the size of which is the same as the original DFG. We do not unfold the DFGs while computing schedules.

The following theorem provides the necessary and sufficient condition for the existence of a schedule with cycle period c and unfolding factor f.

Theorem 5.3 *Let G be a general-time DFG, c a (fractional) cycle period, and f an unfolding factor. $c/f \geq \mathcal{B}(G)$ if and only if there exists a legal fractional schedule with unfolding factor f and cycle period c.*

Proof: Although the cycle period c can be fractional and G is a general-time DFG, the proof of this lemma is analogous to that of Lemma 4.2.3 for a unit-time DFG.

Since $\mathcal{B}(G)$ is the lower bound of iteration period, we know every legal schedule must satisfy the inequality $c/f \geq \mathcal{B}(G)$. Therefore, the if part is proved. Assume that $c/f \geq \mathcal{B}(G)$. From Lemma 5.3.1, we know there is no cycles having negative weights in G^s. Thus, we can find the value $sh(v)$ for every node v by the shortest path algorithm. From these values, we can generate a legal fractional schedule with

Scheduling Algorithm
Input : a DFG G, an unfolding factor f and a cycle period c.
Output : A schedule for the first f iterations such that
every f iterations can start in an interval of length c.

```
begin /* Q is a first-in first-out queue. pass is used to prevent
  the algorithm being trapped in negative cycles. */
    for every node v ∈ V do begin
        sh(v) ⟵ 0;    PUSH v to Q;
    end
    pass ⟵ 0;
    last ⟵ the last element in Q;
    while Q is not empty and pass < |V| do begin
        Pop v from Q
        for every edge e = (v, w) ∈ E do begin
            sh(w) ⟵ min{sh(v) + d(e) − t(v) · f/c, sh(w)}
            if w is not in Q then    PUSH w to Q;
        end
        if v = last then begin
            pass ⟵ pass + 1;
            last ⟵ the last element in Q;
        end
    end
    if Q is empty then
        for i = 0 to f − 1 do
            for every v ∈ V do
                S^f(v, i) ⟵ −sh(v) · c/f + i · c/f
                S^i(v, i) ⟵ ⌈−sh(v) · c/f + i · c/f⌉
    else "No such schedule exists."
end
```

Figure 5.7: Scheduling Algorithm

unfolding factor f and cycle period c from Lemma 5.2-(a). Therefore, the only-if part is proved. $\square$

By using Lemma 5.2-(b) as a legal integral schedule, similar result can be obtained for the integral model.

Theorem 5.4 *Let G be a general-time DFG, c a (integral) cycle period, and f an unfolding factor. $c/f \geq \mathcal{B}(G)$ if and only if there exists a legal integral schedule with unfolding factor f and cycle period c.*

5.4 Integral-Grid Model

All legal schedules can be implemented as static schedules by using pipelined processors. In Chapter 4 we modeled a general-time DFG as a unit-time DFG to produce schedules for a pipelined design. Analogously, the minimum rate-optimal unfolding factors for general-time DFGs under the pipelined implementation can be derived.

Theorem 5.5 *Let G be a general-time DFG and ρ the denominator of $\mathcal{B}(G)$ in its irreducible form. The minimum rate-optimal unfolding factor under the integral model and pipelined implementation is ρ.*

Consider the example in Figure 5.4. Since $\mathcal{B}(G)$ is $3/2$, we know ρ is 2. Figure 5.8 shows a rate-optimal schedule with unfolding factor 2. From Theorem 5.5, we know that 2 is the minimum rate-optimal unfolding factor under the integral model. Figure 5.9 shows the corresponding static schedule.

Suppose we want to find a rate-optimal schedule. We can first find the iteration bound $\mathcal{B}(G)$ in its irreducible form. The iteration bound for a DFG can be found in time $O(|V||E|\log|V|)$ [46]. Then, we can use the scheduling algorithm to find a schedule with unfolding factor f and cycle period c in time $O((|E|+f)|V|)$.

It is possible that this minimum rate-optimal unfolding factor is still too large for the design requirement in some cases. The designer may want to trade off between an unfolding factor and the corresponding minimum iteration period. For an unfolding factor f, the minimum feasible c from the inequality $c/f \geq \mathcal{B}(G)$ is $MCP_p(f) = \lceil f \cdot \mathcal{B}(G) \rceil$. Thus, all pairs of f and $MCP_p(f)$ are easily generated, and the designer

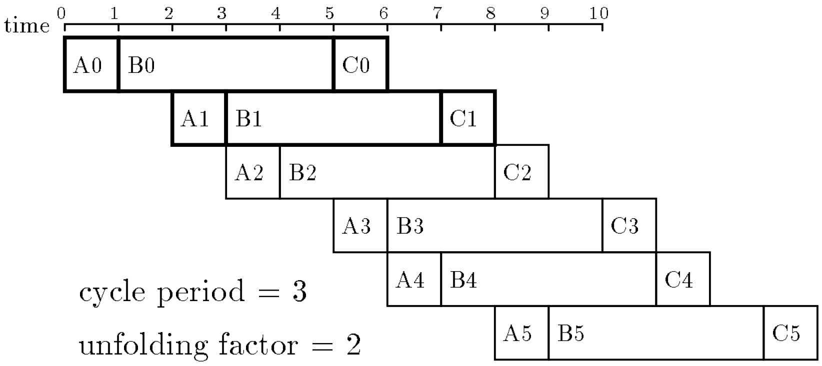


Figure 5.8: Integral model and pipelined implementation: time table

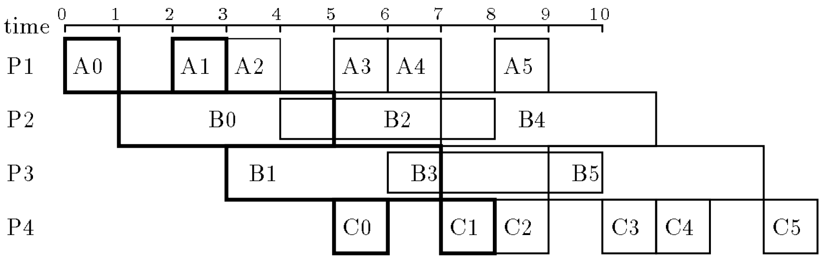


Figure 5.9: Integral model and pipelined implementation: static schedule

may choose the suitable pair according to the design requirement. We summarize this in the following corollary.

Corollary 5.4.1 *Let G be a DFG and f a given unfolding factor. The pipelined schedule using unfolding factor f has the minimum cycle period*

$$MCP_p(f) = \lceil f \cdot \mathcal{B}(G) \rceil \, .$$

The larger the unfolding factor is, the larger will be the memory used to store the static schedule. Under design requirements, we usually have a limit on the amount of memory available. The following corollary finds the unfolding factor and cycle period which gives the minimum iteration period, i.e. the average computation time per iteration.

Corollary 5.4.2 *Let G be a DFG and F the maximum unfolding factor feasible under resource requirements. The pipelined schedule with minimum iteration period using* *an unfolding factor not exceeding F has unfolding factor $\hat{f}$ and cycle period $\hat{c}$, where $\hat{c}/\hat{f}$ is the irreducible form of the value*

$$\min_{1 \leq f \leq F} \frac{\lceil f \cdot \mathcal{B}(G) \rceil}{f}.$$

For example, for a DFG with $\mathcal{B}(G) = 15/11$, the minimum rate-optimal unfolding factor is 11. But, the maximum feasible unfolding factor under design requirements is only 6. From the above corollary, we know the minimum iteration period is obtained from $\hat{f} = 5$ and $\hat{c} = 7$.

Next we discusses the static schedules for non-pipelined processors under the integral model. The following theorem combines the requirements for rate-optimality (Theorem 5.4) and non-pipelined design (Theorem 5.1) to derive the rate-optimal minimum unfolding factor.

Theorem 5.6 *Let σ and ρ be the numerator and denominator of the iteration bound $\mathcal{B}(G)$ in its irreducible form. Set*

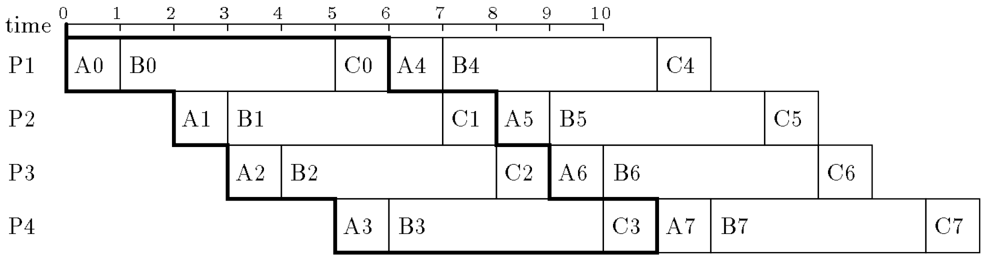
$$\hat{f} = \left\lceil \frac{\frac{\max_v t(v)}{\mathcal{B}(G)}}{\rho} \right\rceil \cdot \rho = \left\lceil \frac{\max_v t(v)}{\sigma} \right\rceil \cdot \rho.$$

(a) *There exists a rate-optimal static schedule with unfolding factor $\hat{f}$ under non-pipelined design.*

(b) *$\hat{f}$ is the minimum rate-optimal unfolding factor to deliver a static schedule under non-pipelined design.*

Proof: In a rate-optimal schedule with unfolding factor $\hat{f}$, the cycle period $\hat{c}$ is $\hat{f} \cdot \mathcal{B}(G)$. From Theorem 5.4, we know that there exists a legal schedule with cycle period $\hat{c}$ and unfolding factor $\hat{f}$. From the definition of $\hat{f}$, we know that $\hat{f} \geq \max_v t(v)/\mathcal{B}(G)$. Thus, we have $\hat{c} \geq \max_v t(v)$. From Theorem 5.1, we know that this rate-optimal schedule is a static schedule under non-pipelined design.

Next, we show that $\hat{f}$ is the minimum rate-optimal unfolding factor. For any rate-optimal unfolding factor f smaller than $\hat{f}$, the cycle period c is $f \cdot \mathcal{B}(G)$. We consider the following two cases:



cycle period = 6, unfolding factor = 4

Figure 5.10: Integral model and non-pipelined implementation

- $f < \dfrac{\max_v t(v)}{\mathcal{B}(G)}$.
  Since the cycle period c is smaller than $\max_v t(v)$, From Theorem 5.1, there does not exist any non-pipelined schedule for unfolding factor f.
- $\dfrac{\max_v t(v)}{\mathcal{B}(G)} \le f < \hat{f}$.
  A valid schedule with unfolding factor f under the integral model must have an integral cycle period c. Since $c = f \cdot \sigma/\rho$ is integral, there exists an integer k such that $f = k\ \rho$. We have

  $$\begin{array}{ccccl} \dfrac{\max_v t(v)}{\sigma/\rho} & \le & k\ \rho & < & \left\lceil \dfrac{\max_v t(v)}{\sigma} \right\rceil \cdot \rho, \text{ or} \\ \dfrac{\max_v t(v)}{\sigma} & \le & k & < & \left\lceil \dfrac{\max_v t(v)}{\sigma} \right\rceil \end{array}$$

  Thus, we know that in this range of f, no integral k exists such that the cycle period c is integral.

Therefore, $\hat{f}$ is the minimum unfolding factor to deliver a rate-optimal static schedule under non-pipelined implementation. □

Our scheduling algorithm can find a schedule S^i. From Theorem 5.1, this schedule is a static schedule under non-pipelined design. Consider the example in Figure 5.4. The iteration bound $B(G)$ is $3/2$, and $\max_v t(v)$ is 4. From Theorem 5.6, we derive the minimum rate-optimal unfolding factor as

$$\hat{f} = \left\lceil \frac{\max_v t(v)}{\sigma} \right\rceil \cdot \rho = \left\lceil \frac{4}{3} \right\rceil \cdot 2 = 4.$$

From the scheduling algorithm in Figure 5.7, we obtain $S^i(A,0) = 0$, $S^i(B,0) = 1$, $S^i(C,0) = 5$, $S^i(A,1) = \lceil 3/2 \rceil = 2$, $S^i(B,1) = \lceil 1 + 3/2 \rceil = 3$, and $S^i(C,1) = \lceil 5 + 3/2 \rceil = 7$. Figure 5.10 shows the rate-optimal schedule with unfolding factor 4. This schedule is static under non-pipelined design, where the operation B5 starts after B1 is finished.

In practice, the minimum unfolding factor to achieve rate-optimal non-pipelined design may be too large to be realistic. It is necessary to find a schedule of the highest rate achievable under practical design requirements.

Lemma 5.4.3 *Let f be a given unfolding factor. The static schedule under non-pipelined design using unfolding factor f has the minimum cycle period $MCP_n(f)$, where*

$$MCP_n(f) = \begin{cases} \max_v t(v) & \textit{if } \max_v t(v) \geq f \cdot \mathcal{B}(G), \\ \lceil f \cdot \mathcal{B}(G) \rceil & \textit{otherwise.} \end{cases} .$$

Proof: In order to have a non-pipelined implementation of cycle period c and unfolding factor f, the following inequalities have to be satisfied: $c/f \geq \mathcal{B}(G)$ (Theorem 5.4) and $c \geq \max_v t(v)$ (Theorem 5.1). It is easy to show that $MCP_n(f)$ is the minimum cycle period satisfying these inequalities in both cases. $\Box$

For example, given a DFG with $\mathcal{B}(G) = 15/11$, $f = 5$, if $\max_v t(v) = 8$, we have $MCP_n(5)$ is 8; if $\max_v t(v) = 6$, we have $MCP_n(5)$ is 7.

Theorem 5.7 *Let F be the maximum unfolding factor feasible under resource constraints. The non-pipelined schedule with the minimum iteration period using an unfolding factor not exceeding F has minimum unfolding factor $\hat{f}$ and cycle period $\hat{c}$. The values of $\hat{f}$ and $\hat{c}$ can be found as follows.*

(a) *If $\max_v t(v) \geq F \cdot \mathcal{B}(G)$, we have $\hat{f} = F$ and $\hat{c} = \max_v t(v)$.*

(b) *If $\max_v t(v) < F \cdot \mathcal{B}(G)$, the minimum iteration period $\hat{c}/\hat{f}$ is the irreducible form of the value*

$$\min\{\frac{\max_v t(v)}{\left\lfloor \frac{\max_v t(v)}{\mathcal{B}(G)} \right\rfloor}, \min_{\frac{\max_v t(v)}{\mathcal{B}(G)} < f \leq F} \frac{\lceil f \cdot \mathcal{B}(G) \rceil}{f}\}.$$

Proof: We need to prove that $MCP_n(F)/F$ is the minimum iteration period achievable and F is the minimum unfolding factor capable of achieving the iteration period $MCP_n(F)/F$.

Consider an unfolding factor f smaller than F. In case that $\max_v t(v) \geq F \cdot \mathcal{B}(G)$, we know $\max_v t(v) \geq f \cdot \mathcal{B}(G)$ and $MCP_n(f) = MCP_n(F)$. Thus, the iteration period achieved by unfolding factor f is $MCP_n(F)/f$, which is larger than $MCP_n(F)/F$. The unfolding factor f could not achieve the minimum iteration period.

In case that $\max_v t(v) < F \cdot \mathcal{B}(G)$, we have two cases:

- $\max_v t(v) \geq f \cdot \mathcal{B}(G)$.
  The larger the f, the better the iteration period achieved. The best iteration period for all f such that $\max_v t(v) \geq f \cdot \mathcal{B}(G)$ is $\hat{c}/\hat{f} = \max_v t(v) / \left\lfloor \frac{\max_v t(v)}{\mathcal{B}(G)} \right\rfloor$.
- $\max_v t(v) < f \cdot \mathcal{B}(G)$.
  Under this case, the minimum cycle period for non-pipelined design is $\lceil f \cdot \mathcal{B}(G) \rceil / f$, which the same as that for pipelined design. Since the minimum iteration period for every unfolding factor has to be examined, the second term of the minimum function is served for this purpose.

Therefore, the minimum iteration period under all cases is the minimum within the resource constraints. □

For example, given a DFG with $\mathcal{B}(G) = 15/11$, $F = 6$, and $\max_v t(v) = 4$, we have $\max_v t(v) = 4 < 6 \cdot 15/11$. Thus, we use the formula in **(b)**. The first term is $\frac{4}{\left\lfloor \frac{4}{15/11} \right\rfloor} = 4/2 = 2$; the second term is $\min_{2<f\leq 6} \lceil f \cdot 15/11 \rceil / f = 7/5$. Therefore, $\hat{c} = 7$ and $\hat{f} = 5$.

5.5 Fractional Model

Under some implementation models, the starting time of a node can be any fractional number, i.e. the schedule can have an arbitrarily fine grid. We can reduce the unfolding factor by using a finer time slice.

We use our algorithm to find a schedule with a fractional cycle period c and an (integral) unfolding factor f . First, we find the smallest integer ρ such that ρc is integral, i.e. ρ is the denominator of c in its irreducible form. Then, we use a finer time slice: $1/\rho$ time unit is a time slice. All the computation times in the DFG are modified to refer to time slices. Thus, we have a new DFG $G' = (V, E, d, t')$, where $t'(v) = \rho \cdot t(v)$. The lower bound of the new DFG becomes $\mathcal{B}(G') = \rho \cdot \mathcal{B}(G)$. Thus, we can use the theorems for the integral model to compute rate-optimal unfolding factors.

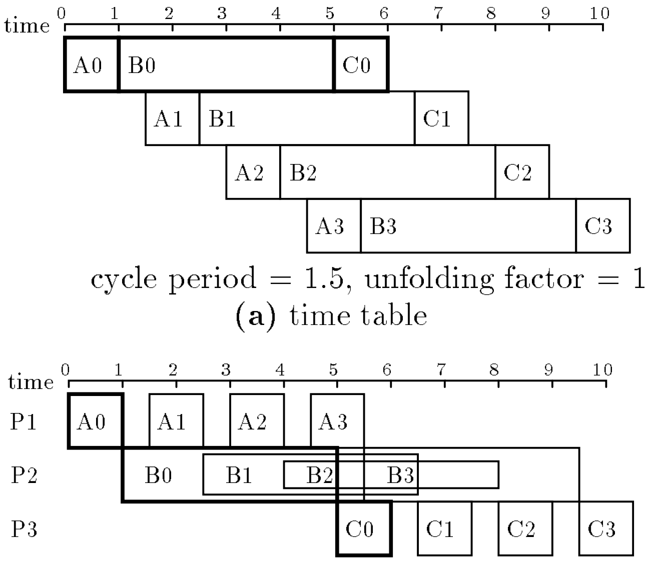
By using the fractional model, we can reduce the unfolding factor caused by the iteration bound $\mathcal{B}(G)$. The new iteration bound is $\rho \cdot \mathcal{B}(G)$. If we choose the right value for ρ, we can reduce the denominator of the iteration bound by a factor of ρ. Thus, under pipelined design, the rate-optimal unfolding factor is reduced by a factor of ρ. Theoretically, we can always choose ρ to be the denominator of $\mathcal{B}(G)$ in its irreducible form so that a static schedule without unfolding can be found. For the non-pipelined design, Theorem 5.1 gives the necessary and sufficient condition on the existence of a static schedule.

However, the fractional model does not relieve the inequality for non-pipelined design $c \geq \max_v t(v)$.

The minimum rate-optimal unfolding factor for pipelined and non-pipelined design under the fractional model are given in Theorem 5.8 and Theorem 5.9.

Theorem 5.8 *Under the fractional model and pipelined design, there always exists a rate-optimal pipelined static schedule without unfolding.*

Proof: Let σ and ρ be the numerator and denominator of the iteration bound $\mathcal{B}(G)$ in its irreducible form. Let $1/\rho$ be the size of a time slice. We apply Theorem 5.5 on the modified DFG G'. The iteration bound of G', $\mathcal{B}(G')$, is $\rho\mathcal{B}(G) = \sigma$, which is an integer. Thus, the numerator and denominator of $\mathcal{B}(G')$ are σ and 1, respectively. Since the minimum rate-optimal unfolding factor is 1, there exists a rate-optimal schedule without unfolding. $\square$



cycle period = 1.5, unfolding factor = 1
(a) time table

cycle period = 1.5, unfolding factor = 1
(b) static schedule

Figure 5.11: Fractional model and pipelined implementation

Theorem 5.9 *The minimum unfolding factor to deliver a rate-optimal static schedule under non-pipelined implementation on the fractional model is*

$$\hat{f} = \left\lceil \frac{\max_v t(v)}{\mathcal{B}(G)} \right\rceil .$$

Proof: Let σ and ρ be the numerator and denominator of the iteration bound $\mathcal{B}(G)$ in its irreducible form. Let $1/\rho$ be the size of a time slice. We apply Theorem 5.6 on the modified DFG G'. As shown in Theorem 5.8, the numerator and denominator of $\mathcal{B}(G')$ are σ and 1, respectively. Thus, the minimum unfolding factor $\hat{f}$ is

$$\left\lceil \frac{\max_v \rho\, t(v)}{\sigma} \right\rceil \cdot 1 = \left\lceil \frac{\max_v t(v)}{\sigma/\rho} \right\rceil = \left\lceil \frac{\max_v t(v)}{\mathcal{B}(G)} \right\rceil .$$

□

Consider the example in Figure 5.4. Under the fractional model and pipelined design, Theorem 5.8 shows that there always exists a rate-optimal schedule without unfolding. The rate-optimal schedule obtained by our scheduling algorithm is shown in Figure 5.11. Under the fractional model and non-pipelined design, we derive the rate-optimal minimum unfolding factor from Theorem 5.9 is

$$\hat{f} = \left\lceil \frac{\max_v t(v)}{\mathcal{B}(G)} \right\rceil = \left\lceil \frac{4}{3/2} \right\rceil = 3.$$

Figure 5.6 shows the rate-optimal schedule with unfolding factor 3 obtained by our scheduling algorithm.

We still measure the cycle period in terms of time units, instead of time slices. The minimum cycle periods for a given unfolding factor under the fractional model are similar to the ones under the integral model for either pipelined or non-pipelined design.

Lemma 5.5.1 *Let G be a DFG and f be a given unfolding factor.*

(a) *The pipelined schedule using unfolding factor f has the minimum cycle period $MCP_p(f) = f \cdot \mathcal{B}(G)$.*

(b) *The non-pipelined schedule using unfolding factor f has the minimum cycle period*

$$MCP_n(f) = \begin{cases} \max_v t(v) & \textit{if } \max_v t(v) \geq f \cdot \mathcal{B}(G), \\ f \cdot \mathcal{B}(G) & \textit{otherwise.} \end{cases} .$$

Proof: From Corollary 5.4.1, for pipelined design the minimum cycle period in terms of time slices is $\lceil f \cdot \rho\, \mathcal{B}(G) \rceil$. Since $\rho\mathcal{B}(G)$ is integral, the minimum cycle period in terms of time slices is $\rho \cdot f\mathcal{B}(G)$, which is $f \cdot \mathcal{B}(G)$ in terms of time units. Similarly, from Lemma 5.4.3, we can derive the minimum cycle period in terms of time units for non-pipelined design. □

From the above lemma, we can derive the minimum iteration period under a maximum unfolding factor F for non-pipelined design in the following lemma. For pipelined design, the minimum rate-optimal unfolding factor is already 1, so we need not consider the case of smaller F.

Theorem 5.10 *Let F be the maximum unfolding factor feasible under resource requirements. The non-pipelined schedule with the minimum iteration period using an unfolding factor not exceeding F has minimum unfolding factor $\hat{f}$ and cycle period $\hat{c}$.*

(a) *If $\max_v t(v) \geq F \cdot \mathcal{B}(G)$, we have $\hat{f} = F$ and $\hat{c} = \max_v t(v)$.*

(b) *If* $\max_v t(v) < F \cdot \mathcal{B}(G)$, $\hat{f}$ *equals the minimum rate-optimal unfolding factor* $\left\lceil \frac{\max_v t(v)}{\mathcal{B}(G)} \right\rceil$ *and* $\hat{c} = \hat{f} \cdot \mathcal{B}(G)$.

5.6 Conclusion

This chapter described a unified static scheduling technique on two models and two implementation styles. All the minimum rate-optimal unfolding factors under these four combination were obtained. Given a DFG with the iteration bound $\mathcal{B}(G) = \sigma/\rho$ in its irreducible form, we summarize the minimum rate-optimal unfolding factors in the following table.

Min Rate-optimal Unfolding Factor	Timing Models	
	Fractional	Integral
Pipelined Design	1	ρ
Nonpipelined Design	$\left\lceil \dfrac{\max_v t(v)}{\mathcal{B}(G)} \right\rceil$	$\left\lceil \dfrac{\max_v t(v)}{\sigma} \right\rceil \cdot \rho$

For a given maximum unfolding factor, we can compute the corresponding best legal static schedule. An efficient polynomial-time scheduling algorithm to compute a static schedule with unfolding factor f was presented which works on the original DFG instead of the larger unfolded DFG. We should point out that this algorithm has performed the effect of retiming implicitly; in other words, explicit retiming cannot get better results. Therefore, the results in this chapter still hold even if explicit retiming is performed. This chapter lays a theoretical foundation for static scheduling, and has potential to be generalized to other scheduling models.

The static schedule generated for the software pipelining problem in Section 4.4 is actually under the integral timing model and pipelined design, since only the issue times of operations are considered. For this reason, the theorems in Chapter 4 appear again in Section 5.4. For a cycle period c, the algorithm in Section 4 generates the repeating pattern of length c directly, while the algorithm in this chapter generates a schedule to be overlapped with initiation interval c. These two results actually correspond to the same schedule.

In order to obtain non-pipelined implementation under the model of Chapter 4, we can add a self cycle of f delay for every node if we are seeking a schedule with unfolding factor f. Then, the algorithm in Chapter 4 can be applied to find schedules under non-pipelined implementation.

Chapter 6

General-Time Data-Flow Graphs

In the previous chapter, an algorithm for pipeline scheduling of general-time DFGs is designed by using the concept of retiming on general-time DFGs with splitable nodes. This chapter is concerned with retiming and unfolding on general-time DFGs when node splitting is not allowed. The schedules generated have the property that the execution period of any node does not span across the boundary of the repeating schedule, as mentioned in Section 2.6.

Chapter 3 showed that the order of retiming and unfolding is immaterial. Thus, we use the approach of retiming first, then unfolding. In this chapter, two retiming algorithms are designed for general-time data-flow graphs with implicit unfolding to find a retiming function to achieve a cycle period c for a given unfolding factor f. More precisely, for a general-time data-flow graph G, we would like to find a retiming r such that the cycle period of the unfolded retimed DFG, denoted by $\mathcal{C}(G_{r,f})$, is no more than c for a given c and unfolding factor f, without explicitly constructing the unfolded graph.

When one cannot put delays arbitrarily in the middle of a node, the retiming of general-time DFGs has less freedom and cannot use extensions of the results for unit-time DFGs. The design of retiming algorithms for general-time DFGs is intrinsically harder. For unit-time DFGs in Chapter 4 or general-time DFGs with splitable nodes in 5, the minimum cycle period can be found by a simple formula of the iteration bound $\mathcal{B}(G)$ and the given unfolding factor. For a general-time DFG without node splitting, the minimum cycle periods cannot be characterized in a simple formula because one cannot always place delays evenly in a cycle. In order to find the minimum cycle period, a preprocessing step is performed to fine all possible values of the cycle periods. Then, a binary search is performed to find the minimum cycle period.

When unfolding is not involved, Leiserson and Saxe presented retiming algorithms to find the minimum cycle period over all retimings [53]. To find a retiming achieving a given cycle period, an $O(|V|^3)$-time retiming algorithm is presented with $O(|V|^3)$ or $O(|V|\,|E| + |V|^2 \log |V|)$ preprocessing time (from a choice of all-pairs shortest path algorithms). Another $O(|V|\,|E|)$-time retiming algorithm is also designed without preprocessing. To achieve a single cycle period, the second algorithm is always a better choice since the preprocessing is not needed. In order to find the minimum cycle period, the same preprocessing step has to be performed for both algorithms. To find the minimum cycle period by binary search, the first algorithm takes $O(|V|^3 \log |V|)$ total time, and the second algorithm takes $O(|V|\,|E| \log |V|)$ total time. For sparse DFGs, the second algorithm performs better. However, for dense DFGs, the first algorithm is computationally more efficient because of the preprocessing. Notice that the computational complexities are dominated by the retiming algorithm with binary search.

This chapter will follow similar approachs to design two retiming algorithms with implicit unfolding: one for dense DFGs and the other for sparse DFGs. For a given cycle period c and a given unfolding factor f, with $O(f^2|V|\,|E| + f^2|V|^2 \log(f\,|V|))$ preprocessing time, an $O(f|V|^2 + |V|^3)$-time retiming algorithm is presented for dense DFGs. Another $O(f\,|V|\,|E|)$-time retiming algorithm is also designed for sparse DFGs. The second algorithm performs better if the cycle period and the unfolding factor are given. In order to find the minimum cycle period for a given unfolding factor, the same preprocessing step has to be performed for both algorithms. To find the minimum cycle period by binary search, the total retiming time for the first algorithm is $O((f\,|V|^2 + |V|^3) \log |V|)$; that for the second algorithm is $O(f\,|V|\,|E| \log(f\,|V|))$. The preprocessing step dominates the time complexities of both algorithms. Thus, for a given unfolding factor, the minimum cycle period can be found in time $O(f^2|V|\,|E| + f^2|V|^2 \log(f\,|V|))$. In terms of the retiming time only, the first algorithm is more efficient asymptotically, and the second algorithm is more efficient for sparse DFGs.

The size of program code or a control unit is proportional to the unfolding factor. Let F be the maximum value of the unfolding factor allowed under resource constraints. When we want to obtain all the pairs of unfolding factors and the corresponding minimum iteration periods, our preprocessing algorithm needs to be performed only once for the maximum unfolding factor F instead of F times.

6.1 An Algorithm for Dense Data-Flow Graphs

The approach we use is analogous to that of [53] for retiming only. A preprocessing algorithm is performed first in order to find a set of critical paths, which is presented in Section 6.2. In this section, some properties of these critical paths are derived in order to represent the above problem in a simple Integer Linear Programming (ILP) form, which can be solved in time $O(|V|^3)$ by a general shortest path algorithm. Then, by binary search on all the $O(f|V|^2)$ possible cycle periods, we can find a legal retiming r which achieves the minimum cycle period on DFG $G_{r,f}$ in time $O((f|V|^2+|V|^3)\log(f|V|))$ for a given unfolding factor f.

Since the delay count of every path from u to v is changed by the same amount $r(u)-r(v)$ after the DFG is retimed by r, the quantities defined below specify a set of critical paths for all paths from u to v, in such a way that we only need to look at these quantities in order to decide whether $\mathcal{C}(G_f)\leq c$.

Definition 6.1 *Let u and v be nodes in G and s an integer where $0\leq s<f$.*

(a) *Define $\mathcal{D}(u,v)=\min\{d(p)\mid p \text{ is a path from } u \text{ to } v \text{ in } G\}$.*

(b) *Define $\mathcal{T}^{<s>}(u,v)=\max\{t(p)\mid p \text{ is a path from } u \text{ to } v \text{ with } d(p)=\mathcal{D}(u,v)+s\}$.*

(When there is no path $u\overset{p}{\rightsquigarrow}v$, $\mathcal{D}(u,v)$ is undefined and $\mathcal{T}^{<s>}(u,v)=0$. When there is no path $u\overset{p}{\rightsquigarrow}v$ such that $d(p)=\mathcal{D}(u,v)+s$, we define $\mathcal{T}^{<s>}(u,v)=0$.)

For an integer s, the value $\mathcal{D}(u,v)$ is the minimum delay count of all the paths from u to v. The value $\mathcal{T}^{<s>}(u,v)$ is the maximum computation time among all the paths from u to v where the delay counts are equal to $\mathcal{D}(u,v)+s$. The paths $u\overset{p}{\rightsquigarrow}v$

such that $d(p) = \mathcal{D}(u,v) + s$ and $t(p) = \mathcal{T}^{<s>}(u,v)$ are called *critical paths*. The preprocessing algorithm described in Section 6.2 computes $\mathcal{D}(u,v)$ and $\mathcal{T}^{<s>}(u,v)$ for every u and v in V and every $0 \le s < f$.

Theorem 6.1 *Let $G = (V, E, t, d)$ be a DFG, c be a cycle period and f be an unfolding factor. Then $\mathcal{C}(G_f) \le c$* if and only if *for all nodes u and v in V and for all s where $0 \le s < f$ if $\mathcal{T}^{<s>}(u,v) > c$ then $\mathcal{D}(u,v) + s \ge f$.*

Proof: To prove the only-if part by contradiction, we assume $\mathcal{C}(G_f) \le c$, and that there exist $u, v \in V$ and $s \in [0, f)$ such that $\mathcal{D}(u,v) + s < f$ and $\mathcal{T}^{<s>}(u,v) > c$. Let p be a critical path $u \overset{p}{\rightsquigarrow} v$ with $d(p) = \mathcal{D}(u,v) + s$ and $t(p) = \mathcal{T}^{<s>}(u,v)$. Thus, there exists a path p in G such that $d(p) < f$ and $t(p) > c$. From Lemma 3.3.2, we know $\mathcal{C}(G_f) > c$. This is a contradiction.

Now we prove the if-part. Assume that for all $u, v \in V$ and for all s where $0 \le s < f$ if $\mathcal{D}(u,v) + s < f$ then $\mathcal{T}^{<s>}(u,v) \le c$. Under this assumption we claim that for every path p in G if $d(p) < f$ then $t(p) \le c$, which implies $\mathcal{C}(G_f) \le c$. Let $u \overset{p}{\rightsquigarrow} v$ be a path in G. If p is a critical path, the claim is true. Otherwise, we want to show that $d(p) \ge f$ or $t(p) \le c$. In case that $d(p) \ge \mathcal{D}(u,v) + f$, we know $d(p) \ge f$ since $\mathcal{D}(u,v) \ge 0$. Otherwise, $d(p) = \mathcal{D}(u,v) + s$ where $0 \le s < f$. Since p is not a critical path, we have $t(p) \le \mathcal{T}^{<s>}(u,v)$. From the assumption, we know that either $d(p) \ge f$ or $t(p) \le c$. Therefore, the claim is proved. □

The next theorem gives us the necessary and sufficient conditions for the existence of a retiming r with $\mathcal{C}(G_{r,f}) \le c$ in terms of $\mathcal{D}$ and $\mathcal{T}^{<s>}$. From this theorem we are able to construct a simple ILP form for retiming and unfolding.

Theorem 6.2 *Let $G = (V, E, t, d)$ be a DFG, c a cycle period and f an unfolding factor. The following two statements are equivalent:*

(a) *r is a retiming on G such that $\mathcal{C}(G_{r,f}) \le c$.*

(b) *for all nodes u and v in V and for all s where $0 \le s < f$ if $\mathcal{T}^{<s>}(u,v) > c$ then $\mathcal{D}(u,v) + r(u) - r(v) + s \ge f$.*

Proof: Let $\mathcal{D}_r(u,v)$ and $\mathcal{T}_r^{<s>}(u,v)$ be similarly defined on G_r. First, we prove that the effect of retiming on the functions $\mathcal{D}(u,v)$ and $\mathcal{T}^{<s>}(u,v)$ is similar to that on the functions d and t, i.e.

(a) $\mathcal{D}_r(u,v) = \mathcal{D}(u,v) + r(u) - r(v)$.

(b) $\mathcal{T}_r^{<s>}(u,v) = \mathcal{T}^{<s>}(u,v)$.

Then, this theorem is easily derived from Theorem 6.1 on the retimed graph G_r.

For part (a), $\mathcal{D}_r(u,v)$ is defined as $\min\{d_r(p) \mid u \overset{p}{\rightsquigarrow} v\}$. Since $d_r(p) = d(p) + r(u) - r(v)$, we have

$$\begin{aligned} \mathcal{D}_r(u,v) &= r(u) - r(v) + \min\{d(p) \mid u \overset{p}{\rightsquigarrow} v\} \\ &= r(u) - r(v) + \mathcal{D}(u,v). \end{aligned}$$

Part (a) implies Part (b):

$$\begin{aligned} \mathcal{T}_r^{<s>}(u,v) &= \max\{t(p) \mid u \overset{p}{\rightsquigarrow} v \text{ with } d_r(p) = \mathcal{D}_r(u,v) + s\} \\ &= \max\{t(p) \mid u \overset{p}{\rightsquigarrow} v \text{ with } d(p) + r(u) - r(v) = \mathcal{D}_r(u,v) + s\} \\ &= \max\{t(p) \mid u \overset{p}{\rightsquigarrow} v \text{ with } d(p) = \mathcal{D}(u,v) + s\} \\ &= \mathcal{T}^{<s>}(u,v). \end{aligned}$$

$\Box$

Theorem 6.3 *Let $G = (V, E, t, d)$ be a DFG, c a cycle period and f an unfolding factor. Then r is a legal retiming on G such that $\mathcal{C}(G_{r,f}) \le c$* if and only if

$$\begin{aligned} r(v) - r(u) &\le d(e) \quad \textit{for every } u \xrightarrow{e} v \textit{ of } G, \textit{ and} \\ r(v) - r(u) &\le \mathcal{D}(u,v) + s - f \quad \textit{for all } u, v \in V, \\ &\qquad\qquad 0 \le s < f \textit{ such that } \mathcal{T}^{<s>}(u,v) > c. \end{aligned}$$

Proof: The function r is a legal retiming if and only if $d_r(e) \ge 0$ for every e in G. Since $d_r(e) = d(e) + r(u) - r(v)$, r is a legal retiming if and only if $r(v) - r(u) \le d(e)$ for every $u \xrightarrow{e} v$. The second inequality comes from Theorem 6.2. $\Box$

For a particular cycle period c and an unfolding factor f, this theorem gives a simple ILP form to find a legal retiming r, if one exists, such that $\mathcal{C}(G_{r,f}) \le c$. This ILP form can be easily solved by single-source shortest path algorithms. Assume that the preprocessing of computing $\mathcal{D}$ and $\mathcal{T}^{<s>}$ for a given unfolding factor f is done.

The number of values $\mathcal{T}^{<s>}(u,v)$ is $O(f|V|^2)$. In time $O(f|V|^2)$, we can generate an ILP form for a given cycle period c by scanning the values of $\mathcal{T}^{<s>}(u,v)$. Since the inequalities which have the same left-hand side are covered by the one which has the smallest value on the right-hand side, we obtain an ILP form which has at most $|V|^2$ inequalities. Thus, the legal retiming can be found in time $O(|V|^3)$ by the Bellman-Ford algorithm for shortest-path problems. The Bellman-Ford algorithm from [75] is adopted as the retiming algorithm for dense general-time DFGs, and shown in Figure 6.1. Note that we modified the Bellman-Ford algorithm in [75] slightly so that the replacement of newly-found shortest paths is performed at the end of each phase to comply with the version of Bellman-Ford algorithm in [46]. This modification is done to facilitate the improvement in the Section 6.3 for sparse DFGs.

It is easy to observe that the cycle period of $G_{r,f}$ is a value of $\mathcal{T}^{<s>}(u,v)$ for a certain u, v and s. We can sort the set of values $\{\mathcal{T}^{<s>}(u,v) \mid \text{ every } u \text{ and } v \in G \text{ and every } s \in [0,f)\}$, and perform binary search on this set of values in order to find the minimum cycle period of $G_{r,f}$ over all retimings r. Therefore, the legal retiming such that $\mathcal{C}(G_{r,f})$ is the minimum can be found in time $O((f|V|^2+|V|^3)\log(f|V|))$ plus an $O(f^2|V|\,|E| + f^2|V|^2\log(f\,|V|))$ preprocessing time.

From the definitions, the functions $\mathcal{D}$ and $\mathcal{T}^{<s>}$ are independent of the unfolding factor. Let F be the maximum value of the unfolding factor allowed under resource constraints. After we perform the preprocessing step for the maximum unfolding factor F once in time $O(F^2|V|\;|E| + F^2|V|^2\log(F\;|V|))$, the values of $MCP(f)$ for any f, $f \le F$, can be found by using our retiming algorithm without further preprocessing for each f.

Although the minimum cycle period $MCP(f)$ is an increasing function, the behavior of the minimum iteration period $MIP(f)$, which is $MCP(f)/f$, is hard to characterize. In order to find the minimum iteration period $MIP(f)$ for every $f \le F$, we need to compute the minimum cycle period $MCP(f)$ for every f and then find the one such that $MCP(f)/f$ is minimized. Thus, the minimum $MIP(f)$ for all f, where $f \le F$, can be found in time $O(F^2|V|^3\log(F|V|))$.

As discussed in Section 4.3, we can control the length of prologue and epilogue by adding the inequalities $r(u) \ge 0$ and $r(u) \le \mathsf{p}$ for every u in V, where p is the

```
Retiming Algorithm 1
Input :   a DFG G, an unfolding factor f, a cycle period c,
          and the functions D and T^<s>.
Output :  A retiming r such that C(G_{r,f}) ≤ c.
Step 1.   /* Construct a constraint graph G' = (V, E', w)
                from the ILP form in Theorem 6.3 */
          E' ⟵ E;   w(e) ⟵ d(e) for every e ∈ E;
          for every u and v in V and every 0 ≤ s < f do
              if T^<s>(u, v) > c then
                  if e = (u, v) ∉ E' then begin
                      Add e to E';   w(e) ⟵ D(u, v) + s − f;   end ;
                  else if w(e) > D(u, v) + s − f then
                      w(e) ⟵ D(u, v) + s − f;
Step 2.   /* phase is used to terminate the algorithm
          when negative cycles are present. */
          Q ⟵ ∅;
          for every node v ∈ V do begin
              r(v) ⟵ 0; new_r(u) ⟵ 0;   PUSH v to Q;   end ;
          phase ⟵ 1;   newQ ⟵ ∅;
          while Q is not empty and phase ≤ |V| do begin
              Pop v from Q;
              for every edge e = (u, v) ∈ E' do begin
                  new_r(u) ⟵ max{r(v) + w(e), r(u)};
                  if u is not in newQ then   PUSH u to newQ;
              end ;
              if Q = ∅ then begin   /* Enter a new phase */
                  phase ⟵ phase + 1;
                  r(v) ⟵ new_r(v)   for every v in V.
                  Q ⟵ newQ;   newQ ⟵ ∅;   end ;
          end
Step 3.   if Q = ∅ then   Output the retiming function r;
          else Output "No such a retiming exists."
```

Figure 6.1: Retiming algorithm for dense general-time DFGs [46, 75]

desired prologue length. The modified ILP form can still be solved by shortest path algorithms. The number of iterations in the prologue and epilogue of the unfolded retimed program is no more than p.

6.2 Preprocessing Algorithm

The function $\mathcal{D}$ can be easily computed by Floyd-Warshall's all-pair shortest path algorithms in time $O(|V|^3)$ [46] or by Johnson's algorithm in time $O(|V|\,|E|+|V|^2\log|V|)$ [40]. We first compute the following two functions D_f and T_f defined on the unfolded graph G_f, and then show how to compute the function $\mathcal{T}^{<s>}$ from $\mathcal{D}$, D_f and T_f.

Definition 6.2 *Let $G_f = (V_f, E_f, d_f, t_f)$ be an unfolded graph, u^i and v^j nodes in G_f and s an integer where $0 \le s < f$. The function D_f is a function from $V_f \times V_f$ to the nonnegative integers, and T_f is a function from $V_f \times V_f$ to the positive integers.*

(a) *Define $\mathsf{D}_f(u^i, v^j) = \min\{d_f(p_f) \mid p_f$ is a path from u^i to v^j in $G_f\}$.*

(b) *Define $\mathsf{T}_f(u^i, v^j) = \max\{t_f(p_f) \mid p_f$ is a path from u^i to v^j with $d_f(p_f) = \mathsf{D}_f(u^i, v^j)\}$.*

(When there is no path $u^i \overset{p_f}{\rightsquigarrow} v^j$, $\mathsf{D}_f(u^i, v^j)$ is undefined and $\mathsf{T}_f(u^i, v^j) = 0$.)

With length measure $(d_f(p_f), -t_f(p_f))$ for a path, the functions D_f and T_f can be computed in time $O(|V|^3 f^3)$ by the Floyd-Warshall algorithm for all-pairs shortest path problems [46] or in time $O(f^2|V|\,|E|+f^2|V|^2\log(f\,|V|))$ by Johnson's algorithm [40].

The following lemma shows the relation between the functions $\mathcal{D}$ and D_f.

Lemma 6.2.1 *Let u and v be two nodes in V, f an unfolding factor, and s an integer where $0 \le s < f$. If there exists a path $u \overset{p}{\rightsquigarrow} v$ such that $d(p) = \mathcal{D}(u, v) + s$, $\mathsf{D}_f(u^0, v^{(\mathcal{D}(u,v)+s)\%f}) = \left\lfloor \frac{\mathcal{D}(u,v)+s}{f} \right\rfloor$.*

Proof:

First, we claim that the following property of D_f is true:

$$\mathsf{D}_f(u^0, v^{(\mathcal{D}(u,v)+s)\%f}) \ge \left\lfloor \frac{\mathcal{D}(u,v)+s}{f} \right\rfloor.$$

If the claim is true, we can prove the lemma as follows. Assume that there exists a path $u \overset{p}{\leadsto} v$ such that $d(p) = \mathcal{D}(u,v) + s$. From Property 3.1-(b) we know the corresponding path $u^0 \overset{p_f}{\leadsto} v^{(\mathcal{D}(u,v)+s)\%f}$ in G_f has delay $d_f(p_f) = \left\lfloor \frac{\mathcal{D}(u,v)+s}{f} \right\rfloor$. Thus, $\mathsf{D}_f(u^0, v^{(\mathcal{D}(u,v)+s)\%f}) \leq \left\lfloor \frac{\mathcal{D}(u,v)+s}{f} \right\rfloor$. From the above claim we have $\mathsf{D}_f(u^0, v^{(\mathcal{D}(u,v)+s)\%f}) = \left\lfloor \frac{\mathcal{D}(u,v)+s}{f} \right\rfloor$. Thus, the lemma is proved.

Now, we prove this claim by contradiction. Assume that $\mathsf{D}_f(u^0, v^{(\mathcal{D}(u,v)+s)\%f}) < \left\lfloor \frac{\mathcal{D}(u,v)+s}{f} \right\rfloor$. Then, there exists a path p'_f from u^0 to $v^{(\mathcal{D}(u,v)+s)\%f}$ such that $d_f(p'_f) = \mathsf{D}_f(u^0, v^{(\mathcal{D}(u,v)+s)\%f})$. Let path p' from u to v in G be the corresponding path of p'_f. From Property 3.1-(a), we know $d(p') = d_f(p'_f) \cdot f + (\mathcal{D}(u,v) + s) \,\%\, f$. From $\mathsf{D}_f(u^0, v^{(\mathcal{D}(u,v)+s)\%f}) < \left\lfloor \frac{\mathcal{D}(u,v)+s}{f} \right\rfloor \leq \left\lfloor \frac{\mathcal{D}(u,v)+s}{f} \right\rfloor - 1$, we derive

$$d(p') \leq \left\lfloor \frac{\mathcal{D}(u,v)+s}{f} \right\rfloor \cdot f - f + (\mathcal{D}(u,v) + s) \,\%\, f.$$

Let k and w be nonnegative integers such that $\mathcal{D}(u,v) = k \cdot f + w$, where $0 \leq w < f$. After substituting $\mathcal{D}(u,v)$ by $k \cdot f + w$, we derive

$$\begin{aligned} d(p') &\leq \left\lfloor \frac{k \cdot f + w + s}{f} \right\rfloor \cdot f - f + (k \cdot f + w + s) \,\%\, f \\ &= kf + \left\lfloor \frac{w+s}{f} \right\rfloor \cdot f - f + (w + s) \,\%\, f \\ &= kf + w + s - f \quad < \mathcal{D}(u,v). \end{aligned}$$

The last inequality comes from $0 \leq s < f$. The inequality $d(p') < \mathcal{D}(u,v)$ contradicts the definition of $\mathcal{D}(u,v)$. Thus, the claim is proved. $\square$

The following theorem shows us how to generate $\mathcal{T}^{<s>}(u,v)$ from $\mathcal{D}(u,v)$, $\mathsf{D}_f(u,v)$, and $\mathsf{T}_f(u,v)$.

Theorem 6.4 *Consider two nodes u and v in G. For every s where $0 \leq s < f$, we have*

$$\mathcal{T}^{<s>}(u,v) = \begin{cases} \mathsf{T}_f(u^0, v^{(\mathcal{D}(u,v)+s)\%f}) & \text{if } \mathsf{D}_f(u^0, v^{(\mathcal{D}(u,v)+s)\%f}) = \left\lfloor \frac{\mathcal{D}(u,v)+s}{f} \right\rfloor, \\ 0 & \text{otherwise.} \end{cases}$$

Proof: If $\mathsf{D}_f(u^0, v^{(\mathcal{D}(u,v)+s)\%f}) \neq \left\lfloor \frac{\mathcal{D}(u,v)+s}{f} \right\rfloor$, from Lemma 6.2.1, there is no path $u \overset{p}{\leadsto} v$ such that $d(p) = \mathcal{D}(u,v) + s$, and we have $\mathcal{T}^{<s>}(u,v) = 0$.

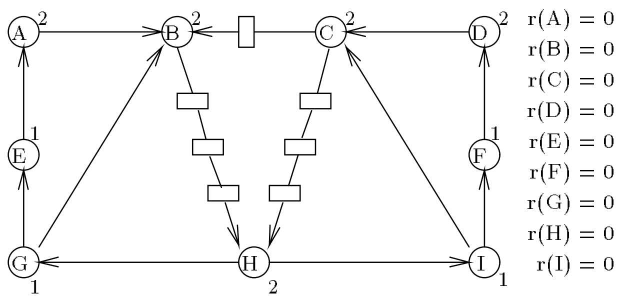


Figure 6.2: A data flow graph G.

Recall that $\mathcal{T}^{<s>}(u,v) = \max\{t(p) \mid u \overset{p}{\rightsquigarrow} v$ in G such that $d(p) = \mathcal{D}(u,v) + s\}$. Since the computation time of the path p and that of its corresponding path in G_f are the same, we can rewrite the definition of $\mathcal{T}^{<s>}(u,v)$ in terms of the paths in the unfolded graph as

$$\max\{t_f(p_f) \mid u^0 \overset{p_f}{\rightsquigarrow} v^{(\mathcal{D}(u,v)+s)\% f} \text{ in } G_f \text{ such that } d_f(p_f) = \left\lfloor \frac{\mathcal{D}(u,v)+s}{f} \right\rfloor\}.$$

If $\mathsf{D}_f(u^0, v^{(\mathcal{D}(u,v)+s)\% f}) = \left\lfloor \frac{\mathcal{D}(u,v)+s}{f} \right\rfloor$, from the definition of T_f we know that $\mathcal{T}^{<s>}(u,v) = \mathsf{T}_f(u^0, v^{(\mathcal{D}(u,v)+s)\% f})$. Thus, the theorem is derived. $\Box$

The function $\mathcal{D}$ can be easily computed by all-pair shortest path algorithms in time $O(|V|^3)$ or $O(|V|\,|E|+|V|^2 \log|V|)$. In time $O(|V|^2 f)$, we can construct $\mathcal{T}^{<s>}(u,v)$ for every u and v from T_f by using the above theorem. Therefore, the preprocessing time is dominated by the time to compute D_f and T_f, which is $O(|V|^3 f^3)$ or $O(f^2|V|\,|E| + f^2|V|^2 \log(f\,|V|))$.

6.3 An Algorithm for Sparse Data-Flow Graphs

This section presents another retiming algorithm to find a legal retiming r such that $\mathcal{C}(G_{r,f}) \le c$ for a given cycle period c and unfolding factor f. This algorithm runs in only $O(f\,|V|\,|E|)$ time and $O(f\,|V|)$ extra space, and is easy to implement. However, the minimum cycle period still needs to be found by binary search on all possible cycle periods as mentioned in the previous section.

Figure 6.3 shows the algorithm `Retiming Algorithm 2`, which computes certain longest-path information, and updates the retiming function, iteratively. Procedure

```
Retiming Algorithm 2
Input :   DFG G, cycle period c and factor f.
Output :  a retiming r such that C(G_{r,f}) ≤ c.
for every vertex u in V do    r(u) ⟵ 0;
iter ⟵ 1;    finished ⟵ FALSE;
while iter ≤ |V| and not finished do
begin
    Compute L(u^i) for every u^i in V_f
            by Procedure Cycle_Period(G_r,f);
    finished ⟵ TRUE; iter ⟵ iter + 1;
    for every u^i in V_f such that L(u^i) > c do
    begin
        r(u) ⟵ r(u) + 1;
        finished ⟵ FALSE;
    end ;
end ;
if finished then Output r;
else No retiming exists.
```

Figure 6.3: Retiming algorithm for sparse general-time DFGs

`Cycle_Period`(G, f) from Section 3.3 is used in every iteration to compute the longest-path function $L(u^i)$. Recall that $L(u^i)$ is the length of the longest zero-delay path from node u^i to any node in the unfolded graph G_f. The activation of Procedure `Cycle_Period` with parameters G_r and f computes the function $L(u^i)$ on the DFG $G_{r,f}$.

We will show that each iteration of Retiming Algorithm 2 simulates one phase of the Bellman-Ford algorithm on the ILP in Theorem 6.3. Thus, after a total of $|V|$ iterations, the retiming function achieving the cycle period c and unfolding factor f can be found.

Theorem 6.5 *If and only if there exists a retiming r such that $\mathcal{C}(G_{r,f}) \leq c$, after $|V|$ iterations the algorithm finds such a retiming (i.e. the boolean variable* finished *is* `TRUE`*).*

Thus, if finished is `FALSE`, there does not exist any retiming such that $\mathcal{C}(G_{r,f}) \leq c$.

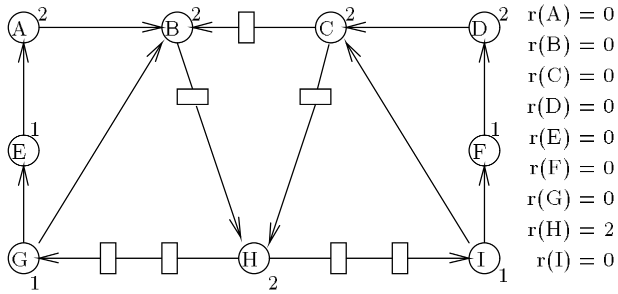


Figure 6.4: The retimed DFG after the first iteration of relaxation.

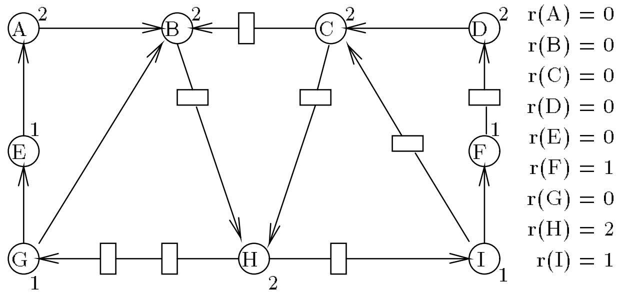


Figure 6.5: After the second iteration of relaxation.

Besides the space for G, the amount of extra space needed is only $O(f|V|)$ for the storage of $L(u^i)$. Therefore, in time $O(f\ |V|\ |E|)$ and extra space $O(f\ |V|)$, the above algorithm can decide whether there exists a retiming r such that $\mathcal{C}(G_{r,f}) \leq c$, and find a legal retiming if exists.

Consider the DFG in Figure 6.2, where $\mathcal{C}(G_f)$ equals 10 for $f = 3$. For an unfolding factor 3 and cycle period 8, the retiming functions r found after iteration 1, 2 and 3 are shown in Figure 6.4, Figure 6.5, and Figure 6.6 respectively. The retimed unfolded DFG in Figure 6.6 achieves the cycle period 8 for the unfolding factor 3, which is rate-optimal.

Proof of Theorem 6.5

We will show that each iteration of Retiming Algorithm 2 computes one phase of the Bellman-Ford algorithm, Retiming Algorithm 2, in Figure 4.4. Since the Bellman-Ford algorithm needs at most $|V|$ phases, Theorem 6.5 is proved if the above claim is true.

Consider one iteration of Retiming Algorithm 2 where $r(u)$ is increased by $\delta(u)$ for

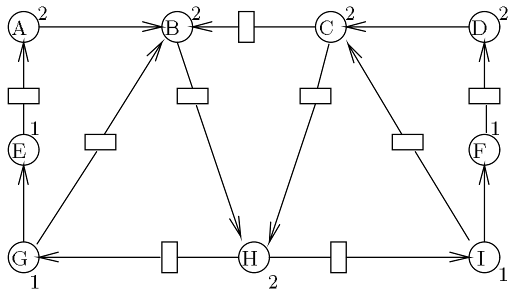


Figure 6.6: After the third iteration of relaxation.

every u in V. We look at the modification involving node u in one iteration. From the definition of $L(u^i)$ in Section 3.3 and Lemma 3.3.3, we have

$$L(u^i) = \max\{t(p) \mid p \text{ is a path emitting from } u \text{ such that } i + d_r(p) < f\}.$$

It is easy to observe that $L(u^0) \geq L(u^1) \geq \ldots \geq L(u^{f-1})$. The value $\delta(u)$ is the cardinality of the set $\{u^i \mid$ every node u^i with $L(u^i) > c\}$. And, the cardinality of the set $\{u^i \mid$ every node u^i with $L(u^i) \leq c\}$ is $f - \delta(u)$. Thus, we have

$$L(u^0) \geq \ldots \geq L(u^{\delta(u)-1}) > c \geq L(u^{\delta(u)}) \geq \ldots \geq L(u^{f-1}).$$

Since $c \geq L(u^{\delta(u)})$, for every path p emitting from u with $d_r(p) < f - \delta(u)$, we know $t(p) \leq c$. Since $L(u^{\delta(u)-1}) > c \geq L(u^{\delta(u)})$, there exists a path p emitting from u with $t(p) > c$ such that $\delta(u) - 1 + d_r(p) < f$ and $\delta(u) + d_r(p) \geq f$. Therefore, there exists a path p emitting from u such that $d_r(p) = f - \delta(u)$ and $t(p) > c$. Actually, $f - \delta(u)$ can be expressed as

$$\min\{d_r(p) \mid \text{ for every path } p \text{ emitting from } u, t(p) > c\}.$$

We first prove that the retiming function found after each iteration is always legal. Consider an edge $u \xrightarrow{e} v$ after an iteration of the algorithm. After an iteration, $r(u)$ is updated to be $r'(u) = r(u) + \delta(u)$ for every $u \in V$. We want to prove that the new delay count $d_{r'}(e) = d_r(e) + \delta(u) - \delta(v) \geq 0$. The function $\delta(v)$ can be written as: $\delta(v) = \max\{f - d_r(p) \mid \forall w, v \overset{p}{\rightsquigarrow} w, t(p) > c\}$. Consider a path $v \overset{p}{\rightsquigarrow} w$. Let p' be the path $u \xrightarrow{e} v \overset{p}{\rightsquigarrow} w$. we have $d_r(p') = d_r(e) + d_r(p)$ and $t(p') = t(u) + t(p)$. The value of $\delta(u)$ is lower bounded by the value obtained when only a subset of paths emitting from u is considered.

$$\begin{aligned}
\delta(u) &\geq \max\{f - d_r(p') \,|\, \forall w, \forall p' = u \xrightarrow{e} v \overset{p}{\rightsquigarrow} w, t(p') > c\} \\
&= \max\{(f - d_r(p)) - d_r(e) \,|\, \forall w, v \overset{p}{\rightsquigarrow} w, t(u) + t(p) > c\} \\
&= -d_r(e) + \max\{f - d_r(p) \,|\, \forall w, v \overset{p}{\rightsquigarrow} w, t(p) > c - t(u)\} \\
&\geq -d_r(e) + \max\{f - d_r(p) \,|\, \forall w, v \overset{p}{\rightsquigarrow} w, t(p) > c\} \\
&= \delta(v) - d_r(e).
\end{aligned}$$

Next, we rephrase the ILP form to reflect the modification on $r(u)$ in one phase of the Bellman-Ford algorithm, Retiming Algorithm 1. Let $\Delta(u)$ be the amount of increase on $r(u)$ after one phase, i.e. $new\_r(u) - r(u)$. We will prove that $\Delta(u)$ equals $\delta(u)$ if an iteration of Retiming Algorithm 2 and a phase of Retiming Algorithm 1 start from the same legal retiming r. Since r is legal, the first set of inequalities in Theorem 6.3 is satisfied. This phase needs to update the value of $r(u)$ to satisfy the related second set of inequalities:

$$r(v) - r(u) \leq D(u, v) + s - f \quad \forall v \in V, 0 \leq s < f \text{ such that } \mathcal{T}^{<s>}(u, v) > c.$$

For a particular node v, the subset of related inequalities can be rephrased as:

$$\begin{aligned}
&r(v) - r(u) \leq D(u, v) - f + S(u, v) \\
&\qquad \text{where } S(u, v) = \min\{s \,|\, \text{for every } 0 \leq s < f, \mathcal{T}^{<s>}(u, v) > c\}.
\end{aligned}$$

The value $new\_r(u)$ should satisfy

$$r(v) - new\_r(u) \leq D(u, v) - f + S(u, v) \quad \forall v \in V.$$

Therefore, after one phase, the value of $new\_r(u)$ becomes $\max\{r(v) - D(u, v) + f - S(u, v) \,|\, \forall v \in V\}$. Thus, the retiming of node u is increased by the amount of

$$\begin{aligned}
\Delta(u) &= new_r(u) - r(u) \\
&= \max_v\{f - S(u, v) - D_r(u, v)\} \\
&= f - \min_v\{S(u, v) + D_r(u, v)\}.
\end{aligned}$$

From the definition of $\mathcal{T}^{<s>}(u,v)$, $S(u,v)$ can be written as $\min\{s \mid \text{ for every } u \overset{p}{\leadsto} v, t(p) > c \text{ and } d(p) = D(u,v) + s\}$. We have

$$
\begin{aligned}
& S(u,v) + D_r(u,v) \\
&= \min\{s + D_r(u,v) \mid \text{ for every } u \overset{p}{\leadsto} v, t(p) > c \text{ and } d(p) = D(u,v) + s\} \\
&= \min\{s + D_r(u,v) \mid \text{ for every } u \overset{p}{\leadsto} v, t(p) > c \text{ and } d_r(p) = D_r(u,v) + s\} \\
&= \min\{d_r(p) \mid \text{ for every } u \overset{p}{\leadsto} v, t(p) > c\}.
\end{aligned}
$$

Thus, the amount of increase on $r(u)$ is

$$
\begin{aligned}
\Delta(u) &= f - \min_v\{\min\{d_r(p) \mid \text{ for every } u \overset{p}{\leadsto} v, t(p) > c\}\} \\
&= f - \min\{d_r(p) \mid \text{ for every } v, u \overset{p}{\leadsto} v, t(p) > c\} \\
&= \delta(u).
\end{aligned}
$$

The retiming found after the phase is $r+\delta$, which is a legal retiming, as proved earlier in this section. Since initially $r(u) = 0$ is legal, a legal retiming is obtained after each phase of the Bellman-Ford algorithm. Therefore, each iteration of Retiming Algorithm 2 simulates one phase of the Bellman-Ford algorithm. Theorem 6.5 is proved.

6.4 Concluding Remarks

In Chapter 3 we showed that unfolding before retiming or after retiming has no effect on the iteration period. Therefore, we do not need to unfold DFGs first to obtain results; instead we can do operations directly on the original DFG. For an unfolding factor f and a cycle period c, we presented two retiming algorithms with implicit unfolding: one runs in time $O(f|V|^2 + |V|^3)$ after the preprocessing step; the other runs in time $O(f\ |V|\ |E|)$ without preprocessing. For dense data-flow graphs, the first algorithm has a better time complexity. When $|E|$ is linear in $|V|$ and f is small, the second algorithm has a better time complexity. If only a few pairs of f and c are considered, the second algorithm obviously dominates the first algorithm since the time-consuming preprocessing step can be skipped.

In order to find the minimum cycle period for a given unfolding factor f, both algorithms need to perform binary search on all possible cycle periods, which are obtained from the function $\mathcal{T}^{<s>}$ in the preprocessing. The time complexity of finding the minimum cycle period thus is dominated by the time complexity of the preprocessing algorithm, which is $O(|V|^3 f^3)$ or $O(f^2|V|\,|E| + f^2|V|^2 \log(f\,|V|))$ depending on which all-pairs shortest path algorithm is used. We believe that the complexity of the preprocessing algorithm can be further reduced because much of redundant information is generated in the computation of $\mathcal{T}^{<s>}$. After preprocessing, the first retiming algorithm finds the minimum cycle period in time $O((f|V|^2 + |V|^3)\log(f|V|))$; the second retiming algorithm in time $O(f\,|V|\,|E|\log(f|V|))$.

In Chapters 4 and 5, we are able to obtain a simple formula for the existence of pipelined schedules with a given cycle period and a given unfolding factor for both unit-time and general-time DFGs. From this formula, we can obtain the minimum cycle period for a given unfolding factor very easily. As explained in Section 2.6, a repeating schedule obtained by retiming on general-time DFGs has the property that the execution period of any node does not span across the boundary of the repeating schedule. To generate schedules by retiming and unfolding on general-time DFGs (without node splitting), we are not able to obtain such a nice formula. We need to do a binary search to find the minimum cycle period for a given unfolding factor. Due to the above difficulty, we are not able to find rate-optimal unfolding factors without exhaustive search. An interesting open problem is whether there are more efficient methods to find rate-optimal unfolding factors for retiming and unfolding general-time DFGs (without node splitting).

Chapter 7

Scheduling Uniform Nested Loops

7.1 Introduction

In the previous chapters, we focused on the scheduling of the innermost loop body or single-level loop. In order to further improve the performance, we continue to apply the methodology developed to nested loop scheduling. In this chapter, a consistent framework using retiming and unfolding on *multi-dimensional data-flow graphs* is presented for nested loop-scheduling problems.

Most loop-pipelining algorithms for single loops can be applied to nested loops by pipelining each level of the nested loop, from inner to outer, while the inner loops are treated as single statements. In order to further explore the parallelism of nested loops, methodologies are designed especially for nested loops. Two approaches have been used. One treats the loop body as an atomic cell, and only the coarse-grain parallelism among cells is explored. Data dependency graphs are drawn on points in the *iteration space*, which is composed of discrete points representing iterations. This iteration-space approach modifies the order in which iterations are executed without changing the structure of an iteration. Loop skewing [80] and loop quantization [1] are two techniques of this class.

Another approach tries to explore instruction-level parallelism by looking at the structure of each cell. The technique of "Doacross loop" [17] belongs to this class. "Doacross loop" computes a time interval (initiation interval) for every level of the nested loop to decide the starting time of the next iteration in that level. The overlap of iterations contributes to the improvement of the execution rate.

Analogous to our model of loop pipelining for single-level loops, we study the steady state of pipelining nested loops. Our methodology regroups instructions in iterations in order to produce a new iteration structure with higher parallelism. Since the parallelism is explicitly displayed in the new loop body, latter stages of optimization, e.g. resource allocation, can be simplified *without* looking across loop boundaries of the new loop structure. The method can also be combined with the iteration-space algorithm to explore more parallelism.

This chapter defines *multi-dimensional data-flow graphs* and *multi-dimensional retiming and unfolding* in order to generalize the results of retiming and unfolding data-flow graphs to nested loop scheduling.

In the next section, we present the definitions and properties of retiming and unfolding for a multi-dimensional data-flow graph. In section 7.3, we show the effect of retiming functions on the nested loop body, the generation of a static schedule, and the relationship between the retiming function and the size of the prologue. The general framework is presented in Section 7.4 for some forms of nested loops. Integer linear inequality systems are derived to optimize the iteration period of nested loops. Finally, some concluding remarks are made.

7.2 Definitions and Properties

An n-level nested loop is represented as an *n-dimensional DFG*. For simplicity, we specifically study *two-dimensional DFGs (2DFG)*, where the dimensions are referred to as the x-dimension and the y-dimension. All results can be easily generalized to *multi-dimensional DFGs (MDFG)* in a straightforward way.

In this section, we define 2DFG and the concepts of retiming and unfolding on 2DFG. The innermost body of a two-level loop can be represented by a data-flow graph with edges labeled by two-dimensional vectors. Consider the following program and the corresponding 2DFG G in Figure 7.1. Each operation corresponds to a node in the DFG, and each dependency is represented by an edge labeled by a constant

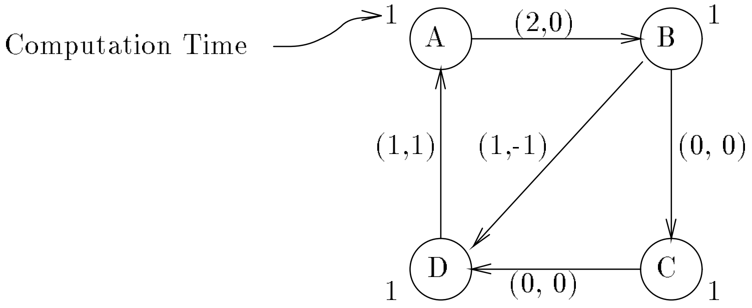


Figure 7.1: A two-dimensional data-flow graph G

delay vector. For ease of illustration, we assume that all operations take one unit time to compute for this example. However, our formulations are not restricted to unit-time operations.

```
for i = 0 to b_x do
   for j = 0 to b_y do
       A[i,j] = D[i-1,j-1] + 17
       B[i,j] = A[i-2, j] + 2
       C[i, j] = B[i,j] + 10
       D[i,j] = C[i,j] + B[i-1,j+1]
   end
end
```

A formal definition of two-dimensional data-flow graphs is given below. This form of data-flow graphs represents only data dependencies between nodes at a constant distance in the iteration space. We call this special class of loops *uniform* loops. Under the assumption that the iteration space is large, we focus on improving the execution rate of the innermost body of uniform nested loops. After the innermost body is restructured by retiming and unfolding, the statements between nested loops can be rearranged according to the retiming and unfolding transformations applied to the innermost body.

Definition 7.1 *A two-dimensional data-flow graph (2DFG) $G = (V, E, d, t)$ is a node-weighted and edge-weighted directed graph, where V is the set of nodes, $E \subseteq$*

$V \times V$ is the set of edges, d is a function from E to $\mathbf{Z} \times \mathbf{Z}$, and t is a function from V to the positive integers.

Set V represents the set of computation nodes and the value $t(v)$ of a node v where $v \in V$ is the computation time of v. Set E represents precedence relations among computation nodes and the vector $d(e)$ of an edge e is the *delay vector* (dependency vector) of edge e.

Let $\mathbf{Z}$ be the set of integers, and $\mathbf{Z} \times \mathbf{Z}$ (alternatively, $\mathbf{Z}^2$) the set of two-dimensional integer vectors. A two-dimensional vector P is written as $(P.x, P.y)$. Let $\mathbf{N}$ be the set of nonnegative integers. The addition (resp. subtraction, multiplication, division) of two two-dimensional vectors is defined as a component-wise addition (resp. subtraction, multiplication, division); for example, $P + Q \cdot R = (P.x + Q.x \times R.x, P.y + Q.y \times R.y)$. The comparison of two two-dimensional vectors is also component-wise and the result is the *conjunction* of results from all components. For example, $P \leq Q$ iff $P.x \leq Q.x$ and $P.y \leq Q.y$.

We define *one iteration* to be the execution of each node in V exactly once. We use a two-dimensional vector $i = (i.x, i.y)$, starting from $(0, 0)$, to identify iterations for a 2DFG. Inter-iteration data dependencies are represented by vector-weighted edges. For any vector j in $\mathbf{N}^2$, an edge e from u to v with delay vector $d(e)$ means that the computation of node v at iteration j depends on the computation of node u at iteration $j - d(e)$. An edge with no delay (with delay vector $(0, 0)$) represents the data dependency within the same iteration.

We can expand both dimensions to show the dependencies of each iteration as Figure 7.2 shows. We call this graph the *infinite graph* for G. The properties of such a pair of a multi-dimensional data-flow graph and an infinite graph has been studied by several researchers under the terms of a *static graph* and a *dynamic graph*, respectively. Orlin [60], Iwano and Steiglitz [35, 34], and Cohen and Megiddo [14] studied some properties of one-dimensional, two-dimensional, and n-dimensional static and dynamic graphs, respectively.

For the corresponding infinite graph to be a meaningful data-flow graph, the summation of delay vectors in any cycle should be nonzero, i.e. the infinite graph is acyclic. The problem of zero-weight cycle detection for a static graph was first

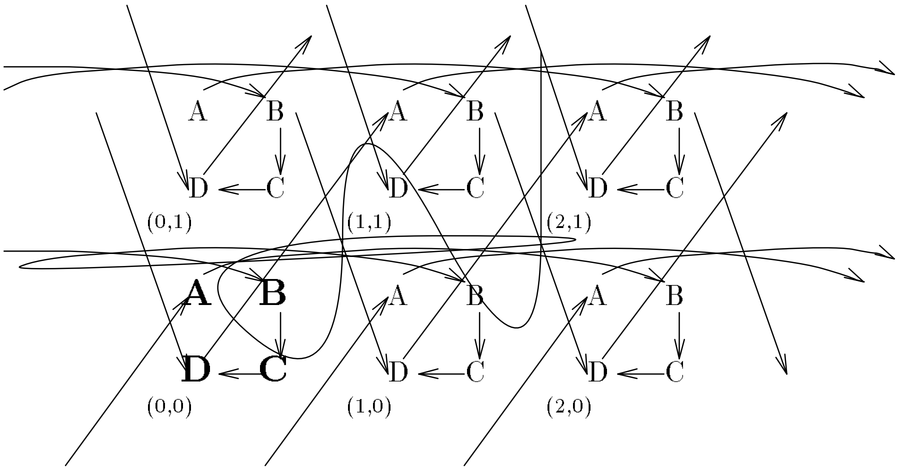


Figure 7.2: The infinite graph for G

introduced by Iwano and Steiglitz [35]. They proposed polynomial-time algorithms for certain special cases. A polynomial-time algorithm was presented in [42] for general static graphs of any fixed dimension. Finally, a strongly polynomial-time algorithm is derived in [13].

For the sake of convenience, we use the following notation. The notation $u \xrightarrow{e} v$ means that e is an edge from node u to node v. The notation $u \overset{p}{\rightsquigarrow} v$ means that p is a path from node u to v. The delay vector of a path $p = v_0 \xrightarrow{e_0} v_1 \xrightarrow{e_1} v_2 \ldots \xrightarrow{e_{k-1}} v_k$ is $d(p) = \sum_{i=0}^{k-1} d(e_i)$. The total computation time of a path p is $t(p) = \sum_{i=0}^{k} t(v_i)$.

The results in this chapter optimize a class of 2DFGs with this property: The total delay vector of any cycle in the 2DFG is nonnegative in every dimension, i.e. $d(l).x \geq 0$ and $d(l).y \geq 0$ for every cycle l in G. Any 2DFG with this property can be retimed into a 2DFG where the delay vectors are nonnegative. We call such a 2DFG a *normalized 2DFG*. In this chapter, we focus on normalized 2DFGs.

The period during which all computation nodes in an iteration are executed subject to the dependencies is called the *cycle period*. The cycle period $\mathcal{C}(G)$ of a 2DFG G is the maximum total computation time of zero-delay paths, that is, $\mathcal{C}(G) = \max\{t(p) \mid \text{p is a path in } G \text{ with } d(p) = (0,0)\}$. The cycle period can also be characterized as follows. The cycle period $\mathcal{C}(G) \leq c$ if and only if for every path p in G if $d(p) = (0,0)$ then $t(p) \leq c$.

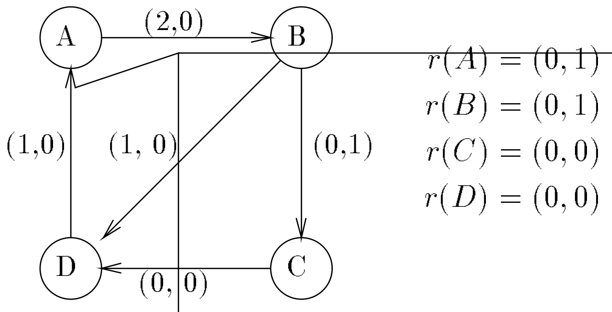


Figure 7.3: The retimed data-flow graph G_r with $r(A) = r(B) = (0, 1)$

7.2.1 Retiming Multi-Dimensional Data-Flow Graphs

The technique of retiming redistributes nodes in G to iterations such that each iteration still has one execution of each node in G while the dependencies represented by G are preserved. We can define a two-dimensional retiming function to represent the reindexing of nodes in the two-dimensional iteration space, as we did in one-dimensional space.

A *two-dimensional retiming* r of a 2DFG G is a function from V to $\mathbf{Z}^2$. The retiming vector $r(v)$ of a node v represents the offset between the location before retiming and that after retiming. The execution of node v in iteration i is moved to be the execution of node v in iteration $i - r(v)$. The delay vectors of edges are changed accordingly in order to reflect the new locations of executions of nodes; that is, there are $r(v)$ delays pushed from all the incoming edges of v to all the outgoing edges of v.

A retiming r is *legal* if all the delay vectors $d_r(e)$ in the retimed graph G_r are nonnegative. For example, the graph in Figure 7.3 is a retimed graph. Two retimings r and r' are *equivalent* if $d_r(e) = d_{r'}(e)$ for every edge e in G.

Lemma 7.2.1 *Let $G = (V, E, d, t)$ be a 2DFG and r a two-dimensional retiming. The 2DFG $G_r = (V, E, d_r, t)$ is the 2DFG obtained by retiming G by r.*

(a) *For any edge $u \xrightarrow{e} v$, we have $d_r(e) = d(e) + r(u) - r(v)$.*

(b) *For any path $u \overset{p}{\rightsquigarrow} v$, we have $d_r(p) = d(p) + r(u) - r(v)$.*

(c) *For a cycle l in the DFG, we have $d_r(l) = d(l)$.*

We identify the execution of v in iteration i, where $i \in \mathbf{N}^2$, as the copy (or the execution) i of v, denoted v^i. The execution of node v in iteration i is moved to be the execution of node v in iteration $i - r(v)$. The retimed infinite graph is shown in Figure 7.7, where we use boldface for those nodes belonging to cell (0,0) in the original infinite graph. The infinite graph for G_r is structurally equivalent to the infinite graph for G. Thus, we know that 2-dimensional retiming preserves the data dependencies.

7.2.2 Unfolding Multi-Dimensional Data-Flow Graphs

Let $f = (f.x, f.y)$ be a positive two-dimensional vector. An *unfolded 2DFG* with *two-dimensional unfolding factor* f, denoted G_f, consists of $f.x \times f.y$ copies of node set V and represents the same precedence relations in G. We say that the unfolded 2DFG $G_f = (V_f, E_f, d_f, t_f)$ is a 2DFG obtained by unfolding G f times. The original 2DFG G can be considered as a 2DFG G_f where $f = (1,1)$. The node sets are indexed by two-dimensional vectors i, where $0 \leq i.x < f.x$ and $0 \leq i.y < f.y$. The node set i is denoted by V^i. For a node u in G, let u^i be the node u in V^i, called the i-th copy of u, and the computation time $t_f(u^i) = t(u)$. The unfolded 2DFG gives a two-dimensional global view of the data dependencies with a manageable graph size, since f is chosen by the user.

One *computation cycle* of G_f consists of all computation nodes in V_f. The period during which all computations in a computation cycle are executed is called the *cycle period*. The cycle period $\mathcal{C}(G_f)$ of G_f equals $\max\{t_f(p_f) \mid p_f$ is a path in G_f with $d_f(p_f) = (0,0)\}$. During a cycle period of G_f, $f.x \times f.y$ iterations of G are executed, which correspond to a rectangle with $f.x$ iterations as width and $f.y$ iterations as height. Thus, the *iteration period* of G_f is equal to $\mathcal{C}(G_f)/(f.x \times f.y)$, in other words, the average computation time for each iteration in G. For the original DFG G, the iteration period is equal to $\mathcal{C}(G)$.

We use the subscript f to represent the correspondences between G and G_f. For a node v (resp. edge e) in G, v_f (resp. e_f) represents any copy of v (resp. e) in G_f. For a path p_f in G_f, there is a unique path p in G corresponding to p_f. In order to describe the unfolded 2DFG, we define the component-wise *modular* operation % and

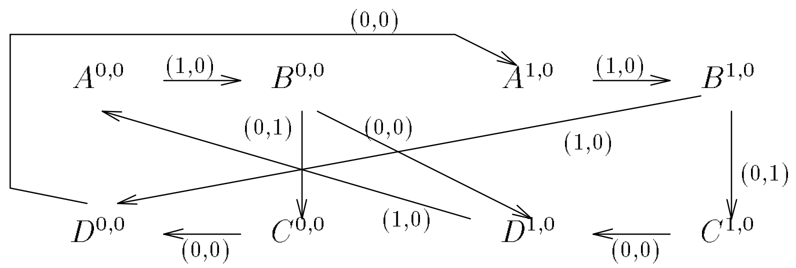


Figure 7.4: The unfolded data-flow graph $G_{r,f}$ with $f = (2, 1)$

$\equiv_f$ on Z^2. Let P, Q, f be two-dimensional vectors. The notation $P \equiv_f Q$ means that there exists $n \in Z^2$ such that $P - Q = n \times f$. The result of $P \,\%\, f$ equals Q iff $(0, 0) \leq Q < f$ and there exists $n \in Z^2$ such that $P = n \times f + Q$.

All the dependencies within a computation cycle are intra-cycle dependencies and represented as edges with no delay (with delay vector $(0, 0)$) in G_f. Let k and n be two-dimensional vectors. The dependency which goes from the k-th computation cycle to the $(k+n)$-th computation cycle is represented by an edge with delay n. Since $f.x \times f.y$ iterations compose one computation cycle, every $(f.x, 0)$ delays in G are represented by one delay $(1, 0)$ in G_f; every $(0, f.y)$ delays in G are represented by one delay $(0, 1)$ in G_f. Let i and j be two-dimensional vectors such that $(0, 0) \leq i, j < f$. Therefore, an edge e_f in E_f from u^i to v^j with delay vector $d_f(e_f)$ means that for every computation cycle k in the iteration space there is a data dependency from node u of the i-th iteration in computation cycle k to the node v of the j-th iteration in computation cycle $k + d_f(e_f)$. The data dependency is represented by an edge e from u to v with delay vector $d_f(e_f) \times f + j - i$ in the original DFG G. Therefore, we have the relation $d(e) = d_f(e_f) \times f + j - i$ for every $u \xrightarrow{e} v$ and $u^i \xrightarrow{e_f} v^j$. This suggests a way of constructing E_f and d_f from G as follows. Every edge $u \xrightarrow{e} v$ has $(f.x, f.y)$ copies in G_f, which are $u^i \xrightarrow{e_f} v^{(d(e)+i)\% f}$ for every $0 \leq i < f$ and $d_f(e_f) = (d(e) + i - ((d(e) + i) \,\%\, f))/f$. For example, the graph in Figure 7.4 is the unfolded graph with factor (2, 1) from the graph in Figure 7.3.

The value of $d_f(e_f)$ can be written in terms of the floor and ceiling operations on $d(e)$.

$$d_f(e_f).x = \begin{cases} \left\lfloor \frac{d(e).x}{f.x} \right\rfloor & \text{if } j.x \geq i.x, \\ \\ \left\lceil \frac{d(e).x}{f.x} \right\rceil & \text{otherwise.} \end{cases}$$

By substituting y for x, $d_f(e_f).y$ is defined similarly. After a simple calculation, we find $\sum_{0 \leq i.x < f.x} d_f(e^i).x = d(e).x$ and $\sum_{0 \leq i.y < f.y} d_f(e^i).y = d(e).y$, where e^i is the i-th copy of e. These properties can be easily extended to paths by substituting p for e and each path in G has exactly f copies in G_f. We use p^i to represent the i-th copy of p , which starts at node u^i if p starts at node u.

In the following lemma, the value of $\mathcal{C}(G_f)$ is obtained from the original DFG G and we show that the cycle period in the unfolded graph G_f is the maximum total computation time among all paths where delay vectors are less than f in the original graph G.

Lemma 7.2.2 *Let f be a two-dimensional unfolding factor and G_f be the 2DFG unfolded from G. $\mathcal{C}(G_f)$ is equal to* $\max\{t(p) \mid d(p) < f$ *for every path p in G* $\}$. *In other words, $\mathcal{C}(G_f) \leq c$ if and only if for every path p in G if $d(p) < f$ then $t(p) \leq c$.*

Proof: Let node u^i in G_f be the ith copy of the node u in G. Thus, we know

$$\begin{aligned} \mathcal{C}(G_f) &= \max\{t_f(p_f) \mid d_f(p_f) = (0,0) \quad \text{for every path } p_f \text{ in } G_f\} \\ &= \max\{t_f(p^i) \mid u^i \overset{p^i}{\rightsquigarrow} v^j \text{ and } d_f(p^i) = (0,0) \;\; \text{for } (0,0) \leq i,\ j < f \text{ and } u^i, v^j \text{ in } V_f\}. \end{aligned}$$

Since we know $d(p) = d_f(p^i) \times f + j - i$ and $d_f(p^i) = (0,0)$, $d(p)$ is equal to $j - i$. From the facts $-f < j - i < f$ and $d(p) \geq (0,0)$, we have $(0,0) \leq d(p) < f$. Thus, this lemma is proved. $\Box$

7.3 Loop Schedulings and Retimings

In this section, we explain the effect of retiming on the schedule of loops, and how to obtain the prologue, static schedule, and epilogue from a retiming function. Consider

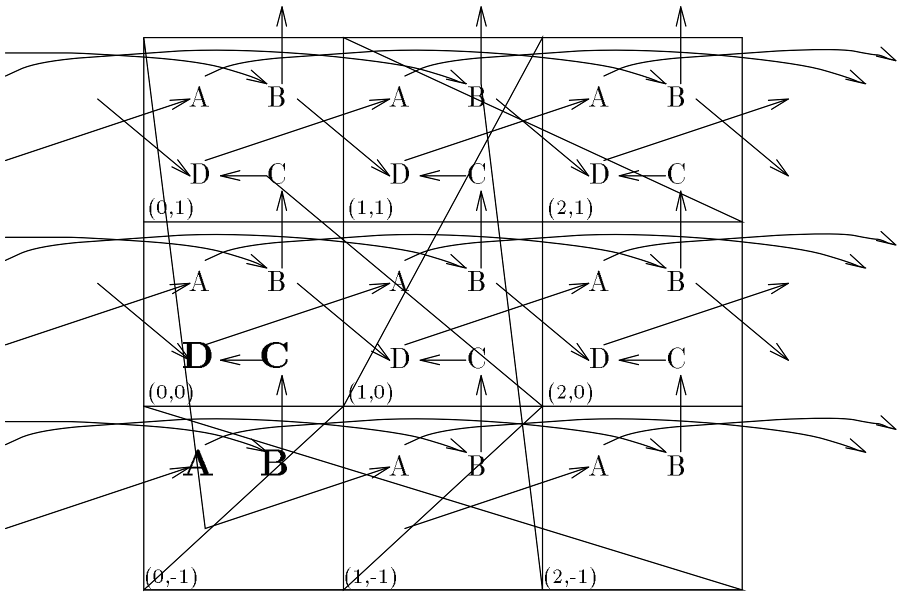
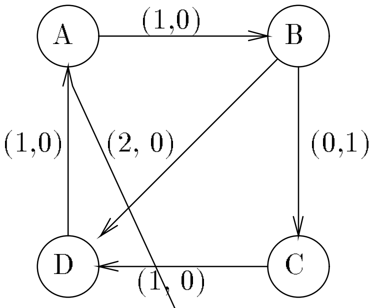


Figure 7.5: The infinite graph for G_r

the 2DFG G in Figure 7.1. We first normalize it into the DFG G_r in Figure 7.3 by the retiming function $r(A) = r(B) = (0, 1)$ and $r(C) = r(D) = (0, 0)$. This normalization can be done in $O(|V||E|)$ time by using retiming. In the following sequence of retimings, we will use boldface for those nodes belong to cell (0,0) in the infinite graph (figure 7.2) for the original DFG. We can see how these nodes are moved around due to the effect of retiming.

Consider the two-dimensional infinite graph of G in Figure 7.2. The effect of the retiming r on G is that every copy of node B and node A are shifted down by one cell, and the retimed DFG is shown in Figure 7.3. The infinite graph in Figure 7.5 shows this effect of retiming. A prologue in the y-dimension appears as the bottom-most row, consisting of node B and node A only. The infinite graph for G_r is structurally equivalent to the infinite graph for G. Thus, we know that 2-dimensional retiming preserves the data dependencies.

Note that the length of the longest zero-delay path of G_r is 2, which is the cycle period of G_r, reduced from 3 in G. Furthermore, we can retime DFG G_r by another retiming function $r'(B) = r'(C) = (1, 0)$ to minimize the cycle period to 1. As a result, the node B and node C in every cell of Figure 7.5 are shifted left by one cell.



$$r''(A) = (0,1)$$
$$r''(B) = (1,1)$$
$$r''(C) = (1,0)$$
$$r''(D) = (0,0)$$

Figure 7.6: The retimed data-flow graph $G_{r''}$

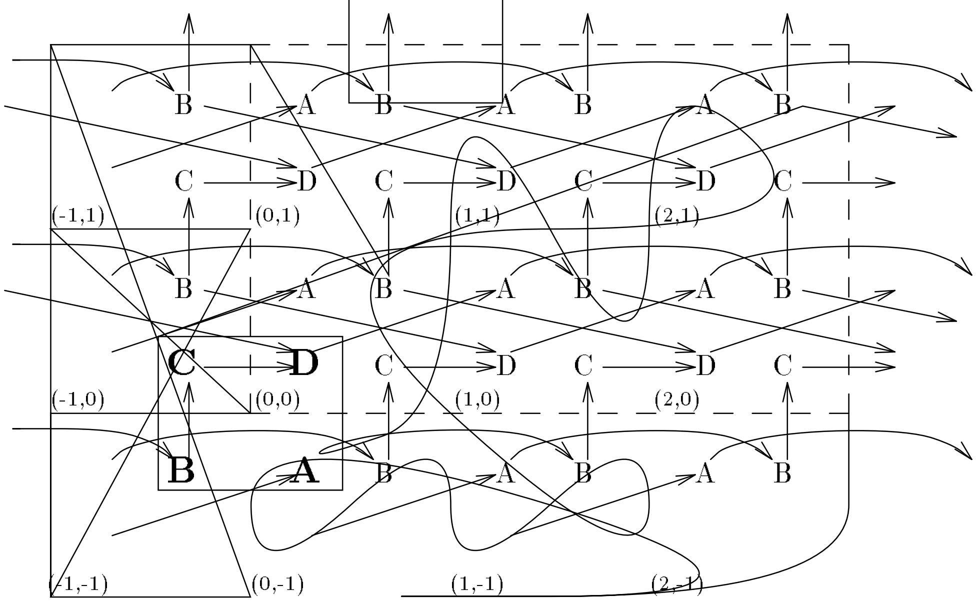


Figure 7.7: The infinite graph for $G_{r''}$

The resultant DFG is in Figure 7.6. In the corresponding infinite graph, there are prologues in both the x- and y-dimensions, as shown in Figure 7.7. Let r'' be the composition of r and r'. The DFG in Figure 7.6 is $G_{r''}$.

The technique of retiming regroups operations in a loop into new iterations, where each operation is executed once in each new iteration. As shown in the above examples, the node v in new iteration i comes from the old iteration $i + r(v)$. The four nodes in the new iteration can be executed at the same time. Thus, the cycle period is improved from 3 to 1 time unit. Figure 7.8 shows the retimed loop program. The total computation time of the retimed loop is $b_x \times b_y + 2b_x + 2b_y + 4$ instead of

$3 \times (b_x + 1) \times (b_y + 1)$. Therefore, the new retimed loop will run roughly three times faster than the original loop.

The acyclic subgraph consisting of edges of delay vector (0,0), i.e. the edges within one cell in the infinite graph, gives the static schedule for the new innermost loop. The statements between nested loops can be merged with prologue or epilogue. Since the innermost loop body is the most time-consuming part, we focus on this part first. Then, we can use the same technique to improve the time performance of statements in outer loops.

A retiming function r is called *normalized* if $r(v)$ is nonnegative and $\min_v r(v) = (0,0)$. For any given retiming function r, we can always normalize it into a normalized function r, where $r(v) = \mathsf{r}(v) - \min_v \mathsf{r}(v)$ for every v in V. Because the retiming function of each node is changed by the same amount, $\min_v \mathsf{r}(v)$, we know r is equivalent to r. Therefore, we only need to consider normalized retiming functions.

After retiming by function r, the node v originally in cell i will be moved into cell $i - r(v)$. In general, those nodes moved into cells with negative indices are part of the prologue. The prologue cell p contains the set of nodes $\{v \mid v \in V$ such that $p + r(v) \geq (0,0)\}$. For example, in Figure 7.7 those nodes in cell $(-1, \bullet)$ and cell $(\bullet, -1)$ compose the prologue.

Moreover, the retiming function provides the structure of the prologue and epilogue. Let $\mathcal{P}(r).x$ be $\max_v r(v).x$, and $\mathcal{P}(r).y$ be $\max_v r(v).y$. In the prologue and epilogue, there are $\mathcal{P}(r).x$ iterations in the x-dimension and $\mathcal{P}(r).y$ iterations in the y-dimension, as shown in Figure 7.9. We can control the approximate size of prologue and epilogue by adding constraints for $\mathcal{P}(r)$ during the optimization process. For a given prologue volume p, the addition of the constraints $r(v) \geq (0,0)$ and $r(v) \leq \mathsf{p}$ for every v in V to our integer linear programming formulations derived in Section 7.4 will find a retiming such that the volume of the prologue is no more than p.

Let n be the vector of upper bounds for every level of the nested loop,[1] i.e. $n = (b_x, b_y)$ for the example. The volume of iterations executed at full speed by the static schedule is $n - \mathcal{P}(r)$, where n is the total number of iterations to be executed

[1] The bounds can be variables unknown at compile time, or even infinite as in signal-processing applications.

```
/* Prologue in y-dimension */
B[0,0] = A[-2,0] + 2
for j = 0 to b_y - 1 do
  B[0,j+1] = A[-2,j+1] + 2
  C[0,j] = B[0,j] + 10
end
C[0,b_y] = B[0,b_y] + 10
for i = 0 to b_x - 1 do
  /* Prologue in x-dimension */
  A[i,0] = D[i-1,-1] + 17
  B[i+1,0] = A[i-1,0] + 2
  /* The retimed inner loop */
  for j = 0 to b_y - 1 do
    A[i,j+1] = D[i-1,j] + 17
    B[i+1,j+1] = A[i-1,j+1] + 2
    C[i+1,j] = B[i+1,j] + 10
    D[i,j] = C[i,j]+B[i-1,j+1]
  end
  /* Epilogue in x-dimension */
  C[i+1,b_y] = B[i+1,b_y] + 10
  D[i,b_y] = C[i,b_y]+B[i-1,b_y + 1]
end
/* Epilogue in y-dimension */
A[b_x,0] = D[b_x - 1,-1] + 17
for j = 0 to b_y - 1 do
  A[b_x,j+1] = D[b_x - 1,j] + 17
  D[b_x,j] = C[b_x,j]+B[b_x - 1,j+1]
end
D[b_x,b_y] = C[b_x,b_y]+B[b_x - 1,b_y + 1]
```

Figure 7.8: The program for the retimed DFG $G_{r''}$

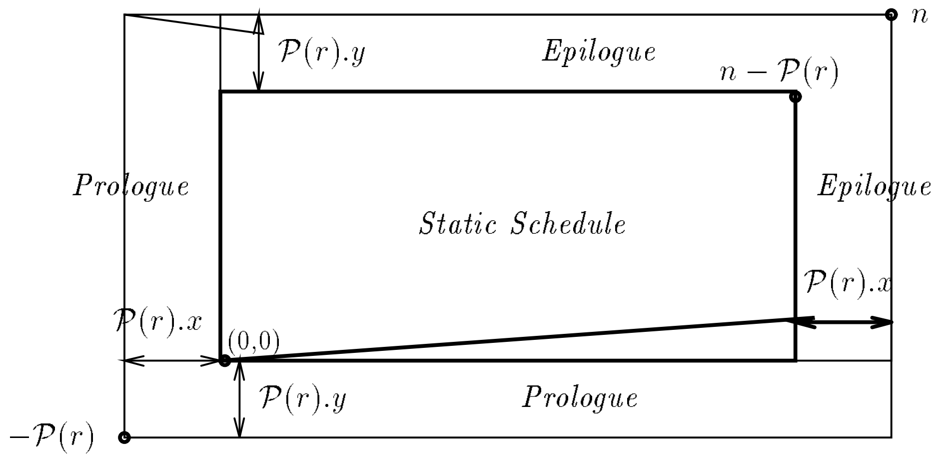


Figure 7.9: The prologue and the epilogue

(assumed to be large). Therefore, when a retiming is given, we not only have the entry status and the length of the static schedule, but also the volume of the prologue and the epilogue.

7.4 Multi-Dimensional Retiming and Unfolding

Just as the static scheduling for one-level loops corresponds to the retiming and unfolding on one-dimensional DFG's, the static scheduling for the data dependencies in the innermost body of a nested loop can be modeled using a multi-dimensional DFG. Since the performance of the innermost body usually dominates the overall performance, we optimize the iteration period of the innermost body by the techniques of retiming and unfolding. After the multi-dimensional retiming of the innermost body is decided, we can retime the operations between adjacent levels of the nested loop accordingly. The problem of nested loop scheduling turns out to be much more difficult than the problem of one-level loop scheduling.

In this section we derive a general framework and properties for retiming and unfolding on multi-dimensional DFG's. We derive an integer linear programming (ILP) formulation to test the feasibility of a cycle period for retiming an MDFG. Another ILP formulation is also derived to test the feasibility for retiming and unfolding a MDFG. The minimum cycle period can obtained by using these ILP forms. For brevity, we specifically study two-dimensional DFG, since all the results can be generalized to higher dimensions in a straightforward way. Our results lead to a consistent solution for the loop scheduling of nested loops.

Let $G_{r,f}$ denote the graph obtained by unfolding G_r f times. In order to avoid manipulating the unfolded graphs, we would like to find a retiming r on the original graph G such that $\mathcal{C}(G_{r,f}) \leq c$ for a given cycle period c and a given unfolding factor f. In one-dimensional case, we have proved that the order of retiming and unfolding is immaterial in Chapter 3. We have not proved the analogous result regarding the order of retiming and unfolding for multi-dimensional retiming and unfolding.

In order to determine the cycle period of a DFG, we define functions $\mathcal{D}$ and $\mathcal{T}$ to identify a set of critical paths such that the cycle period can be determined by examining these critical paths. The *minimum* operation over a set of vectors is defined component-wise by taking the minimum in all dimensions.

Definition 7.2 *Let u and v be nodes in 2DFG G and s a two-dimensional vector where $(0,0) \leq s < f$. The function $\mathcal{D}(u,v)$ is a function from $V \times V$ to $\mathbf{N}^2$, and the function $\mathcal{T}^{<s>}(u,v)$ is a function from $V \times V$ to $\mathbf{N}$.*

(a) *Define $\mathcal{D}(u,v) = min\{d(p) \mid for\ every\ u \overset{p}{\rightsquigarrow} v\ in\ G\}$.*

(b) *Define $\mathcal{T}^{<s>}(u,v) = \max\{t(p) \mid u \overset{p}{\rightsquigarrow} v \text{ and } d(p) = \mathcal{D}(u,v) + s\}$.*

(When there is no path $u \overset{p}{\rightsquigarrow} v$, $\mathcal{D}(u,v)$ is undefined and $\mathcal{T}^{<s>}(u,v)$ is zero. When there is no path $u \overset{p}{\rightsquigarrow} v$ such that $d(p) = \mathcal{D}(u,v) + s$, we define $\mathcal{T}^{<s>}(u,v) = 0$.)

An algorithm modified from Floyd-Warshall shortest-paths algorithm can compute $\mathcal{D}$ and $\mathcal{T}^{<s>}$. The paths $u \overset{p}{\rightsquigarrow} v$ such that $d(p) = \mathcal{D}(u,v) + s$ and $t(p) = \mathcal{T}^{<s>}(u,v)$ are called *critical paths* from u to v for every $(0,0) \leq s < f$.

Theorem 7.1 *Let $G = (V, E, t, d)$ be a 2DFG, c a positive integer and f a positive two-dimensional vector. Let $G_f = (V_f, E_f, t_f, d_f)$ be the DFG obtained by unfolding G with factor f. Then, the cycle period $\mathcal{C}(G_f) \leq c$ if and only if for all nodes u and v in V and for all s where $(0,0) \leq s < f$ if $\mathcal{D}(u,v) + s < f$ then $\mathcal{T}^{<s>}(u,v) \leq c$.*

Proof: To prove the only-if part by contradiction, we assume $\mathcal{C}(G_f) \leq c$, and that there exists $u, v \in V$ and $(0,0) \leq s < f$ such that $\mathcal{D}(u,v) + s < f$ and $\mathcal{T}^{<s>}(u,v) > c$.

Let p be a critical path $u \overset{p}{\rightsquigarrow} v$ with $d(p) = \mathcal{D}(u,v) + s$ and $t(p) = \mathcal{T}^{<s>}(u,v)$. Thus, there exists a path p in G such that $d(p) < f$ and $t(p) > c$. From Lemma 7.2.2, we know $\mathcal{C}(G_f) > c$. This is a contradiction.

Now we prove the if-part. Assume that for all u, $v \in V$ and for all $(0,0) \leq s < f$ if $\mathcal{D}(u,v) + s < f$ then $\mathcal{T}^{<s>}(u,v) \leq c$. Under this assumption we claim [2] that for every path p in G either $d(p) \not< f$ or $t(p) \leq c$, which implies $\mathcal{C}(G_f) \leq c$. The if part follows from the claim directly. Let $u \overset{p}{\rightsquigarrow} v$ be a path in G. If p is a critical path, we know $d(p) = \mathcal{D}(u,v) + s$ and $t(p) = \mathcal{T}^{<s>}(u,v)$ for some s where $(0,0) \leq s < f$; thus, the claim is true. Otherwise, if p is not a critical path, we want to show that $d(p) \not< f$ or $t(p) \leq c$. There are two cases. First, in case that $d(p) \not< \mathcal{D}(u,v) + f$, we know $d(p) \not< f$ since $\mathcal{D}(u,v) \geq 0$. Otherwise, in case that $d(p) = \mathcal{D}(u,v) + s$ where $(0,0) \leq s < f$, we have $t(p) \leq \mathcal{T}^{<s>}(u,v)$. From the assumption, we know that either $d(p) \not< f$ or $t(p) \leq c$. Therefore, the claim is proved.

□

The next theorem gives the necessary and sufficient conditions for a retiming r with $\mathcal{C}(G_{r,f}) \leq c$ in terms of $\mathcal{D}$ and $\mathcal{T}^{<s>}$.

Theorem 7.2 *Let $G = (V, E, t, d)$ be a DFG, c a cycle period and f an unfolding factor. The following two statements are equivalent:*

(a) *r is a retiming on G such that $\mathcal{C}(G_{r,f}) \leq c$.*

(b) *for all nodes u and v in V and for all s where $(0,0) \leq s < f$ if $\mathcal{D}(u,v) + r(u) - r(v) + s < f$ then $\mathcal{T}^{<s>}(u,v) \leq c$.*

Proof: Let $\mathcal{D}_r(u,v)$ and $\mathcal{T}_r^{<s>}(u,v)$ be defined on G_r in the natural way. First, we prove that the effect of retiming on the functions $\mathcal{D}(u,v)$ and $\mathcal{T}^{<s>}(u,v)$ is similar to that on the functions d and t, i.e.

(a) $\mathcal{D}_r(u,v) = \mathcal{D}(u,v) + r(u) - r(v)$.

(b) $\mathcal{T}_r^{<s>}(u,v) = \mathcal{T}^{<s>}(u,v)$.

Then, this theorem is easily derived from Theorem 7.1 on the retimed graph G_r.

For part (a), $\mathcal{D}_r(u,v)$ is defined as $\min\{d_r(p) \mid u \overset{p}{\rightsquigarrow} v\}$. Since $d_r(p) = d(p) +$

---

[2] $d(p) \not< f$ iff $d(p).x \geq f.x$ or $d(p).y \geq f.y$. Note that $d(p) \geq f$ implies $d(p) \not< f$, but the reverse is not true.

$r(u) - r(v)$, we have

$$\begin{aligned} \mathcal{D}_r(u,v) &= r(u) - r(v) + \min\{d(p) \mid u \overset{p}{\rightsquigarrow} v\} \\ &= r(u) - r(v) + \mathcal{D}(u,v). \end{aligned}$$

Part (a) implies Part (b):

$$\begin{aligned} \mathcal{T}_r^{<s>}(u,v) &= \max\{t(p) \mid u \overset{p}{\rightsquigarrow} v \text{ and } d_r(p) = \mathcal{D}_r(u,v) + s\} \\ &= \max\{t(p) \mid u \overset{p}{\rightsquigarrow} v \text{ and } d(p) + r(u) - r(v) = \mathcal{D}_r(u,v) + s\} \\ &= \max\{t(p) \mid u \overset{p}{\rightsquigarrow} v \text{ and } d(p) = \mathcal{D}(u,v) + s\} \\ &= \mathcal{T}^{<s>}(u,v). \end{aligned}$$

□

For one-dimensional DFG's, the above theorem gives an ILP formulation identical to the one we obtained in Theorem 6.3 for ordinary DFGs.

$$\begin{aligned} r(u) - r(v) &\leq d(e) \quad \text{for every } u \xrightarrow{e} v \text{ of } G, \text{ and} \\ r(u) - r(v) &\leq \mathcal{D}(u,v) - f + s \quad \text{for all } u, v \in V, \;\; 0 \leq s < f \text{ such that } \mathcal{T}^{<s>}(u,v) > c. \end{aligned}$$

In time $O(f|V|^2)$, an ILP formulation can be generated for a given c by scanning the values of $\mathcal{T}^{<s>}(u,v)$. Since the inequalities which have the same left-hand side can be covered by the one which has the smallest value on the right-hand side, an ILP formulation with at most $|V|^2$ inequalities is obtained. Thus, a legal retiming can be found in time $O(|V|^3)$ by the Bellman-Ford algorithm for the single-source shortest path problem [75]. A slightly complicated ILP formulation is derived for two-dimensional DFG's in the following theorem and can be easily extended to multi-dimensional DFG's.

Theorem 7.3 *Let $G = (V, E, t, d)$ be a 2DFG, r a two-dimensional retiming on G, c a positive integer and f a positive two-dimensional vector. Then r is a legal retiming* *on G such that $\mathcal{C}(G_{r,f}) \leq c$ if and only if*

$$
\begin{aligned}
(ILP)\quad & r(v).x - r(u).x \leq d(e).x \text{ and } r(v).y - r(u).y \leq d(e).y \\
& \qquad\qquad \text{for every } u \xrightarrow{e} v, \text{ and} \\
& r(v).x - r(u).x \leq \mathcal{D}(u,v).x - (f.x - s.x) \times Fx(u,v,s) \\
& r(v).y - r(u).y \leq \mathcal{D}(u,v).y - (f.y - s.y) \times Fy(u,v,s) \\
& Fx(u,v,s) + Fy(u,v,s) = 1, \quad Fx(u,v,s) \geq 0, \quad Fy(u,v,s) \geq 0 \\
& \qquad\qquad \text{for every } u,v \in V, \quad (0,0) \leq s < f \text{ such that } \mathcal{T}^{<s>}(u,v) > c, \\
& \text{where } r(v).x, r(v).y, Fx(u,v,s), \text{ and } Fy(u,v,s) \text{ are integral.}
\end{aligned}
$$

Proof: We first show that r is a legal retiming such that $\mathcal{C}(G_{r,f}) \leq c$ if and only if r satisfies the following mathematical program (IMP).

$$
\begin{aligned}
(IMP)\quad & r(v).x - r(u).x \leq d(e).x \quad \text{and} \quad r(v).y - r(u).y \leq d(e).y \\
& \qquad\qquad \text{for every } u \xrightarrow{e} v, \text{ and} \\
& r(v).x - r(u).x \leq \mathcal{D}(u,v).x - (f.x - s.x) \text{ or} \\
& \qquad\qquad r(v).y - r(u).y \leq \mathcal{D}(u,v).y - (f.y - s.y) \\
& \qquad\qquad \text{for every } u,v \in V, \quad (0,0) \leq s < f \text{ such that } \mathcal{T}^{<s>}(u,v) > c.
\end{aligned}
$$

The function r is a legal retiming if and only if $d_r(e) \geq (0,0)$ for every e in G. Since $d_r(e) = d(e) + r(u) - r(v)$, r is a legal retiming if and only if $r(v) - r(u) \leq d(e)$ for every $u \xrightarrow{e} v$. The second set of inequalities comes from Theorem 7.2, which can be rephrased as $\mathcal{C}(G_r) \leq c$ iff for every u and v in G and every s where $(0,0) \leq s < f$ if $\mathcal{T}^{<s>}(u,v) > c$ then $\mathcal{D}(u,v) + r(u) - r(v) + s \not< f$. However, (IMP) is not an ILP formulation.

We claim that r satisfies (IMP) if and only if r satisfies (ILP). In the (ILP), $Fx(u,v,s)$ equals either 0 or 1. If $Fx(u,v,s) = 1$, the inequality $r(v).x - r(u).x \leq \mathcal{D}(u,v).x - (f.x - s.x) \times Fx(u,v,s)$ becomes $r(v).x - r(u).x \leq \mathcal{D}(u,v).x - (f.x - s.x)$. Thus, it is obvious that if r satisfies (ILP) then r satisfies (IMP). Assume that r satisfies (IMP). For every $u,v \in V$, $(0,0) \leq s < f$ such that $\mathcal{T}^{<s>}(u,v) > c$, if $r(v).x - r(u).x \leq \mathcal{D}(u,v).x - (f.x - s.x)$ is satisfied, we set $Fx(u,v,s) = 1$ and $Fy(u,v,s) = 0$; otherwise, we set $Fx(u,v,s) = 0$ and $Fy(u,v,s) = 1$. If $Fx(u,v,s) = 0$, the inequality $r(v).x - r(u).x \leq \mathcal{D}(u,v).x - (f.x - s.x) \times Fx(u,v,s)$ becomes $r(v).x -$

$r(u).x \leq \mathcal{D}(u,v).x$, which is satisfied by any legal retiming, since this inequality is a linear combination of the first set of inequalities for legal retiming. Therefore, (ILP) and (IMP) are equivalent. $\square$

Actually, the above ILP form is a mixed 0-1 linear programming formulation. As long as Fx and Fy values are integal, we can always set r's to be their ceilings of the fractional solutions to satisfy the same inequalities. When the technique of unfolding is not used, the cycle period of G_r can be characterized as follows.

Corollary 7.4.1 *Let $G = (V, E, t, d)$ be a 2DFG, r a two-dimensional retiming on G, c a positive integer. Let $\mathsf{D}(u,v)$ and $\mathsf{T}(u,v)$ be $\mathcal{D}(u,v)$ and $\mathcal{T}^{<s>}(u,v)$ respectively where $s = (0,0)$. Then r is a legal retiming on G such that $\mathcal{C}(G_r) \leq c$ if and only if*

$$\begin{array}{ll} (ILP') & r(v).x - r(u).x \leq d(e).x \quad \textit{and} \quad r(v).y - r(u).y \leq d(e).y \\ & \qquad \textit{for every } u \xrightarrow{e} v, \textit{ and} \\ & r(v).x - r(u).x + r(v).y - r(u).y \leq \mathsf{D}(u,v).x + \mathsf{D}(u,v).y - 1 \\ & \qquad \textit{for every } u, v \in V \textit{ such that } \mathsf{T}(u,v) > c. \end{array}$$

Proof: The graph G_r is equivalent to the graph $G_{r,f}$ where $f = (1,1)$. The range of the superscript s contains only one vector $(0,0)$, so we can reduce the functions $\mathcal{D}$ and $\mathcal{T}^{<s>}$ to the functions D and T. The (IMP) formulation in the proof of Theorem 7.3 becomes

$$\begin{array}{ll} (IMP') & r(v).x - r(u).x \leq d(e).x \quad \text{and} \quad r(v).y - r(u).y \leq d(e).y \\ & \qquad \text{for every } u \xrightarrow{e} v, \text{ and} \\ & r(v).x - r(u).x \leq \mathsf{D}(u,v).x - 1 \quad \text{or} \quad r(v).y - r(u).y \leq \mathsf{D}(u,v).y - 1 \\ & \qquad \text{for every } u, v \in V \text{ such that } \mathsf{T}(u,v) > c. \end{array}$$

Since $r(v).x - r(u).x \leq d(p).x$ for any path from u to v, we have $r(v).x - r(u).x \leq \mathsf{D}(u,v).x$. Similarly, $r(v).y - r(u).y \leq \mathsf{D}(u,v).y$. Therefore, (IMP') is equivalent to (ILP'). $\square$

From Theorem 7.3, we can find the minimum cycle period for a given unfolding factor f by binary search of all the values appeared as $\mathcal{T}^{<s>}(u,v)$ for some u and v, which are all the candidates for cycle period. Since there are $|V|^2 f.x f.y$ values of $\mathcal{T}^{<s>}(u,v)$, at most $O(\log |V|)$ ILP formulations are needed to find the minimum cycle period, under the usual assumption that $f.x$ and $f.y$ are linear in $|V|$.

Just as the ILP formulation for one-level loops is the dual problem of a network flow problem, there is an interesting flow interpretation of the dual of the above ILP formulation. Three sets of flow variables, say U, V and W, are present. There are only two flow-conservation equations for the combined flow of U and W and the combined flow V and W. Although it is slightly more complicated than usually multi-commodity flow problem, such an interpretation might shed some light on finding polynomial or approximation algorithms.

7.5 Concluding Remarks

We use retiming and unfolding to solve the problem of static loop scheduling of uniform nested loops. We combine both retiming and unfolding, and give a consistent framework to multi-dimensional DFG's for solving loop schedulings of a wide class of nested loops. We believe that this framework gives more insight into loop schedulings, and should be studied more for different requirements.

We derive an ILP formulation to test the feasibility of the cycle period for retiming an MDFG. Another ILP formulation is also derived to test the feasibility for retiming and unfolding a MDFG. Note that all of these ILP formulations are simple to formulate. An open question is how to design provably efficient algorithms for solving these ILP formulations.

In Chapter 3, we proved the independence of the order for retiming and unfolding for one-dimensional DFGs. It is still open whether there is an analogous relationship between retiming and unfolding for multi-dimensional DFGs.

Retiming on multi-dimensional DFGs itself has interesting applications to optimizing the cycle period for systolic arrays. Usually, after a projection vector is decided, the multi-dimensional dependency graph is projected to a one-dimensional graph where nodes represent physical processing elements. The (one-dimensional) retiming technique is then applied to make the new graph systolic, i.e. every edge has unit delay [52]. Each node in the multi-dimensional dependency graph can be a compound node performing a certain function. The usual technique of retiming does not improve the execution time of each compound node. If the compound node of a dependency graph is represented by a MDFG, we can apply multi-dimensional retiming to restructure the compound cell, before it is mapped to a physical array, such that the cycle period is reduced.

Chapter 8

Rotation Scheduling

8.1 Introduction

For real-time or high-performance computing, a synthesis system needs to have the ability to optimize the execution rate of a design. Since loops are usually the most time-critical parts of an application, the parallelism embedded in the repetitive pattern of a loop needs to be explored. However, the problem of scheduling loops under resource constraints is NP-complete even if there is no inter-iteration dependencies, i.e. the data-flow graph contains no delays, and all operations have the same computation time [29]. This chapter proposes a generic technique for the scheduling of loops when resource constraints are present. A heuristic scheduling algorithm is proposed based on the technique.

In this chapter, we design a novel and flexible technique, called *rotation scheduling*, for scheduling cyclic DFGs using loop pipelining. The rotation technique repeatedly transforms a schedule to a more compact schedule. We provide a theoretical basis for the operations based on retiming. We propose two heuristics to perform rotation scheduling, and give experimental results showing that they have very good performance. Chapter 9 will investigate theoretical properties of rotations, and show the generality and sufficiency of a sequence of *optimal rotations*. The results from the two chapters about rotations are combined in Section 9.5.

A practical algorithm for high-level synthesis must be able to handle *chained* operations, *multi-cycle* operations and *pipelined functional units*. In data-path design, the length of a *clock cycle* dictates the length of one control step. If the total computation time of two operations where one must precede the other does not exceed the length of one clock cycle, the two operations can be *chained* to fit into the same clock cycle. Multi-cycle operations refer to operations that cannot be finished within one clock cycle. Thus, the corresponding resource must be available for several control steps. A pipelined functional unit can accept a new computation before the previous computation finishes. Usually, a new computation can come in at each control step. Rotation scheduling introduced in this chapter is able to handle all these types of operations.

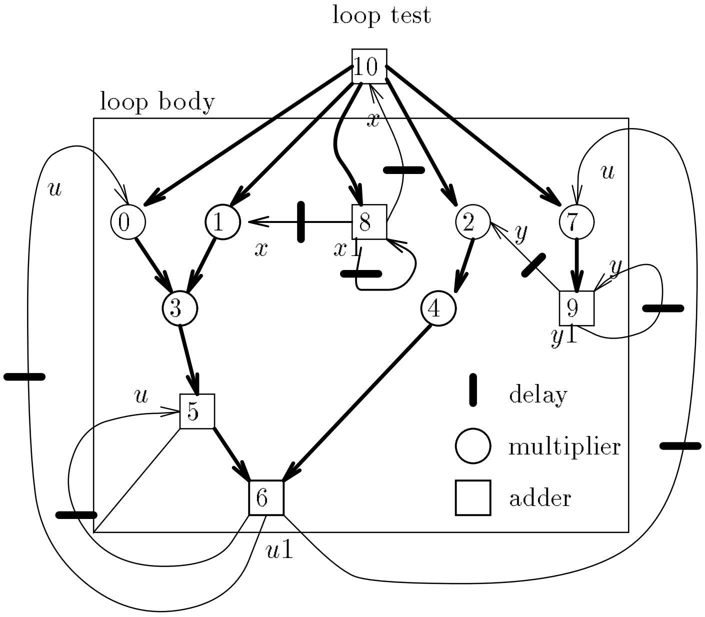
In the previous chapters, we have applied retiming and unfolding to schedule cyclic DFGs without resource constraints. This chapter concerns resource constraints while scheduling. Scheduling cyclic DFGs with resource constraints is more difficult than scheduling *acyclic* DFGs with resource constraints or scheduling cyclic DFGs without resource constraints. The loops are usually pipelined in order to increase the execution rate, where the execution periods of several iterations are overlapped. Loop winding [30] was proposed to pipeline loops with acyclic DFGs, where theoretically the performance can be made arbitrarily good. However, when pipelining cyclic DFGs, the cycles confine the depth and freedom of loop pipelining. In contrast to acyclic DFGs, cycles in a DFG provide bounds on the improvement we can achieve by pipelining. In this chapter, we propose a generic technique to optimize a cyclic DFG under resource constraints.

Previous work on loop pipelining for loops with cyclic dependencies appears in several systems [50, 31, 68, 77]. Percolation-based scheduling [58, 68] unfolds (unwinds) the loop incrementally to find a repeating pattern in the schedule of the unfolded loop without resource constraints, and then schedules it with resources. The size of the pipeline schedule cannot be predicted until several incremental unfoldings are applied. The MARS system [77] first finds all cycles in the DFG and computes the iteration bound. A set of cycle sections is derived to cover all cycles. Resource conflicts are

To solve $y'' + 3xy' + 3y = 0$.

```
while (x < a) do
    x1 =  x + dx;
    u1 =  u − (3 ∗ x ∗ u ∗ dx)
          −(3 ∗ y ∗ dx);
    y1 =  y + u ∗ dx;
    x = x1;   u = u1;   y = y1;
end
```

(a) The behavioral description



(b) The cyclic data flow graph

Figure 8.1: The differential equation solver

resolved after the cycle sections are scheduled individually. The nodes not belonging to any cycle are scheduled at the last step. Retiming is performed implicitly in MARS. In Cathedral II [31], a DFG is retimed to meet an estimated schedule length without resource constraints, and then another graph is constructed from the DFG and retiming function to reschedule the loop entirely under resource constraints. Iteratively, the estimated schedule length is decreased one by one from an upper bound obtained from list scheduling. Recently, the functional pipelining algorithm by Lee *et. al* [50] uses a priority function, called *variability*, for rescheduling operations in an initial as-soon-as-possible pipeline schedule to resolve resource conflicts.

Our loop pipelining algorithm improves a resource-constrained legal schedule and performs implicit retiming incrementally by the *rotation* technique. An existent schedule is partially rescheduled by rotation to obtain a shorter and valid schedule under resource constraints. The result of rotation retimes the DFG implicitly to naturally produce a pipeline schedule. The state of a sequence of rotations is recorded by a simple retiming (node-labeling) function. In fact, rotation is a generic technique that can be used to design a class of heuristic algorithms for loop pipelining. The two

CS	Mult	Adder
1	–	10
2	1	8
3	0	–
4	3	–
5	2	5
6	4	–
7	7	6
8	–	9

(a) an optimal DAG schedule

CS	Mult	Adder
2	1	8
3	0	**10**
4	3	–
5	2	5
6	4	–
7	7	6
8	–	9

(b) first rotation

CS	Mult	Adder
3	0	10
4	3	**8**
5	2	5
6	4	–
7	7	6
8	**1**	9

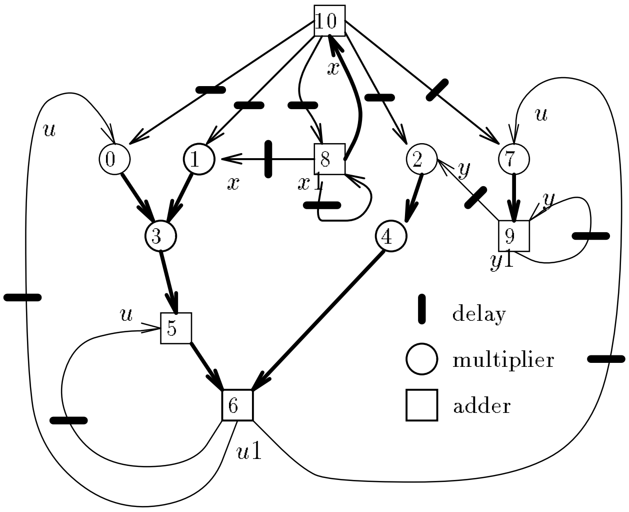
(c) second rotation

Figure 8.2: Two down-rotations of size 1 for unit-time operations

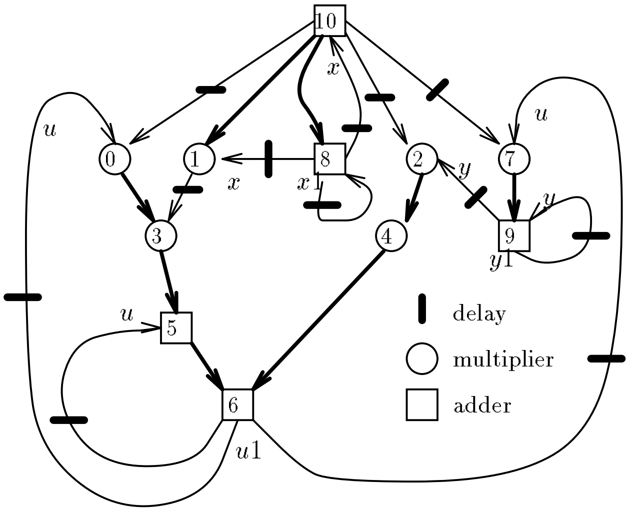
simple heuristics proposed give very good experimental results.

We use the differential equation solver in Figure 8.1 as an example throughout the chapter. The derivation of the DFG from the behavioral description is described in Subsection 2.2.1. Here, we assume that additions and multiplications take one time unit, and a *control step (CS)*, also called *clock cycle*, consists of one time unit. For a resource of 1 multiplier and 1 adder, an optimal schedule for the DAG part of the DFG is shown in Figure 8.2-(a). Figure 8.2-(b) shows a more compact schedule, in which Node 10 has been rotated down and then pushed up to its new position. This new schedule is a schedule of the retimed graph G_r, as shown in Figure 8.3-(a), where a delay is pushed through Node 10 from the original DFG. Intuitively, Node 10, which was a source in the original DFG, is *rotated down* into a sink. The Nodes 1 and 8 in Figure 8.2-(b) are rotated down and then rescheduled to their new positions in Figure 8.2-(c), which is an optimal schedule for this example. The retimed DFG is shown in Figure 8.3-(b), where a delay is pushed through Nodes 1 and 8 respectively from the DFG in (a). We provide theoretical foundations to support such movement of nodes, and methods to check the retimed DFG for rescheduling.

Rotation uses a DAG scheduling algorithm, like *list scheduling* [18], as a subroutine, and can be easily incorporated into an existing DAG scheduler. This method can handle chained operations, multi-cycle operations, and pipelined functional units. Since only a part of the DFG is rescheduled in each rotation, much computation time



(a) $r(10) = 1$ after the first rotation



(b) $r(10) = r(8) = r(1) = 1$ after the second rotation

Figure 8.3: The corresponding retimed graphs after rotations

can be saved by only rescheduling the rotated part. The model of rotations by retiming simplifies the checking of precedence constraints without reconstructing a DFG after each rotation. There is no need to construct the retimed graphs in our procedure. The retimed graphs are drawn in this chapter to help the presentation.

The next section defines *down* and *up* rotations. A basic rotation algorithm is presented in Section 8.3 on DFGs with single-cycle and chained operations. An efficient algorithm is presented to find a pipeline with a shallow depth from a schedule. In Section 8.4, the basic algorithm is then refined to handle multi-cycle operations and pipelined functional units. The concept of a *wrapped* schedule is also introduced there. A couple of heuristics are proposed in Section 8.5. These heuristics give very good experimental results, which are presented in Section 8.6. We believe that the rotation technique lays a good foundation for the resource-constrained scheduling of loops.

8.2 Definitions

Let G be a DFG as defined in Chapter 2. A *static* schedule of a loop is repeatedly executed for the loop. An edge without delays represents an intra-iteration precedence relation. A static schedule must obey the precedence relations defined by the subgraph consisting of edges without delays in a DFG; thus, this subgraph must be a directed acyclic graph (DAG). The path with the maximum total computation time of the DAG defines the *cycle period* of the DFG. The cycle period of the DFG is the length of the static schedule for the DAG without resource constraints.

The technique of retiming moves around delays in the following way: a delay is drawn from *each* of the incoming edges of v, and then a delay is pushed to *each* of the outgoing edges of v and vice versa. A retiming r of a DFG G is a function from V to the integers [53]. The value $r(v)$ is the number of delays pushed through node v from the incoming edges to the outgoing edges. Let $G_r = (V, E, d_r, t)$ be the DFG retimed by a retiming r from G, where $d_r(e) = d(e) + r(u) - r(v)$ for every edge e. A retiming r is *legal* if the value $d_r(e)$ is nonnegative for every edge e in E.

Leiserson and Saxe designed an algorithm to find the iteration period of an DFG G and an $O(|V||E| \log |V|)$-time algorithm to find a retiming such that the iteration period of G is the minimum [53]. Without loss of generality, we consider only *normalized* retiming functions. A retiming function r is *normalized* if $\min_v r(v) = 0$. For any retiming function r', we can normalize it into r where $r(v) = r'(v) - \min_v r(v)$ for every v in V.

A set X of nodes in V is represented by a 0-1 valued function X from V to $\{0, 1\}$. A node v belongs to set X if and only if $X(v) = 1$. This set representation is also used as a retiming function in our rotation operation.

The operation of *down-rotation* is defined as follows.

Definition 8.1 *Let $G = (V, E, d, t)$ be a DFG and X a subset of V. The down-rotation of set X pushes one delay from each of the incoming edges of X to each of the outgoing edges of X. The DFG G is transformed into DFG G_X after set X is rotated down.*

When a node is rotated down, a delay is pushed from all of its incoming edges to all of its outgoing edges. Consider a simple down-rotation of Node 10 in Figure 8.1-(b). This node, a source of the original DAG, becomes a sink node in the new DAG, shown in Figure 8.3-(a). Thus, intuitively it is *rotated down.* The operation of *up-rotation* on set X transforms G into G_{-X} by pushing one delay in the reverse direction. In this chapter, we will only use down-rotations. Similar properties and algorithms can be derived for up-rotations.

We say that a set X is *down-rotatable* if X is a legal retiming for G. Not any subset in V is rotatable. It is not hard to show the following property.

Property 8.1 *Let $G = (V, E, d, t)$ be a DFG and X a subset of V. A set X is down-rotatable if and only if every path from $V - X$ to X contains at least one delay.*

For example, in Figure 8.1-(b), the sets $\{10, 8, 1\}$ and $\{10\}$ are down-rotatable sets, but $\{8, 1\}$, $\{1\}$, and $\{8\}$ are not. Figure 8.3-(a) shows the retimed DFG after $\{10, 8, 1\}$ is rotated down. In this chapter, a restricted form of down rotations is used for rotations under resource constraints. General properties of up/down rotations are studied in Chapter 9.

We define the composite of two retimings as $r_1 \circ r_2(v) = r_1(v) + r_2(v)$. The composite of a sequence of down-rotations is the composite of the retimings of the down-rotation sets.

The advantage of associating retiming functions with rotations is that the precedence constraints, captured by d_r, of a retimed graph can be easily examined from the original DFG and the retiming r. Therefore, after each rotation, the scheduler does not need to construct a new retimed graph in order to find a new schedule. Much computation time can be spared.

The loop, represented by a DFG, is executed *in pipeline* if the execution periods of several iterations are overlapped. A static schedule describes a loop pipeline at its stable state. The length of a static schedule corresponds to the minimum cycle period of the loop pipeline, also called the *minimum initiation interval.*

If the nodes in a static schedule are from p different iterations, there are p pipeline stages. We call such a p the *depth* of a loop pipeline. Consider the retimed DFG in Figure 8.3-(b). There are two stages in the pipeline: The set of nodes with $r(v) = 1$

is in the first stage, i.e. $\{10, 8, 1\}$; the set of nodes with $r(v) = 0$ is in the second stage, i.e. the rest of nodes. We have the following property.

Property 8.2 *Let G be a DFG and r be a retiming function. The* depth *of a loop pipeline represented by a retiming function r is*

$$1 + \max_v r(v) - \min_v r(v).$$

The depth of a normalized retiming function is $1 + \max_v r(v)$.

An algorithm is presented in the next section to reduce the depth of a given loop pipeline.

8.3 The Basic Algorithms

In this section, we design a technique to compact a given schedule of a DFG under resource constraints by the technique of rotations. The basic rotation algorithm works for control steps with chained operations. We refine the algorithm for more general models (multi-cycle operations, and pipelined functional units) in the next section. After a sequence of rotations is performed, the depth of a loop pipeline might be too long. An efficient algorithm is presented in Subsection 8.3.2 to reduce the depth of a pipelined schedule.

Let $[l, r]$ be the set of integers $\{i \,|\, l \leq i \leq r\}$. A schedule s is a mapping from V to control steps such that the resource constraints are satisfied. The computation node v starts its execution at control step $s(v)$. The *DAG schedule* of a DFG G is *legal* if for every edge $(u, v) \in E$, $s(u) + t(u) \leq s(v)$ if $d(u, v) = 0$. A *static schedule* s is *legal* if there exists a retiming r such that schedule s is a legal DAG schedule for the retimed DFG G_r. The following property gives a characterization of *legal static schedule.*

Property 8.3 *Let G be a DFG, and s be a schedule of the DFG. Schedule s is a legal static schedule of G if and only if there exists a legal retiming r such that $s(u) + t(u) \leq s(v)$ if $d_r(u, v) = 0$.*

We say that the retiming r satisfying this property *realizes* the schedule s. A schedule may be realized by several retimings. In Subsection 8.3.2, we present an algorithm to find a retiming with a short pipeline depth for a given schedule.

8.3.1 The Basic Rotation Algorithm

For any legal DAG schedule of a DFG, we use the technique of rotation to compact a schedule to obtain a legal static schedule. For example, consider the DFG in Figure 8.1-(b). By list scheduling, which uses the number of descendants to prioritize the list, the DFG has a schedule of length 8 (see Figure 8.2-(a)), which is an optimal DAG schedule. We can rotate down the node $X_1 = \{10\}$ in the first control step. Then, we try to push the node 10 up to its earliest possible control step according to the precedence constraints of the DAG of G_{R_1}, where $R_1 = X_1$, as shown in Figure 8.3-(a). This is actually achieved by rescheduling the set of nodes rotated down, $\{10\}$, by list scheduling. We obtain a schedule of length 7 as shown in Figure 8.2-(b). After another rotation of one control step, we reschedule the nodes $X_2 = \{1, 8\}$ according to the DAG of G_{R_2}, where $R_2 = R_1 \circ X_2$, as shown in Figure 8.3-(b). The optimal schedule in Figure 8.2-(c) is then obtained after two rotations of one control step. The retimings obtained from a sequence of rotations, e.g. R_2, are called *rotation functions*.

Figure 8.4 shows the effect of rotation in a global view after the above two rotations. Part (b) shows the situation where rotations are performed without rescheduling. The prologue and epilogue are introduced, and the static schedule is a schedule for the DAG part of the retimed DFG G_r with $r(1) = r(8) = r(10) = 1$, as shown in Figure 8.3-(b). Figure 8.4-(c) shows the global view of the schedule in Figure 8.2-(c), where nodes 1,8 and 10 are rescheduled according to r. Intuitively, when a node is rotated, each copy of the node is pushed up by one iteration to a location in the previous iteration, and the first copy of the node is pushed into the prologue.

In general, we can find any down-rotatable set, rotate it down, and reschedule it at the end of the original schedule according to the DAG of the retimed DFG. Assume that s is a schedule of length k with range $[1, k]$. Let S_i be the set of nodes scheduled

Mul	Add
-	10
1	8
0	-
3	-
2	5
4	-
7	6
-	9
-	**10**
1	**8**
0	**-**
3	**-**
2	**5**
4	**-**
7	**6**
-	**9**
-	10
1	8
0	-
3	-
2	5
4	-
7	6
-	9
-	10
1	8
0	-
3	-
2	5
4	-
7	6
-	9

(a) Initial schedule

-	10	Prologue
1	8	
0	-	Static Schedule
3	-	
2	5	
4	-	
7	6	
-	9	
-	**10**	
1	**8**	
0	**-**	
3	**-**	
2	**5**	
4	**-**	
7	**6**	
-	**9**	
-	10	
1	8	
0	-	
3	-	
2	5	
4	-	
7	6	
-	9	
-	10	
1	8	
0	-	Epilogue
3	-	
2	5	
4	-	
7	6	
-	9	

(b) Rotate Down

Prologue	-	10
	1	8
Static Schedule	0	**10**
	3	**8**
	2	5
	4	-
	7	6
	1	9
	0	10
	3	8
	2	**5**
	4	**-**
	7	**6**
	1	**9**
	0	10
	3	8
	2	5
	4	-
	7	6
	1	9
Epilogue	0	-
	3	-
	2	5
	4	-
	7	6
	-	9

(c) Reschedule

Figure 8.4: The entire loop schedules after rotations

in the first i control steps. From Property 8.1, we know that S_i is a down-rotatable set for every i in $[1, k]$. Only the down-rotatable sets S_i are considered. The number of control steps rotated down in one down-rotation is called the *size* of a down-rotation. We have a new valid schedule s' of the same length k with range $[i+1, i+k]$, i.e.

$$s'(v) = \begin{cases} s(v) + k & \text{if } 0 < s(v) <= i, \\ s(v) & \text{otherwise.} \end{cases}$$

Notice that s' is a valid schedule for DFG G_{S_i}. Thus, after any down-rotation, there always exists a DAG schedule for the retimed DFG which is at least as short as the original one. The nodes in the last i control steps of s' can be rescheduled together with the partial schedule s' in range $[i+1, k]$ to get a shorter schedule. Experiments in Section 8.6 show that we are able to converge DAG schedules to the optimal lengths in almost all cases.

In the algorithm description, we use PartialSchedule $(G,\ s,\ X)$ to denote a procedure which returns a DAG schedule of G without changing the existent schedule s for nodes in $V - X$. In examples throughout this chapter, we use list scheduling for the procedure PartialSchedule(G, s, X) with the number of descendants as the weight function. The following procedure DownRotate performs one rotation of size i on a schedule s of DFG G, and returns a new schedule of length L.

DownRotate(G, s, i)
begin
  $X \longleftarrow \{v \mid 1 \leq s(v) \leq i\}$;
  Deallocation nodes in X from schedule s;
  Shift s up by i control steps[1];
  $s \longleftarrow$ PartialSchedule(G_X, s, X);
  Return (s, L, X); /* X: the set of nodes rotated down. */
end

[1]For examples in this chapter, we do not shift schedules up after each down-rotation so that the readers can compare the schedules before and after rotation easily.

CS	Mult	Adder
1	2	5
2	4	10
3	7	6
4	1	9
5	0	8
6	3	–

(a) a static schedule

Node	R	r	Node	R	r
0	3	1	6	2	0
1	3	1	7	2	0
2	2	0	8	5	1
3	3	1	9	2	0
4	2	0	10	5	1
5	2	0			

(b) R has larger depth than r

Figure 8.5: R and r both realize the same static schedule

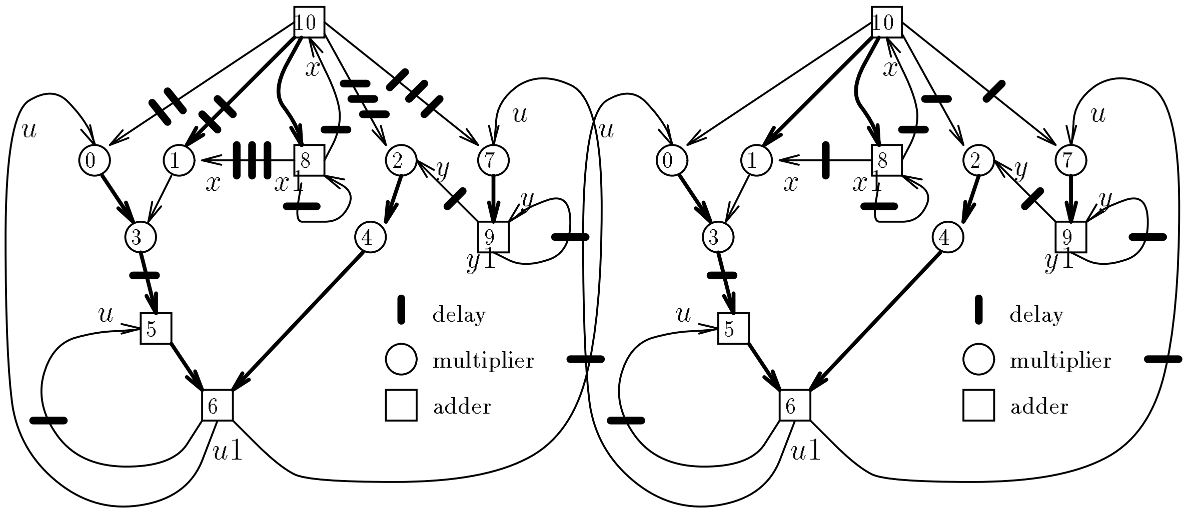
The basic algorithm can handle single-cycle and chained operations. It is implemented in an efficient way: only nodes in X and the nodes connected to X are involved in computing new weights in list scheduling and precedences. We do not really need to construct the DAG of G_X, so a lot of computation is saved. Any scheduling algorithm which does not change the scheduled part can serve as the algorithm for rescheduling.

Section 8.4 will refine the basic algorithm to handle the pipelined functional units and multi-cycle operations. In Section 8.5, two heuristics are designed to apply a sequence of down-rotations more effectively.

8.3.2 Depth of Loop Pipelining

For a given static schedule realized by a rotation function R, we present an algorithm to find a loop pipeline with shallow depth. The algorithm uses a single-source shortest path algorithm. This algorithm will be used only on the optimal schedules, whose length matches the lower bound given, or the final schedules found by our heuristics. For differential equation solver, an optimal schedule in Figure 8.5-(a) is obtained after 7 rotations of size 2. The corresponding rotation function R, shown in Figure 8.5-(b), has depth 4 as defined in Property 8.2. We will reduce its depth to 2 by finding a new retiming r. Although G_R and G_r are not equivalent, as shown in Figure 8.6, they realize the same static schedule.

For any static schedule, we use a simple Linear Programming formulation (LP form) to find a retiming r with smaller $\max_v r(v)$ such that the given static schedule is a DAG schedule of the retimed DFG G_r. From Property 8.3, we can generate



(a) G_R with depth 4 (a) G_r with depth 2

Figure 8.6: Two different graphs realize the same schedule

an LP form, which is the dual of a shortest path problem [46], and solvable in time $O(|V|\,|E|)$ [75].

Theorem 8.1 *Let $G = (V, E, d, t)$ be a DFG and s be a schedule of G. There exists a legal retiming r such that $s(u) + t(u) \leq s(v)$ for $d_r(u, v) = 0$ if and only if there exists a solution for the following LP form:*

$$r(v) - r(u) \leq d(u, v) \qquad \textit{for every } (u, v) \in E, \textit{and}$$

$$r(v) - r(u) \leq d(u, v) - 1$$

$$\textit{for every } (u, v) \in E \textit{ such that } s(u) + t(u) > s(v).$$

Proof: The inequality $r(v) - r(u) \leq d(u, v)$ ensures that $d_r(u, v) \geq 0$, i.e. r is a legal retiming of G. Since r is a legal retiming, the statement that $s(u) + t(u) \leq s(v)$ if $d_r(u, v) = 0$ is equivalent to the statement if $s(u) + t(u) > s(v)$ then $d_r(u, v) \geq 1$. Since $d_r(u, v) = d(u, v) + r(u) - r(v)$, the inequality $d_r(u, v) \geq 1$ becomes $r(v) - r(u) \leq d(u, v) - 1$. Thus, the theorem is proved. □

A single-source shortest path algorithm can find a retiming r with small $\max_v r(v)$. We construct a graph $H = (\{v_0\} \cup V, E_H, l_H)$ from the above LP form, where v_0 is a

CS	Mult	Mult	Adder
1	–	–	10
2	0	1	8
3	0'	1'	–
4	2	3	–
5	2'	3'	–
6	7	4	5
7	7'	4'	–
8	–	–	9
9	–	–	6

(a) initial schedule

CS	Mult	Mult	Adder
2	0	1	8
3	0'	1'	**10**
4	2	3	–
5	2'	3'	–
6	7	4	5
7	7'	4'	–
8	–	–	9
9	–	–	6

(b) first rotation

CS	Mult	Mult	Adder
3	–	–	10
4	2	3	**8**
5	2'	3'	–
6	7	4	5
7	7'	4'	–
8	**1**	–	9
9	**1'**	–	6
10	**0**	–	–
11	**0'**	–	–

(c) second rotation

Figure 8.7: Two rotations of size 1 for multi-cycle operations

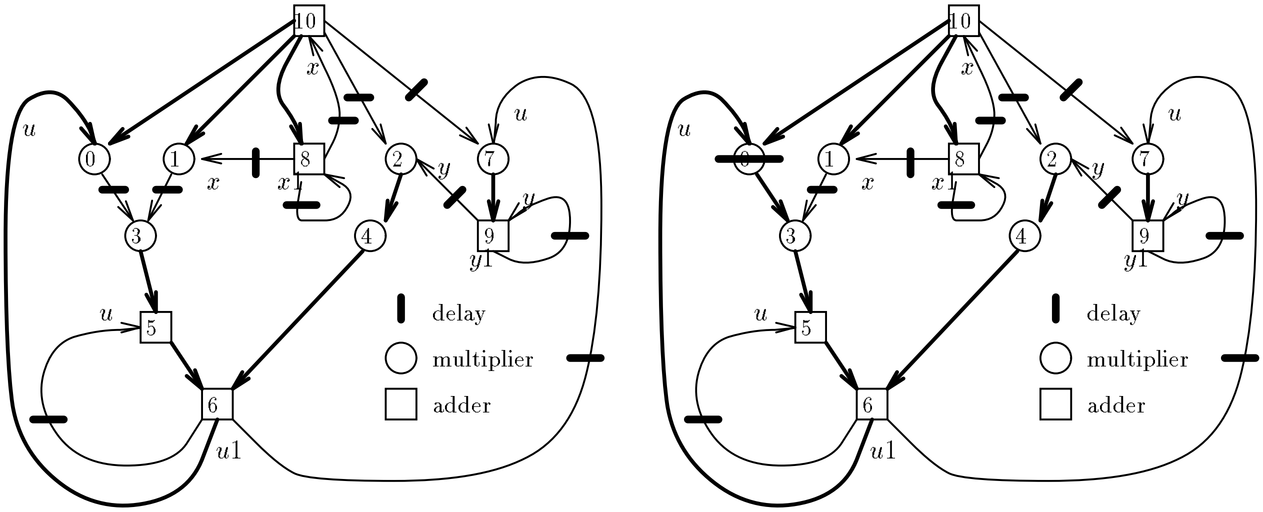
pseudo-node not in V. For every inequality $r(v) - r(u) \leq k$ in the LP form, where k is either $d(u,v)$ or $d(u,v) - 1$, we add an edge (u,v) to the edge set E_H with length $l_H(u,v) = k$. For every node v in V, an edge from v_0 to v with length 0 is added to E_H.

Lemma 8.3.1 *If there is a negative cycle in the graph H, there is no solution for the LP form in Theorem 8.1, and the given static schedule is illegal. Otherwise, the retiming $r(v) = -Sh(v)$ for every $v \in V$ is a solution for the LP form, where $Sh(v)$ is the length of the shortest path from node v_0 to node v in graph H.*

Proof: If there is a negative cycle in the graph H, the LP form is not consistent, and thus has no solutions. Assume that there is no negative cycle in the graph H. From the definition of $Sh(v)$, for every edge (u,v) in E_H, we know $Sh(u) \leq Sh(v) + l_H(u,v)$. By substituting $-r(v)$ into the above inequality, we have $r(v) - r(u) \leq l_H(u,v)$. Thus, $r(v) = -Sh(v)$ is a solution for the LP form in Theorem 8.1. $\square$

Note that the retiming obtained by this algorithm has the minimum depth and is normalized.

8.4 Multi-cycle Operations and Pipelined Functional Units



(a) $r(10) = r(0) = r(1) = 1$
after the second rotation

(b) Node 0 is split after the wrapping

Figure 8.8: The corresponding retimed graphs after rotations

In this section, we discuss how to use the basic algorithm to perform down-rotations on a schedule when *multi-cycle* operations and *pipelined functional units* are involved. We assume that the computation time of a multi-cycle operation is an integral multiple of the length of a control step. In our model for a pipelined functional unit, an operation can start execution in every control step, i.e. each stage in the pipeline takes one control step to finish. The computation time of an operation is the number of stages multiplied by the length of a control step.

When the starting time of a multi-cycle operation is in the first i control steps, the operation is rotated down by a rotation of size i. If the multi-cycle operation does not finish before the i-th control step, we still rotate the whole node down. The post-rotation schedule may be longer than the pre-rotation schedule because of some multi-cycle operations. The same is true for pipelined functional units. We will apply a technique, called *wrapping*, to overcome this problem.

Let's consider the initial optimal DAG schedule in Figure 8.7-(a). We assume that the multiplications take 2 time units, the additions take 1 time unit, and a control step contains 1 time unit. After a rotation of size 1, we get the schedule in Figure 8.7-(b), which has length 8, improved from 9 in (a). In the next rotation of size 1, we will rotate Node 0 and 1 entirely down, including their tails in control step 3. Then, we reschedule them according to the retimed graph in Figure 8.8-(a). At this point, we have a schedule which is one longer than the previous one because of the tail of a multi-cycle operation.

Here we introduce the idea of the *wrapped* schedule. We can *wrap* the tail of Node 0 up to the first control step of the schedule and obtain the *wrapped* schedule in Figure 8.9-(b). There are two conditions to check for a valid wrapping. First, there are spare resources, which is satisfied here. Second, the precedence constraints should be satisfied, Assuming that we wrap the tail of Node 0, denoted by 0', up, the corresponding retimed graph is shown in Figure 8.8-(b), where Node 0 is split by 1 delay. The new precedence constraint introduced is from Node 0' to Node 3, which needs to be satisfied. The valid wrapped schedule is shown in Figure 8.9-(b), which has length 8. In general, the outgoing edges of a wrapped node with one delay are the new precedence constraints, because they become zero-delay edges after the wrapping.

Thus, the schedule length of a DFG with multi-cycle operations is defined as *the length of the wrapped schedule*. The wrapping only needs to be performed after the last rotation. The rotations in our example still proceed from the unwrapped version of the schedule. After 8 rotations of size 1 from the initial schedule, we get a wrapped schedule of length 6. However, after 4 rotations of size 4, where some nodes are rotated more than once, we are able to get an unwrapped schedule of length 6, as shown in Figure 8.9-(c).

Sometimes, a wrapped schedule can be easily rotated to be an unwrapped one. A schedule of a loop can be regarded as a cylinder of instructions, which are repeatedly executed. We can consider any control step i as the first control step of the cylinder by rotating control steps before i without rescheduling them. Therefore, the wrapped schedule in Figure 8.9-(b) can be rotated into an unwrapped schedule by rotating down control step 3 with control step 4 as the first control step. The differences among these schedules are the prologues because they have different rotation functions.

For some DFGs, however, optimal schedules should be wrapped. Consider a simple DFG consisting of 3 nodes, A, B and C; all are multiplications. These three nodes

CS	Mult	Mult	Adder
3	–	–	10
4	2	3	8
5	2'	3'	–
6	7	4	5
7	7'	4'	–
8	1	–	9
9	1'	–	6
10	0	–	–
11	0'	–	–

(a) before wrapping

CS	Mult	Mult	Adder
3	**0'**	–	10
4	2	3	8
5	2'	3'	–
6	7	4	5
7	7'	4'	–
8	1	–	9
9	1'	–	6
10	**0**	–	–

(b) wrapped schedule

CS	Mult	Mult	Adder
17	7	0	8
18	7'	0'	10
19	4	3	9
20	4'	3'	–
21	1	2	5
22	1'	2'	6

(c) optimal unwrapped schedule

Figure 8.9: The wrapping of multicycle operations

form a cycle with 2 delays on the edge (A, B). If we would like to schedule it within 3 control steps (assuming enough resources), we have to split one of the nodes to obtain a wrapped schedule.

The rotation for nodes implemented as pipelined functional units is similar. The only difference is resource allocation. For a multi-cycle operation, we have to allocate a resource for every control step during which this operation is executing. In contrast, for a pipelined functional unit, we only need to allocate a resource in the control step in which this operation starts execution.

For example, we consider the differential equation example with one pipelined multiplier and one adder. Figure 8.10 shows the initial optimal DAG schedule in (a) and the schedule after a rotation of size 4 in (b). Although no resource is allocated at control step 11, the tail of Node 2 is running there. And the length of the schedule is 7, instead of 6. The wrapped version of Figure 8.10-(b) is in (c), which is valid after checking the precedence constraints in the corresponding retimed graph. Since there are 6 multiplications and only one pipelined multiplier, a schedule of length 6 must be wrapped.

8.5 Heuristics

In this section, we propose some heuristic algorithms based on the technique of rotation scheduling. It is assumed that the input DFG already has been retimed into one

CS	Mult	Adder
1	–	10
2	1	8
3	0	–
4	2	–
5	3	–
6	4	–
7	7	5
8	–	6
9	–	9

(a) initial schedule

CS	Mult	Adder
5	3	10
6	4	8
7	7	5
8	1	6
9	0	9
10	2	–
11	–	–

(b) unwrapped optimal schedule

CS	Mult	Adder
5	3	10
6	4	8
7	7	5
8	1	6
9	0	9
10	2	–

(c) wrapped optimal schedule

Figure 8.10: A rotation of size 4 for the pipelined functional units

with the optimal cycle period (without resource constraints) when necessary. In the algorithm descriptions, we use FullSchedule(G) to denote a procedure which returns a DAG schedule of G.

We partition the heuristic algorithms into phases. For a given parameter δ, δ rotations are performed for each phase. An initial schedule S_{init} and a set Q of current shortest schedules of length L_{opt} are given as inputs. The outputs are the resultant schedule S after all rotations and a set Q of shortest schedules of new length L_{opt}. For a schedule of length L, a rotation of size i with $i \geq L$ is illegal. During a phase of size i, when the rotation size is larger than the schedule length L, we divide i by 2 until $i < L$. The following procedure describes a rotation phase of size i.

RotationPhase($S_{init}, L_{opt}, Q, G, i, \delta$)
begin /* Perform a phase of size i with δ rotations. */
    $S \longleftarrow S_{init}$; $L \longleftarrow$ length of S_{init};
    $i' \longleftarrow i$; $R \longleftarrow \emptyset$;
    for $j = 1$ **to** δ **do begin**
        while $i' \geq L$ **do** $i' \longleftarrow \lceil i'/2 \rceil$;
        $(S, L, X) \longleftarrow$ DownRotate(G_R, S, i');
        $R \longleftarrow R \circ X$;
        if $L < L_{opt}$ **then begin**
            $L_{opt} \longleftarrow L$; $Q \longleftarrow \{S\}$;   **end** ;
        else if $L = L_{opt}$ **then** $Q \longleftarrow Q \cup \{S\}$;
    end
    Return(L_{opt}, Q, S, R);
end   /* R is the rotation function for the schedule S */

In the worst case where i is large, each node is rotated at most δ times in a phase. It takes time $O(|V| + |E|)$ to rotate all nodes down once. The time complexity of a rotation phase is $O(\delta\,|E|)$.

In the first simple heuristic, as shown below with parameters δ and ρ, we start every phase from the initial list scheduling of the original DFG. Phases of different sizes are performed according to the length of the initial schedule. The phases in this heuristic are performed independently of each other. Hence, the behavior of this heuristic is more predictable. The larger the value of δ, the better the performance in each phase. The larger the value of ρ, the better the final schedule we obtain.

```
Heuristic 1(G, δ, ρ)
begin /*   δ : the number of down-rotations in each phase.
           ρ : the range of phases of different sizes. */
      S_init ⟵ FullSchedule(G);
      L_opt ⟵ length of S_init;    Q ⟵ {S_init};
      for i = 1 to ρL do    /* For every phase of size i */
           (L_opt, Q, S, R) ⟵ RotationPhase(S_init, L_opt, Q, G, i, δ);
end
```

In the second heuristic, each phase uses the final rotation function of the previous phase as an initial retiming function. Since the schedule is improved from phase to phase, the rotation function tends to reflect a better DAG structure for the DFG. These rotation functions give us more variations of the input DFG, and may expose us to chances of better schedules. The initial scheduling of a phase is performed on the retimed DFG. For a range of sizes chosen, we perform the phases in decreasing order of sizes. This heuristic is described as follows.

```
Heuristic 2(G, δ, ρ)
begin /* δ, ρ : as in Heuristic 1. */
   S ⟵ FullSchedule(G);
   L_opt ⟵ length of S;    Q ⟵ {S};    R ⟵ ∅;
   /* iterative compaction */
   for i = ρL to 1 by −1 do begin    /* size-i phase */
      (L_opt, Q, S, R) ⟵ RotationPhase(S, L_opt, Q, G, i, δ);
      /* Find a new initial schedule for the next phase */
      S ⟵ FullSchedule(G_R);    L ⟵ length of S;
      if L < L_opt then    L_opt ⟵ L;  Q ⟵ {S}; end ;
      else if L = L_opt then    Q ⟵ Q ∪ {S};
   end
end
```

The time complexity of both heuristics is $O(\delta\,|V|\,|E|)$, assuming that the schedule

Benchmark	#Mults	#Adds	CP	IB
5-th Order Elliptic Filter	8	26	17	16
Differential Equation	6	5	7	6
4-stage Lattice Filter	15	11	10	2
All-pole Lattice Filter	4	11	16	8
2-cascaded Biquad Filter	8	8	7	4

Table 8.1: Characteristics of the benchmarks

length is no more than $|V|$. These simple heuristics give very good experimental results, which will be presented in the next section. In most cases, the two heuristics get the same results. However, the second heuristic gives better schedules in one of the cases. The next section will report the results based on the second heuristic.

8.6 Experimental Results

We have experimented with our strategy on several benchmarks and the results are very promising. All our results are as good as or better than other systems which perform loop pipelining under the same assumptions. In fact, we can prove all the results except one are optimal. Our results are compared against the following three systems: Percolation-based scheduling (PBS) with loop pipelining [68, 58], MARS design system [77], and the scheduler by Lee *et. al* [50]. Cathedral II is not included for comparison because of insufficient data in the paper [31].

We use LB to denote the lower bounds we derive for different resource constraints, RS to denote our Rotation Scheduling, and the number in parentheses is the pipeline depth. The figures for the other systems appearing in the following tables are adopted from the papers referenced above.

In our experiments, we use a simple list scheduling for both procedures FullSchedule and PartialSchedule with the number of descendants as the weight function. The input DFGs are the original DFGs of the benchmarks. In most cases, the two heuristics presented in Section 8.5 get the same results, except for the $2A\ 1M_p$ case of Table 8.2, where the second heuristic gets a better result. The schedules we obtained usually

Resources	**LB**	PBS	MARS	Lee *et. al*	**RS**
Nonpipelined Multipliers					
$3A\ 3M$	**16**	16	n/a	16	16 (2)
$3A\ 2M$	**16**	17	n/a	16	16 (2)
$2A\ 2M$	**17**	17	n/a	17	17 (2)
$2A\ 1M$	**17**	20	n/a	19	19 (2)
Pipelined Multipliers					
$3A\ 2M_p$	**16**	16	n/a	16	16 (2)
$3A\ 1M_p$	**16**	16	16	16	16 (2)
$2A\ 1M_p$	**17**	18	17	17	17 (2)

Table 8.2: Results for the elliptic filters

have shallow pipeline depth, like 2 or 3.

It is assumed that the computation time of an adder for one addition is $40ns$, the computation time of a multiplier for a multiplication is $80ns$, and the clock cycle period for a control step is $50ns$, with $10ns$ for the latch time of buffers. The pipelined multiplier, denoted by M_p, is assumed to consist of 2 stages, each of which takes no more than $40ns$. The numbers in the tables are the number of control steps.

The experiments are done on the five benchmarks in Table 8.1. The CP (Critical Path) equals the minimum-length schedule without loop pipelining. The IB (Iteration Bound) is the theoretical lower bound on the schedule length, which is the ceiling of the maximum time-to-delay ratio among all cycles in the DFG [71].

The results for the elliptic filters [2] are shown in Table 8.2, and the rest of the results are shown in Table 8.3. Our results compare favorably with those of other systems. Every experiment is finished within seconds on a DEC 5000 workstation. The algorithm was written in the C programming language. For example, in 2.5 seconds RS can find the optimal schedules shown for the elliptic filter; in less than 1 second for the other four benchmarks. For the elliptic filter, the number of optimal schedules found ranges from 15 to 35, depending on the availability of resources. The first optimal schedule is usually found within 1 second.

[2] As pointed out by Reference [67], there are errors in the DFG of [44]. We derived the correct DFG from the signal flow graph description of the elliptic filter in [44].

Resources	Pipelined Multipliers LB	MARS	**RS**	Nonpipelined Multipliers Resources	LB	**RS**
Differential Equation						
$1A\ 1M_p$	6	n/a	6(2)	$1A\ 2M$	6	6(2)
				$1A\ 1M$	12	12(2)
The 4-stage Lattice Filter						
$6A\ 8M_p$	2	2	2(6)	$6A\ 15M$	2	2(5)
$4A\ 5M_p$	3	n/a	3(4)	$4A\ 10M$	3	3(5)
$3A\ 4M_p$	4	n/a	4(3)	$3A\ 8M$	4	4(3)
$3A\ 3M_p$	5	n/a	5(2)	$3A\ 6M$	5	5(4)
$2A\ 3M_p$	6	n/a	6(2)	$2A\ 5M$	6	6(2)
$2A\ 2M_p$	8	n/a	8(2)	$2A\ 4M$	8	8(2)
All-pole Lattice Filter						
$3A\ 2M_p$	8	8	8(3)	$3A\ 2M$	8	8(3)
$2A\ 2M_p$	9	n/a	9(2)	$2A\ 2M$	9	9(2)
$2A\ 1M_p$	9	n/a	9(2)	$2A\ 1M$	10	10(2)
$1A\ 1M_p$	11	n/a	11(2)	$1A\ 1M$	11	11(2)
The 2-cascaded Biquad Filter						
$2A\ 2M_p$	4	4	4(2)	$2A\ 4M$	4	4(2)
$2A\ 1M_p$	8	n/a	8(2)	$2A\ 3M$	6	6(2)
$1A\ 2M_p$	8	n/a	8(2)	$1A\ 2M$	8	8(2)
$1A\ 1M_p$	8	n/a	8(2)	$1A\ 1M$	16	16(2)

Table 8.3: The experimental results for the other four benchmarks

In addition, we compare our results with the lower bounds (LB) we derive. Larger lower bounds on the schedule lengths are obtained when we have more strict resource constraints. The detailed derivations of those lower bounds are presented in the appendix to this chapter. For the elliptic filter, we achieve the derived lower bounds, except in the case of $2A\ 1M$. In Table 8.3 we always meet the lower bounds in every resource requirement for the other four benchmarks.

8.7 Conclusion

In this chapter, we presented a theoretical foundation and experimental results for a new transformation, called *rotation*, for data-flow graphs. This technique is simple and easy to implement. It can be incorporated with any DAG scheduling algorithm which does not perform loop pipelining.

We showed the effectiveness of the technique of rotation scheduling under various resource constraints in our experiments. In addition, through a sequence of rotations, many optimal schedules can be found, which expose more chances of optimization for the following stages of high-level synthesis, eg. connection binding, allocation or data-path generation.

Rotation and retiming are both techniques for pipelining. Although strictly speaking, it is a restricted form of retiming, the concept of rotation is easier to manipulate in an optimization process. We have demonstrated its effectiveness by very simple heuristics, based on the procedure RotationPhase. There is a class of heuristics to be developed by varying the parameters δ and ρ.

The advantage of the second heuristic over the first one is that the initial schedule is obtained from a DFG refined by a certain retiming function. Even if the original DFG is optimally retimed, we may not get a good schedule under a certain resource constraint. There are many optimal retimings. If we start from a good optimal retiming, the performance of heuristic algorithms may be improved. The next chapter will present methods to obtain many optimal retimings by using rotations.

Although rotation scheduling almost always finds optimal schedules for these benchmarks, more experiments need to be performed on larger benchmarks. The elliptic filter is the largest benchmark used in the systems we compared with. We need to seek large benchmarks to challenge the performance of scheduling algorithms.

This chapter does not assume the computations of nodes have any special properties like associativity and distributivity. The work we have discussed gives a generic scheduling technique to deal with any DFG without considering the properties of each node. However, in the future we would like to apply transformation techniques to the rotation scheduling, for example, tree height reduction [58].

For a nested loop program, rotation scheduling can pipeline the innermost loop first. After the inner loop is pipelined, we can treat it as a multi-cycle operation while we pipeline the outer loop. If the DFG has mutual exclusive operations, we can check the conditions while assigning an operation to a control step. However, if there are conditional data flows, as in most imperative languages, we need to design a conditional data flow model to capture the situation.

The non-trivial theoretical lower bounds in the appendix are obtained manually by studying the subgraph containing critical and almost critical cycles. We would like to automate this process to obtain more accurate estimation of lower bounds so that the search time for optimal schedules can be reduced.

8.8 Appendix: Theoretical Lower Bounds under Resource Constraints

In this section, we introduce some techniques to obtain the lower bounds of the schedule length in Section 8.6. We first introduce a well-known lower bound from the structure of the DFG, and simple lower bounds from the resource availability. Then, we provide our proofs for the nontrivial lower bounds obtained for the elliptic filters and the all-pole lattice filter from both the DFG structure and the resource availability. The other lower bounds appearing in Section 8.6 comes from either the structure of the DFG or the resource availability.

Lower Bounds from the Structure of the DFG

A lower bound, called *iteration bound* $\mathcal{B}(G)$ [71], of the cycle period was introduced in Subsection 2.2.2. The iteration bound $\mathcal{B}(G)$ is derived as the *maximum time-to-delay ratio* over all cycles of the DFG G. This iteration bound is a lower bound on the schedule length without resource constraints.

A schedule with length equal to the iteration bound is called a *rate-optimal* schedule. A cycle which gives the iteration bound is called a *critical cycle*, l_{cr}. For example, the $\mathcal{B}(G)$ of the DFG in Figure 8.1 is 6 if the multiplications take two time units and additions take one time unit. The critical cycle consists of Nodes 0, 3, 5, and 6 with $T(l_{cr}) = 6$ and $D(l_{cr}) = 1$.

Lower Bounds from Resource Availability

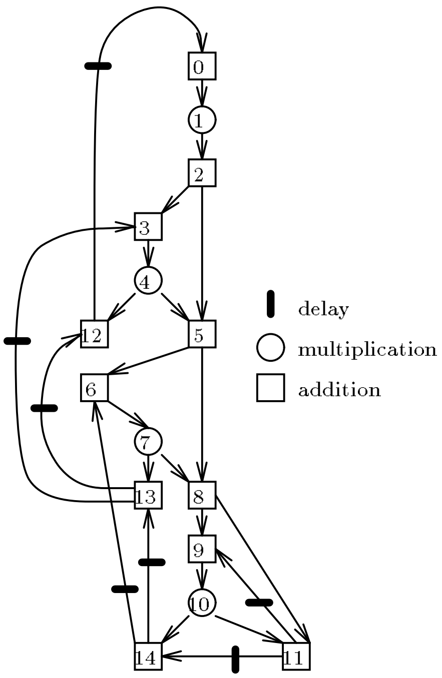


Figure 8.11: The DFG of all-pole lattice filter

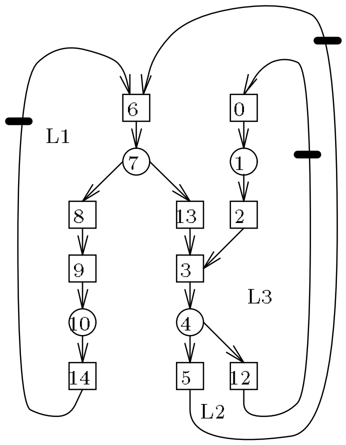


Figure 8.12: Three critical cycles in the all-pole lattice filter

For a type-A functional unit, let F_A be the number of type-A functional units available, N_A the number of operations using type-A functional units in the DFG, and C the number of time units in a control step. Let T_A be the time it takes to perform an operation on a type-A non-pipelined functional unit. The lower bound for the number of control steps is

$$\begin{cases} \left\lceil \frac{N_A}{F_A} \right\rceil & \text{if type-}A\text{ functional units are pipelined,} \\ \left\lceil \frac{N_A}{F_A} \cdot \left\lceil \frac{T_A}{C} \right\rceil \right\rceil & \text{if } T_A > C \quad \text{(multi-cycle operation),} \\ \left\lceil \frac{N_A}{F_A} \cdot \frac{1}{\lfloor C/T_A \rfloor} \right\rceil & \text{if } C > T_A \quad \text{(single-cycle (chained) operation)} \end{cases}$$

The lower bound from the resource availablility can be obtained by taking the maximum of the lower bounds for each resource type.

8.8.1 All-Pole Lattice Filter

The iteration bound of this filter, whose DFG is in Figure 8.11, is 8. In the all-pole lattice, there are three critical cycles sharing nodes as shown in Figure 8.12. Each of them has total computation time 8 and one delay. In order to have a schedule of length 8, the operations on a critical cycle need to be scheduled tightly; no one should be lagged behind. For a rate-optimal schedule, we need at least 2 adders and 3 multipliers so that no operations on the critical cycle is lagged.

If there are only two adders, at least one addition is delayed, and the schedule will be delayed by at least the time it takes to execute an addition, which is one control step. Therefore, the lower bound of the schedule length is 9 control steps when there are only two adders available.

If there is only one multiplier, at least one multiplication is delayed. Thus, a rate-optimal schedule is delayed at least one control step for a pipelined multiplier, and at least two control steps for a non-pipelined multiplier. Therefore, the lower bound of the schedule length is 10 control steps when there is only one non-pipelined multiplier available.

8.8.2 The 5-th Order Elliptic Filter

The iteration bound of this filter, whose DFG is in Figure 8.13, is 16. As shown in Figure 8.14, there are two critical cycles, L1 and L2, and one almost critical cycle, L3, sharing nodes. There is only one delay in each of these cycles. The total computation time of L3 is 15, which means it can tolerate a lag of 1 time unit for a rate-optimal schedule. No matter where we insert the lag in cycle L3, there will still be 3 additions in a row. Thus, a rate-optimal schedule needs at least 3 adders. If there are only two adders, at least one addition is delayed, and the schedule will be delayed by at least one control step. Therefore, the lower bound of the schedule length is 17 control steps when there are only two adders available.

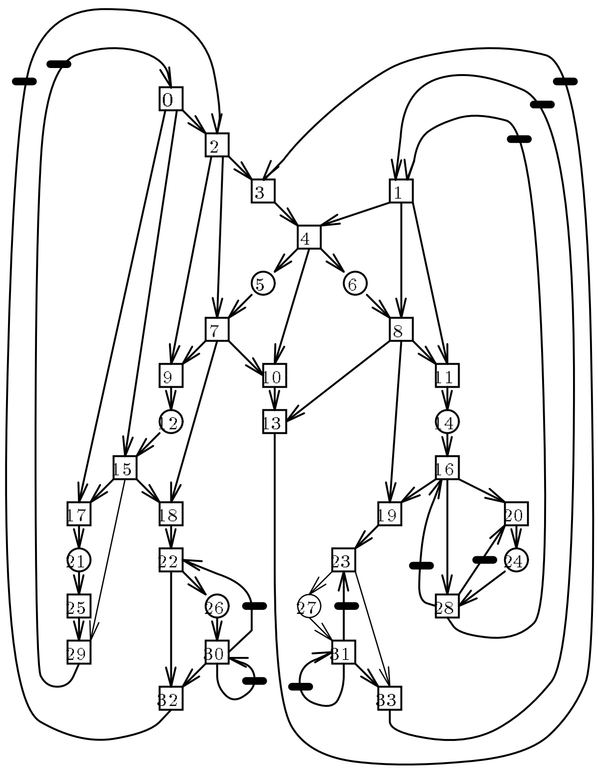


Figure 8.13: The DFG of elliptic filters

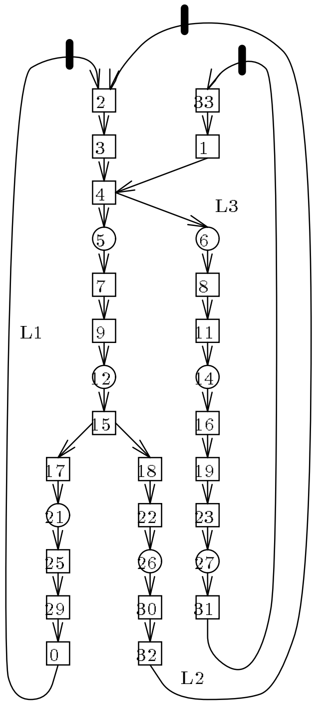


Figure 8.14: Two critical cycles and an almost critical cycle in the elliptic filter

Chapter 9

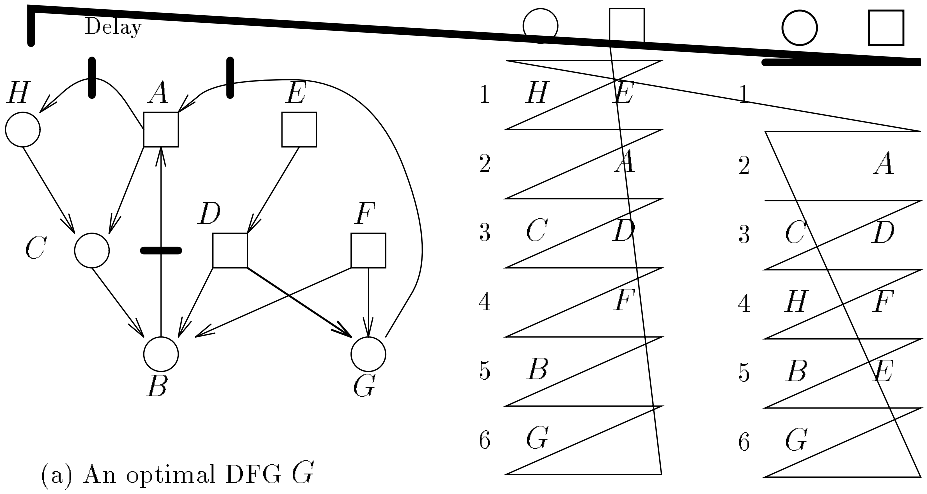
Optimal Rotations

In the previous chapter, we proposed a novel and flexible transformation, called a *rotation*, based on retiming. The basic idea is to transform a DFG in a simple and legal way so that the new DFG is likely to have a better schedule. A restricted form of rotation is used to compact an existing schedule, even though the given DAG part is already scheduled optimally. This chapter continues to study the general rotation technique.

9.1 Introduction

Although the retiming technique can improve the cycle period, the traditional retiming algorithm can only find one retiming to achieve the minimum cycle period without resource constraints. A retiming achieving the minimum cycle period is called *an optimal retiming*, and the resulting DFG is called *an optimal DFG*. For a given DFG, there may exist many different optimal retimings; each of them has different DAG structure. For resource-constrained scheduling, the structure of a DAG may give lower bounds under certain resource constraints. This chapter will develop techniques to find all optimal retimings to be combined with resource constrained scheduling algorithms.

Finding more than one optimal DFG is important for scheduling under resource constraints. Consider an optimal DFG in Figure 9.1-(a). The nodes B, C, G, H need



(a) An optimal DFG G with cycle period 3

(b) A static schedule

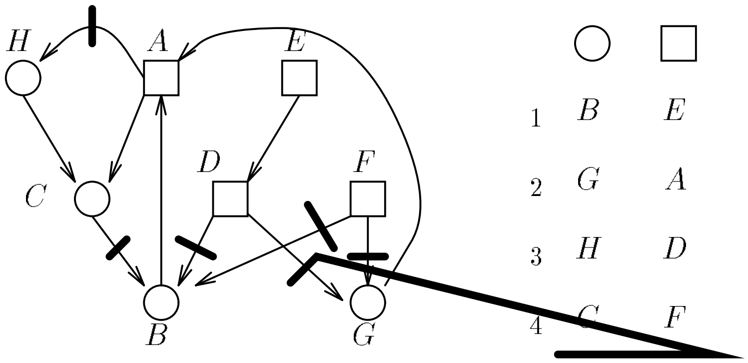
(c) After rotating H and E

Figure 9.1: An optimal DFG G and a static schedule

type-1 resources, denoted by circles; the rest of nodes need type-2 resources, denoted by boxes. For this example, we assume that each computation takes one control step. Figure 9.1-(b) shows an optimal static schedule of length 6 for the corresponding DAG. However, another optimal DFG in Figure 9.2 gives a schedule of length 4 found by a list scheduling algorithm. Actually, this alternative optimal DFG has at least three different optimal schedules. This shows that an optimal DFG might still give a bad schedule under certain resource constraints. We will try to overcome the problem of scheduling cyclic DFGs with resource constraints by finding other optimal DFGs using rotations.

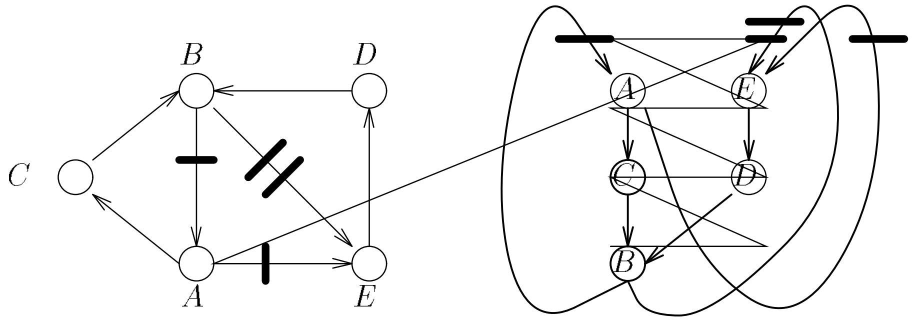
In this chapter, fundamental properties of the rotation transformation are presented. We give a simple necessary and sufficient condition to ensure that a given optimal DFG can be rotated to another optimal DFG. Such a rotation is called an *optimal rotation*. The transformation of any optimal DFG to any other optimal one can be achieved by a sequence of *optimal rotations*. Therefore, we can repeatedly use the optimal rotation technique to transform an optimal DFG and compact a schedule to obtain a better schedule until a satisfactory schedule is found. We believe that our result lays a good foundation for the resource scheduling of loops.

The operation of a rotation was defined in Section 8.2. Some properties of optimal rotations are derived in 9.2. Several procedures are presented to find an optimal DFG



(a) Alternative optimal DFG with cycle period 3 (b) a schedule

Figure 9.2: An alternative optimal DFG for G



(a) An optimal DFG G (b) its DAG in the dotted box

Figure 9.3: An optimal DFG and the corresponding DAG

G_r from a given optimal DFG in Sections 9.3 and 9.4. Finally, the results from the previous chapter are combined to show how to obtain the optimal schedule in Figure 9.2.

9.2 Properties of Optimal Rotations

In this section, several procedures are presented to find an optimal DFG G_r from a given optimal DFG in time $O(|V|^2)$ so that the designer can have alternative starting points to generate good schedules.

For an optimal DFG G, we say that a down-rotatable (resp. up-rotatable) set X is *optimal* if set X can transform G into another optimal DFG G_X (resp. G_{-X}). We will focus on the properties of down-rotations. Similar lemmas and algorithms in this section can be derived for up-rotations. For practical problems, it might be better to apply alternately both rotations with a mixed strategy. Let us first introduce the functions D and T, which will be used to characterize optimal retiming functions, and optimal rotations.

Definition 9.1 [53] *Let $G = (V, E, d, t)$ be an DFG. For a path p from node u to node v, let $d(p)$ be the total number of delays of a path p, and $t(p)$ the total computation time of nodes in p, including u and v. For every nodes u and v in V,*

(a) *Define $D(u, v) = \min\{d(p) \mid$ for every path from u to v in $G\}$.*

(b) *Define $T(u, v) = \max\{t(p) \mid$ for every path from u to v in G such that $d(p) = D(u, v)\}$.*

(When there is no path from u to v, $D(u, v)$ is undefined and $T(u, v) = 0$.)

The value $D(u, v)$ is the minimum number of delays among all the paths from u to v. The value $T(u, v)$ is the maximum computation time among every path from u to v such that the number of delays is equal to $D(u, v)$. The Ford-Warshall algorithm for all-pairs shortest paths can compute D and T in time $O(|V|^3)$. Our algorithms designed in this section involve only the functions D and T. Furthermore, it is easy to observe that for any retiming function r, we have $D_r(u, v) = D(u, v) + r(u) - r(v)$ and $T_r(u, v) = T(u, v)$ [53]. The D and T functions of the DFG in Figure 9.3 are shown in Figure 9.4. We will use the DFG in Figure 9.3 as an example throughout this section.

One characterization of the cycle period in terms of D and T is the following.

Lemma 9.2.1 [53] *$\mathcal{C}(G) \le c$ if and only if for every u and v in V, either $D(u, v) \ge 1$ or $T(u, v) \le c$.*

The following lemma characterizes the necessary and sufficient condition of an *optimal* down-rotatable set.

Lemma 9.2.2 *Let $G = (V, E, d, t)$ be an optimal DFG with cycle period $\hat{c}$, and set X a down-rotatable subset of V. The set X is optimal if and only if for every $u \in V - X$ and $v \in X$, either $D(u, v) \ge 2$ or $T(u, v) \le \hat{c}$.*

D \ T	A	B	C	D	E
A		0 3	0 2	1 3	1 2
B	1 2		1 3	2 4	2 3
C	1 3	0 2		2 5	2 4
D	1 3	0 2	1 4		2 4
E	1 4	0 3	1 5	0 2	

Figure 9.4: The D and T functions

Proof:

Assume that X is optimal, i.e. $\mathcal{C}(G_X) = \hat{c}$. For every $u \in V - X$ and $v \in X$, we have

$$\begin{aligned} D_X(u,v) &= D(u,v) + X(u) - X(v) \\ &= D(u,v) - 1. \end{aligned}$$

Since $\mathcal{C}(G_X) = \hat{c}$, from Lemma 9.2.1 we know either $D_X(u,v) \geq 1$ or $T_X(u,v) \leq \hat{c}$. Since $T_X(u,v) = T(u,v)$, the only-if part is proved.

For the if part, we claim that for every u and v in V, either $D_X(u,v) \geq 1$ or $T_X(u,v) \leq \hat{c}$. There are three cases. Note that $T(u,v) = T_X(u,v)$ for any set X.

- If u and v both are in X or in $V - X$, we have $D_X(u,v) = D(u,v)$.
- If $u \in X$ and $v \in V - X$, we have $D_X(u,v) = D(u,v) + 1$. Since $\mathcal{C}(G) = \hat{c}$ and $D(u,v) \geq 0$, we know $D_X(u,v) \geq 1$ or $T_X(u,v) \leq \hat{c}$.
- If $u \in V - X$ and $v \in X$, from the proof of the only-if part and the assumption that $D(u,v) \geq 2$ or $T(u,v) \leq \hat{c}$, we know $D_X(u,v) \geq 1$ or $T_X(u,v) \leq \hat{c}$.

□

A down-rotation is *optimal* if an optimal down-rotatable set is rotated down. Theorem 9.1 shows that optimal down-rotation is a technique general enough to reach all optimal retiming functions through optimal down-rotations only. For any pair of optimal DFGs, we will show that there is a sequence of optimal down-rotations to transform G into G_r.

Theorem 9.1 *Let $G = (V, E, d, t)$ and G_r be optimal DFGs. Let $Min(r)$ be $\min_{v \in V} r(v)$, and $M(r)$ be $\max_{v \in V} r(v) - Min(r)$. There is a sequence of optimal down-rotations on sets U_1, ..., and $U_{M(r)}$ respectively to transform G to G_r, where U_i is the set of nodes v such that $i \leq r(v) - Min(r) \leq M(r)$.*

Proof: We construct a sequence of optimal down-rotations from a normalized retiming function r'. Let $r'(v) = r(v) - Min(r)$ for every $v \in V$. We know that $G_r = G_{r'}$, and that $0 \leq r'(v) \leq M(r)$ for every $v \in V$. Let V_i be the set of nodes v such that $r'(v) = i$, for every $0 \leq i \leq M(r)$. For brevity, $M(r)$ is denoted by M. We consider the following sequence of DFGs:

$$G = G_{r_0} \xrightarrow{V_1 \cup \cdots \cup V_M} G_{r_1} \xrightarrow{V_2 \cup \cdots\cdots \cup V_M} \cdots G_{r_{M-1}} \xrightarrow{V_M} G_{r_M} = G_r,$$

where G_{r_i} is the DFG obtained by rotating $\cup_{i \leq j \leq M} V_j$ down from the DFG $G_{r_{i-1}}$, for every $1 \leq i \leq M$. Let U_i be $\cup_{i \leq j \leq M} V_j$. The sequence is rewritten as

$$G_{r_0} \xrightarrow{U_1} G_{r_1} \xrightarrow{U_2} \cdots G_{r_n} \xrightarrow{U_{n+1}} G_{r_{n+1}} \cdots\cdots G_{r_{M-1}} \xrightarrow{U_M} G_{r_M}.$$

Since V_M is not an empty set, the set U_i for every i is not empty. We need to prove that the sequence of down-rotations on U_1, ..., and U_M is a sequence of optimal down-rotations, which transforms G to G_r.

Let n be any value where $0 \leq n < M - 1$. We claim that DFG $G_{r_{n+1}}$ is optimal if DFG G_{r_n} is optimal, i.e. U_{n+1} is an optimal down-rotatable set with respect to G_{r_n}. If the claim is true for every n, the theorem is proved by applying the claim from $n = 1$ to $n = M - 2$.

First, we show that U_{n+1} is down-rotatable with respect to G_{r_n}, i.e. for every $u \notin U_{n+1}$ and $v \in U_{n+1}$, $D_{r_n}(u, v) \geq 1$. Consider a pair of nodes $u \notin U_{n+1}$ and $v \in U_{n+1}$. We express $D_{r_n}(u, v)$ in terms of $D_{r_M}(u, v)$.

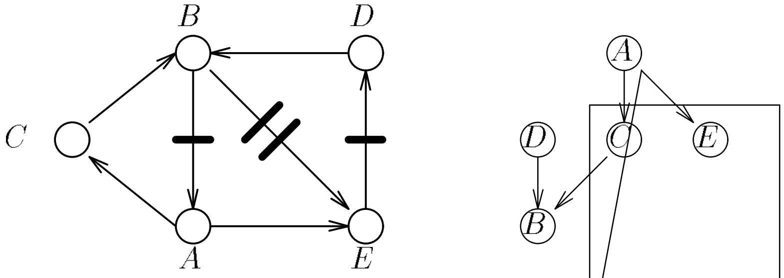
$$\begin{aligned} D_{r_n}(u, v) &= D(u, v) + r_n(u) - r_n(v) \\ &= D_{r_M}(u, v) - r_M(u) + r_n(u) + r_M(v) - r_n(v) \end{aligned}$$

From the derivation of U_i, we know U_i is a subset of U_{i-1} for every i. Thus, a node u that is not in U_{n+1} will not be rotated in the subsequence starting from G_{r_n}. Therefore, for a node $u \notin U_{n+1}$ we know $r_M(u) = r_n(u)$. However, a node v belonging U_{n+1} will be rotated at least once in the subsequence starting from G_{r_n}. Thus, for a node $v \in U_{n+1}$ we have $r_M(v) \geq r_n(v) + 1$. Therefore, we derive $D_{r_n}(u,v) \geq D_{r_M}(u,v) + 1$ for every $u \notin U_{n+1}$ and $v \in U_{n+1}$. Since $D_{r_M}(u,v) \geq 0$, we have $D_{r_n}(u,v) \geq 1$ for every $u \notin U_{n+1}$ and $v \in U_{n+1}$.

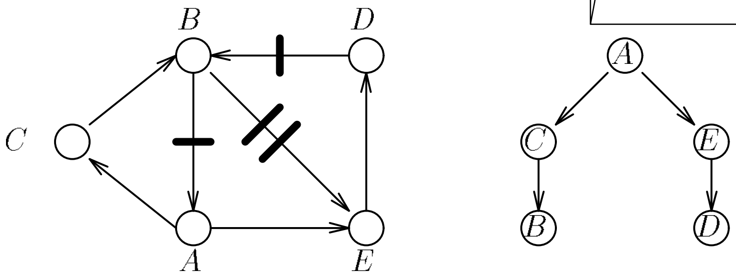
From Lemma 9.2.2, we know that U_{n+1} is optimal down-rotatable with respect to G_{r_n} if and only if for every $u \notin U_{n+1}$ and $v \in U_{n+1}$, either $D_{r_n}(u,v) \geq 2$ or $T_{r_n}(u,v) \leq \hat{c}$. We will prove the above condition from the assumption that G_{r_M} is optimal, i.e. for every $u, v \in V$ we have $D_{r_M}(u,v) \geq 1$ or $T_{r_M}(u,v) \leq \hat{c}$. Consider a pair of nodes $u \notin U_{n+1}$ and $v \in U_{n+1}$. From the derivation in the last paragraph, we know that $D_{r_n}(u,v) \geq D_{r_M}(u,v) + 1$. Also, note that $T_{r_n} = T_{r_M}$ for any n. From the optimality of G_{r_M}, we know for every $u \notin U_{n+1}$ and $v \in U_{n+1}$, either $D_{r_n}(u,v) \geq 2$ or $T_{r_n}(u,v) \leq \hat{c}$. The claim is proved. $\square$

Based on the above theorem, we can search through all optimal DFGs from any given optimal DFG by applying all possible optimal down-rotations systematically. In the next two sections, we present two techniques to find optimal down-rotatable sets. The first technique, *totally-optimal transformations*, finds all down-rotatable sets such that any down-rotatable subset of them is optimal. Such totally-optimal sets usually appear in the non-critical part of the DFG, for example, the nodes which do not belong to any critical cycle or any almost critical cycle. Although *totally-optimal* sets may not exist in some DFGs, we believe that most of the DFGs in practical applications have a few totally-optimal down-rotatable or up-rotatable set to be explored.

In Section 9.4, another more general technique, *minimal optimal transformations*, can find a minimal optimal down-rotatable set from any given set. This technique will lead to a branch-and-bound approach to search for all optimal DFGs.



(a) DFG $G_{\{\mathsf{E}\}}$ and the corresponding DAG



(b) DFG after $G_{\{\mathsf{E},\mathsf{D}\}}$ and the corresponding DAG

Figure 9.5: The totally optimal transformation

9.3 Totally Optimal Transformation

A down-rotatable set X is *totally-optimal* if every down-rotatable subset of X is optimal. If a node v satisfies $D(u, v) \geq 2$ or $T(u, v) \leq \hat{c}$ for every $u \in V - \{v\}$, we say node v is *totally optimal*. Note that a totally optimal node may not be down-rotatable. All the totally optimal nodes can be found in $O(|V|^2)$ time by checking functions D and T. For example, consider the DFG G in Figure 9.3-(a). Its D and T functions are shown in Figure 9.4. The totally-optimal down-rotatable nodes are B, D, and E. Actually, any down-rotatable subset of the set of totally-optimal nodes is totally-optimal, which is shown in the following Lemma.

Lemma 9.3.1 *A down-rotatable set X is totally optimal if and only if every node $v \in X$ is totally optimal.*

Proof: For the if part, assume that every v in X is totally optimal, i.e.

$$\forall v \in X, \forall u \in V - \{v\}, D(u, v) \geq 2 \text{ or } T(u, v) \leq \hat{c}.$$

Let Y be any subset of X. Because $V - Y \subseteq V - \{v\}$ for any v in Y, it is obvious that

$$\forall u \in V - Y, \forall v \in Y, D(u, v) \geq 2 \text{ or } T(u, v) \leq \hat{c}.$$

For the only-if part, assume that X is totally optimal, i.e. for every down-rotatable $Y \subseteq X$, Y is optimal. Consider a node v in X. We claim that v is totally optimal. If the singleton set v is down-rotatable, the claim is proved; therefore, we assume that v is not down-rotatable. Let Y be the set of nodes in X that can reach v through zero-delay path, including v, i.e. $Y = \{u \,|\, u \text{ is a node with } D(u, v) = 0\}$. Since Y is down-rotatable and a subset of X, Y is optimal. Since Y is optimal down-rotatable and v belongs to Y, for every $u \in V - Y$, we have $D(u, v) \geq 2$ or $T(u, v) \leq \hat{c}$. Since G is optimal, for every $u \in Y$, we have $D(u, v) = 0$ and $T(u, v) \leq \hat{c}$. Thus, v is totally optimal. $\square$

For example, the set {E, D} is totally-optimal, but not the set {B, E, D}, which is not down-rotatable in G. The down-rotatable subsets are {E, D} and {E}. Therefore, we know that $G_{\{\mathsf{E},\mathsf{D}\}}$ and $G_{\{\mathsf{E}\}}$ are optimal, as shown in Figure 9.5. From the above lemma, we know that a totally optimal down-rotatable set is composed of totally optimal nodes. First, we find all totally optimal nodes in time $O(|V|^2)$. Then, starting from a totally optimal down-rotatable set, we expand the set with other totally optimal nodes as long as the resulting set is down-rotatable. This method can find many totally optimal down-rotatable sets efficiently.

This section has identified a set of nodes that provides subsets that can be rotated down to obtain alternative optimal DFGs. Such information may be used to refine a resource constrained schedule.

9.4 Minimal Optimal Transformation

Theorem 9.1 shows that a sequence of optimal down-rotations can reach any optimal DFG. However, the number of optimal down-rotatable sets might be large. This section introduces a restricted form of optimal rotations, called *minimal* rotations, and shows that a sequence of minimal rotations can also reach any optimal DFG through a sequence of optimal DFGs. First, we introduce a procedure to find an optimal down-rotatable set from any given set. This procedure is then used in a branch-and-bound algorithm to find all optimal DFGs.

For a down-rotatable set X of an optimal DFG G, the following procedure will find a down-rotatable set Y, where $X \subseteq Y$, to transform G into an optimal DFG G_Y.

Transformation 1 $X \Longrightarrow Y$, *where* $X \subseteq Y$.

Input : A down-rotatable set X of an optimal DFG G with cycle period $\hat{c}$.
Output: A down-rotatable set Y, $X \subseteq Y$, such that G_Y is optimal.

```
begin
    Queue ⟵ X;     Y ⟵ X;
    while Queue is not empty do begin
        Pop v from Queue
        for every u ∈ V − Y do
            if T(u, v) > ĉ and D(u, v) = 1 then begin
                Y(u) ⟵ 1;     Push u to Queue; end
    end
    Return Y.
end
```

In the process of this transformation, set Y may temporarily become not down-rotatable since some nodes haven't been examined. The following theorem shows that Y is down-rotatable after each so-called *phase*, and G_Y is optimal after the last phase.

Theorem 9.2 (Correctness of Transformation 1) *Let G be an optimal DFG with cycle period $\hat{c}$, and X a down-rotatable set. The set Y found by Transformation 1 is down-rotatable and optimal.*

Proof: We partition the transformation into phases: Phase one consists of the processing of nodes in the initial queue, i.e. X. For $j \geq 2$, phase j consists of the processing of nodes in the queue at the end of phase $j-1$. At the end of the last phase, there is no node in the queue. We first show that Y is down-rotatable after each phase, and then show that Y is optimal after the last phase.

Let Y_1 be the set Y obtained after the first phase. Since Y_1 is a superset of X, let Z be $Y_1 - X$, i.e. $Y_1 = X \cup Z$. We claim that Y_1 is down-rotatable if X is down-rotatable. Assume that the claim is true. The second phase starts with the set Z in the queue, and obtains a new set Y, denoted by Y_2, at the end of the phase. Suppose the phase is started with set $X \cup Z$ in the queue. For every $v \in X$ and $u \notin X \cup Z$, the previous phase has checked that the condition $T(u,v) > \hat{c}$ and $D(u,v) = 1$ is not satisfied. Since no node will be added to the queue due to the processing of nodes in X, the set Y obtained after this phase is still Y_2. Thus, we can say the second phase performs one-phase transformation on the down-rotatable set $X \cup Z$ and obtains Y_2. Since the claim is true, we know Y_2 is down-rotatable. With this argument, we can show that Y is down-rotatable after each phase.

We restate Property 8.1 as follows: X is down-rotatable if and only if

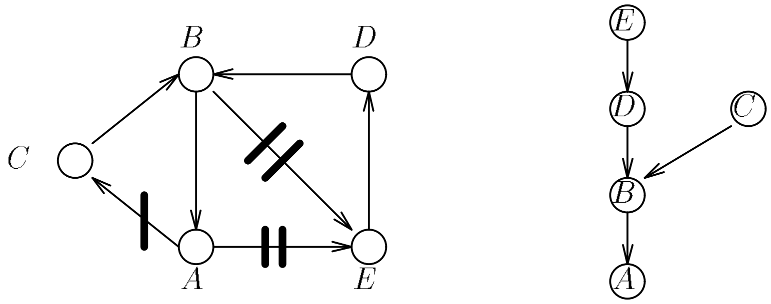
$$\forall v \in X, \forall u \notin X, D(u,v) \geq 1.$$

Assuming that X is down-rotatable, we want to prove that Y_1 $(= X \cup Z)$ is down-rotatable, that is

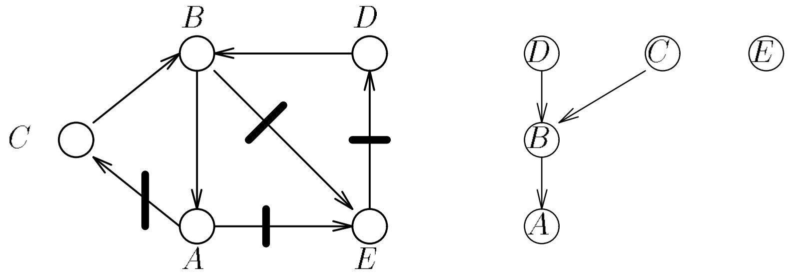
$$\forall v \in X, \forall u \notin X \cup Z, D(u,v) \geq 1, \text{ and}$$
$$\forall v \in Z, \forall u \notin X \cup Z, D(u,v) \geq 1.$$

Since X is down-rotatable, the first criterion is satisfied. For a node v in Z, we prove the second criterion by contradiction. Assume that there exists $u \notin X \cup Z$ such that $D(u,v) = 0$. Since Z consists of nodes added to the queue in the first phase, we can characterize Z as $Z = \{v \mid v \text{ is a node not in } X \text{ such that there exists a node } w \text{ in } X \text{ with } T(v,w) > \hat{c} \text{ and } D(v,w) = 1\}$. Since v belongs to Z, there exists a node w in X such that $T(v,w) > \hat{c}$ and $D(v,w) = 1$. Notice that $D(u,w) \leq D(u,v) + D(v,w) = 1$. Since X is down-rotatable and w belongs to X, we have $D(u,w) \geq 1$. Thus, $D(u,w) = 1$. Because of the relationship $D(u,w) = D(u,v) + D(v,w)$, we derive $T(u,w) \geq T(u,v) + T(v,w)$, which is larger than $\hat{c}$. Hence, u belongs to Z, which contradicts the assumption that u does not belong to $X \cup Z$. Thus, Y_1 is down-rotatable; the claim is proved.

Next, we show that the set Y obtained after the last phase is optimal. Since no node is added to the queue in the last phase, we have $T(u,v) \leq \hat{c}$ or $D(u,v) \neq 1$ for



(a) DFG $G_{\{\mathsf{A}\}}$ and the corresponding DAG



(b) DFG $G_{\{\mathsf{A},\mathsf{E}\}}$ and the corresponding DAG

Figure 9.6: The optimal transformation

every $u \notin Y$ and every $v \in Y$. Since G is optimal, we know $T(u,v) \leq \hat{c}$ or $D(u,v) \geq 1$ for every u and v in V. We derive $T(u,v) \leq \hat{c}$ or $D(u,v) \geq 2$ for every $u \notin Y$ and every $v \in Y$. $\square$

It takes $O(|V|^2)$ time to find the down-rotatable set Y. For example, let $X = \{\mathsf{A}\}$. The DFG G_X in Figure 9.6-(a) is not optimal because the longest path of its DAG has length 4. However, the above algorithm will find the set $Y = \{\mathsf{A}, \mathsf{E}\}$ such that G_Y in Figure 9.6-(b) is optimal.

The procedure will return X if X is already an optimal down-rotatable set. In this case, an optimal transformation from G to G_X is obtained. So, the above procedure can also serve as a checking procedure to decide whether a set X is optimal. The transformation goes back to G if the vertex set V is returned.

Since only the DAG structure of a DFG will affect the resource-constrained scheduling, we consider two optimal DFGs with the same DAG structure as *equivalent*. The following branch-and-bound algorithm can find all optimal DFGs with different DAG structures from a given optimal DFG by using *minimal* down-rotations only. A down-rotatable set X is *minimal* if X is optimal down-rotatable, but any subset of X is not. Any set containing only a partial subset of a minimal down-rotatable set X, not the entire X, is not optimal down-rotatable.

Algorithm 1 *Branch-and-Bound*

Input : An optimal DFG G.
Output: Down-rotatable sets X such that DFG G_X are optimal.
begin
    for every $v \in V$ such that $\{v\}$ is down-rotatable **do begin**
        $Y \longleftarrow$ Transformation 1 ($\{v\}$);
        Output G_Y;
        if not Bound(G_Y) **then** /* If the DAG of G_Y has not been generated */
            Branch-and-Bound(G_Y);
    end
end

In order to avoid generating optimal DFGs with different delay configurations but the same DAG, the algorithm will stop at a branch when the corresponding DAG has been generated. Since rotations only affect delay configurations, there is a finite number of different DAGs. Although the above algorithm only uses minimal rotations, the algorithm can reach all optimal DFGs due to the following lemma.

Lemma 9.4.1 *Let G and G_X be optimal DFGs, and X an optimal down-rotatable set. There exists a sequence of minimal down-rotations to transform G to G_X.*

Proof: We construct the sequence of minimal rotations as follows. Pick any node v from X. Find the minimal rotatable set containing v, denoted by Y. If $Y \not\subseteq X$ but $v \in X$, X is not down-rotatable, which contradicts the assumption. Thus, we have $Y \subseteq X$, and $X - Y$ is down-rotatable with respect to G_Y. Then, continue the process on the set $X - Y$ until $X - Y$ is empty. $\Box$

Theorem 9.3 *Algorithm 1 can find all different DAG structures corresponding to* *optimal DFGs.*

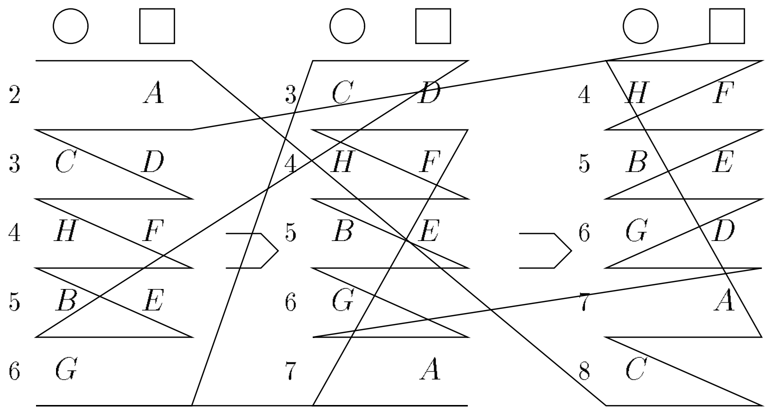
Proof: Algorithm 1 branchs by performing Transformation 1 on all down-rotatable singleton sets. If this branching strategy can produce all minimal rotations of G, from Lemma 9.4.1 and Theorem 9.1, the algorithm can reach all optimal DFGs eventually; thus all different DAG structures corresponding to optimal DFGs. Consider any minimal down-rotatable set X. We show that X will be produced by the branching strategy. Pick any source node v from the DAG of X. Since $\{v\}$ is a down-rotatable singleton set, Transformation 1 is performed on $\{v\}$ to obtain Y, which is also a minimal down-rotatable set.

We claim that there is a unique minimal down-rotatable set containing node v. If the claim is true, X equals Y and the theorem is proved. To prove by contradiction, suppose there are two distinct minimal down-rotatable sets X and Y both containing v. We will derive that $X \cap Y$ is optimal down-rotatable; thus, this contradicts the assumption. Since X and Y are down-rotatable, for every $v \in X$ and every $u \notin X$ we have $D(u,v) \geq 1$, and for every $v \in Y$ and every $u \notin Y$ we have $D(u,v) \geq 1$. Thus, for every $v \in X \cap Y$ and every $u \notin X \cap Y$ we have $D(u,v) \geq 1$, i.e. $X \cap Y$ is down-rotatable. By similar argument, since X and Y are optimal, we can derive $X \cap Y$ is optimal. □

9.5 The Combination of Optimal Transformation and Rotation Scheduling

The optimal retiming found by the retiming algorithm in [53] may not give a satisfactory schedule, because delays in the non-critical parts of the DFG are placed arbitrarily as long as the lengths of paths without delays are shorter than the cycle period. In fact, many critical paths in the non-critical part of the DFG are not necessary, and we should try to use another optimal retiming function to improve such situations.

For example, the DFG in Figure 9.1-(a) is optimal, and the schedule of length 6 in Figure 9.1-(b) is one of the shortest for this optimal DFG. In this section, we will



(a) Rotating H E (b) Rotating A (c) Rotating C D

Figure 9.7: Rotations of three control steps

improve the schedule for that DFG from length 6 to length 4, as shown in Figure 9.2, which is obviously the shortest schedule.

In order to explore more retiming functions, we combine the optimal rotations and resource-constrained rotation scheduling from Chapter 8. The combined approach is shown in Figure 9.8. In the outer loop, we find an optimal retiming for the DFG, schedule the retimed DFG; in the inner loop we compact the DFG by resource-constrained rotations. The inner loop is repeated until a rotated node is going to be rotated again. Therefore, in the compaction process, every node is rotated down at most once. Then, the transformation in Section 9.2 can be applied to find another optimal retiming for the rotated DFG.

For the DFG in Figure 9.1, the compaction process continues for three control steps, as shown in Figure 9.7. Because H is already rotated once, the compaction process stops. The set $X = \{\mathsf{H}, \mathsf{E}, \mathsf{A}, \mathsf{C}, \mathsf{D}\}$ is rotated down, but the corresponding DFG G_X is not optimal, as shown in Figure 9.9. By using Transformation 1, we obtain $Y = X \cup \{\mathsf{F}\}$ such that G_Y is optimal, as shown in Figure 9.2-(a). For DFG G_Y, a simple list scheduling can find the optimal schedule of length 4 in Figure 9.2-(b).

Actually, for DFG G_Y, at least three optimal schedules of length 4 can be found with different heuristics for the list scheduling algorithm. Notice that the DAG of G_Y has only one critical path, while the DAG of the original DFG G has four critical

Algorithm 2 *The Combined Approach*

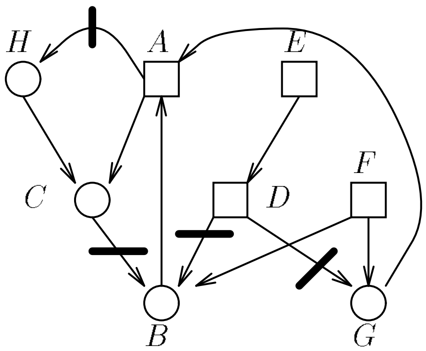
Parameters : cnt : the number of schedules to be found.
i : size of resource-constrained rotations.

begin

Find an optimal retiming r for the DFG G by the retiming algorithm in [53].
for $j = 1$ **to** cnt **do begin** /* The outer loop */
$G \longleftarrow G_r$;
S $\longleftarrow$ FullSchedule(G); /* S is as defined in Section 8.3 */
$R \longleftarrow \emptyset$; /* The set of rotated nodes */
repeat /* The resource-constrained rotations */
$(S, L, X) \longleftarrow$ DownRotate(G_R, S, i);
$R \longleftarrow R \circ X$;
until $R \cap \{v \mid 1 \leq S(v) \leq i\} \neq \emptyset$;
Find another optimal retiming r by the optimal transformations
from the down-rotatable set R;
end
Choose the best schedule found among all iterations.

end

Figure 9.8: The combined approach



DFG G_X with $X = \{$H, E, A, C, D$\}$.

Figure 9.9: The non-optimal DFG after the compaction process stops

paths, and 6 out of 8 nodes belong to critical paths. Intuitively, a DAG with many short paths will give us more freedom in the process of resource scheduling. The optimal DFG such that delays are evenly distributed in every part of the DFG tends to have a better schedule. Therefore, not all optimal DFGs are equally qualified for resource scheduling; some DFGs tend to provide many good schedules under resource constraints.

9.6 Conclusion

This chapter has formed a theoretical foundations for the rotation technique. It is easy to see that a sequence of rotations can represent all retimings. Moreover, we showed that a sequence of *optimal* rotations can represent any *optimal* retiming. Hence, although rotation is a more restrictive form of retiming, a sequence of rotations has the same expressive power as retiming, and rotation is easier to manipulate.

There are usually many optimal retimings for a DFG. As we observed in the last section, we found that some retimings are better than others regardless of the resource requirement; some are better under a specific resource constraint. However, some retimings do not give good structures for the non-critical part of the DFGs, which will affect the schedule length under resource constraints. We would like to study the characteristics of *good* retimings under resource constraints. For example, a retiming with a smaller number of critical paths in the DAG is good; a retiming such that nodes competing for the same resource are spread out evenly is good.

In Chapter 8, the second heuristic algorithm uses rotation functions as the initial retiming functions to initialize a phase. The performance tends to be better than the first heuristic, which starts from the same initial retiming. If we start from a *good* optimal retiming, the number of phases or rotations needed to reach optimal schedules is likely to be reduced. Based on the optimal retiming transformation technique, we would like to study the effect of retiming on the performance of rotations, and the criteria of *good* optimal retimings.

Only down rotations are used in our heuristic algorithms. There are plenty of possibilities of combining up-rotations and down-rotations to improve schedules. Also, the nodes can be rotated up and down independent, without rotating entire control steps. We have more flexibility in exploring scheduling under such *unconstrained rotations*. One possibility is to design rotation scheduling algorithms subject to a given schedule length.

Chapter 10

Conclusions and Future Work

Retiming and unfolding have been the main theme of the thesis. We have studied some interesting graph properties of these two techniques. The next section will discuss those problems still open. *Synchronous data-flow graphs* have been used to describe large-grain tasks. Because a synchronous data-flow graphs can be transformed into an equivalent regular data-flow graph [48], we believe much of our techniques can be extended to that area. Section 10.2 discusses related issues. *Clock skew* is also used to improve the clock cycle period of a sequential circuit with flip-flops [24]. Chapter 5 introduced the notion of scheduling on a fractional timing model, which corresponds to a fractional retiming. Clock skew and fractional retiming seems to have a close relationship, as discussed in Section 10.3. Other extensions to rotation scheduling are described in the last section.

10.1 Retiming and Unfolding

In this thesis we study some fundamental theorems about the combination of two useful techniques: retiming and unfolding. This understanding gives us more insight for many problems in which retiming and unfolding can both be applied such as multiprocessor scheduling. It is important to know how to retime and unfold the original DFG such that the unfolded retimed DFG can be scheduled satisfactorily. We believe that our results can also be applied to other fields.

One interesting result shows that unfolding before retiming or after retiming has no effect on the iteration period. Therefore, we do not need to unfold a DFG first to obtain results; instead we can do operations directly on the original DFG. For multi-dimensional DFGs, it is still open whether the order of retiming and unfolding is immaterial.

When we consider the unit-time DFG, which is applicable to RISC multiprocessors, software pipelining, unit-time parallel pipelines, etc., very good results are obtained. We derived the *minimum* rate-optimal unfolding factor to be the denominator of the iteration bound in its irreducible form. For a given unfolding factor f and cycle period c, the checking of whether there is a legal retiming is very simple: one only needs to check whether or not $c/f \geq B(G)$. A simple $O(|V||E|)$-time retiming algorithm is presented. The similar concepts are used to find pipeline schedules for general-time DFGs under various models. For each model, the minimum rate-optimal unfolding factor and the minimum cycle period for a given unfolding factor are obtained.

We also presented retiming algorithms for general-time DFGs. However, the minimum cycle period has to be found by binary search after a time-consuming preprocessing step. We believe that the time-complexity of the preprocessing step can be further improved. For rate-optimal unfolding factors, to the author's knowledge, there is not even a polynomial-time algorithm to compute them. The schedules produced by retiming and unfolding general-time DFGs have the property that the execution periods of nodes do not span over the boundaries of the repeating pattern. This schedules are natural for hardware pipelines. The necessity and applicability of this type of schedules for multi-processor systems need to be studied.

After understanding these fundamental properties, we have successfully applied the concepts to scheduling problems with resource constraints by using retiming. One important open question is to bound the minimum iteration period for an unfolding factor under resource constraints by that without resource constraints. This may lead to polynomial-time approximation algorithms for multiprocessor scheduling for iterative algorithms.

10.2 Task-Level Data Flow Graphs

Data-flow graphs not only describe the dependencies between different parts of the computation required in an application, but also provide built-in scheduling and synchronization. Due to the generality of this representation, it can be used to specify parallelism at the task level [16, 72] as well as at the instruction level [26]. Large-grain tasks are usually described by *synchronous data flow graphs* [49].

In traditional data-flow graphs, an execution of a node takes one data item from each of the input edges and produces one data item for each of the outgoing edges. Synchronous data flow graphs differ from traditional data flow graphs in that the amount of data produced and consumed for an execution of a data flow node is specified for each input edge and output edge. Therefore, the execution rates of data-flow nodes are different. Static scheduling algorithms are presented for the scheduling of synchronous data flow graphs in [48]. Some theoretical properties are discussed in [47].

In real-time environments, the ability to predict a program's performance is critical. The operations are usually dispatched into ready processor elements when all input data are ready, which leads to an irregular execution pattern. With periodic data arrival, the irregularity arises due to the absence of direct control over execution of nodes that depend only upon the receipt of data from the external world. Therefore, the concept of controlled data flow execution has been addressed in [73, 74]. Static scheduling is used to form predictable execution patterns to increase the predictability of system performance.

In order to find good static schedules, the techniques of pipelining and unfolding need to be considered. Reference [62] proposed a very restricted form of retiming for synchronous data flow graphs. We believe a form of retiming on synchronous data flow graphs can be derived so that it is general enough to represent all pipelines. Also, the relationship between retiming and unfolding on synchronous data flow graphs needs to be investigated.

The static scheduling of a task-level data flow graph is based on an estimation of the computation time of each task. The actual computation time of each task may vary over time. In order to reduce synchronization overhead, the multi-processor system will not be tightly synchronized according to the static schedule determined at compile time. Mechanisms are required to loosely synchronize a static schedule, and to readjust a static schedule according to the actual computation times of tasks or new estimates. Run-time algorithms are especially needed to incrementally readjust a static schedule according to the new estimation of computation times.

10.3 Clock Skew

Synchronous circuit designers ordinarily try to eliminate clock skew, which may be defined as variation in the delays from the central clock structure to the flip-flops of the system. Clock skew can limit the clocking rate of a synchronous system or cause malfunction at certain clock rates. However, the performance of a synchronous system can also be improved by adjusting the clock skews of individual flip-flops [24].

It is observed that adding clock skew to a flip-flop has an effect similar to movement of the flip-flop backwards across combinational logic boundaries. Therefore, clock skew and retiming are continuous and discrete optimization with the same effect.

To optimize a system which is not well balanced, a huge number of clock skews are applied for many of its flip-flops. The extra circuits needed for these skews are impractical. However, retiming has its disadvantages and restrictions, too. For a node with large fanout but little fanin, the movement of a flip-flop from its inputs to its output will have an undesirable increase in the number of flip-flops. Flip-flops corresponding to primary inputs of an entire system should not be removed. Flip-flops integrated with off-the-shelf parts cannot be retimed. Therefore, methodologies are needed to optimize a synchronous system by using retiming where it is applicable and adding clock skew for minor adjustments.

Chapter 5 proposed the concept of integral and fractional timing models. Subsection 2.6 discussed the effect of node splitting for general-time data flow graphs. We know a repeating schedule under the integral timing model is a static schedule of a retimed graph where nodes can be split into subnodes of integral size. In fact, a repeating schedule under the fractional timing model is a static schedule of a fractionally retimed graph, where nodes can be split into subnodes of fractional size. Therefore, the concept of fractional retiming has the same continuous effect as clock skew. In order to combine the use of retiming and clock skew, the actual relationship between them needs to be studied.

10.4 Rotation Scheduling

The current rotation scheduling algorithm only deals with single-level loops without conditional statements. Also, we do not assume the computations of nodes have any special properties like associativity and distributivity. The following extensions should be investigated.

Nested loops For a nested loop, rotation scheduling can pipeline the innermost loop first. After the inner loop is pipelined, we can treat it as a multi-cycle operation while we pipeline the outer loop. The rotation technique needs to be modified to handle multi-cycle operations that have different resource requirements in each cycle. The effectiveness of this approach also needs to be studied with benchmarks.

Conditionals If the DFG has mutual exclusive operations, we can check the conditions while assigning an operation to a control step. If there are conditional data flows, as in most imperative languages, we will need to design a conditional data flow model to capture the situation.

Other transformations There are other data-flow-transformation techniques that can improve schedule lengths beyond some of the structure-related theoretical bounds we derive, for example, redundant operation creation [54], and algebraic transformation [58, 69]. The applicability of these transformations in conjunction with rotation needs to be investigated.

Bibliography

[1] Aiken, A., and Nicolau, A. Loop quantization: an analysis and algorithm. Technical Report 87-821, Department of Computer Science, Cornell University, Mar. 1987.

[2] Aiken, A., and Nicolau, A. Optimal loop parallelization. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation* (June 1988), pp. 308–317.

[3] Baker, K. R. *Introduction to Sequencing and Scheduling.* Wiley, New York, 1974.

[4] Camposano, R. Path-based scheduling for synthesis. *IEEE Transactions on Computer-Aided Design 10*, 1 (1991), 85–93.

[5] Camposano, R., and Rosenstiel, W. Synthesizing circuits from behavioral descriptions. *IEEE Transactions on Computer-Aided Design 8*, 2 (Feb. 1989), 171–180.

[6] Camposano, R., Saunders, L. F., and Tabet, R. M. High-level synthesis from vhdl. *IEEE Design and Test of Computers* (Mar. 1991).

[7] Chao, L.-F., LaPaugh, A., and Sha, E. H.-M. Scheduling cyclic data-flow graphs by retiming with resource constraints. In *Proceedings of the ACM/IEEE International Workshop on High-Level Synthesis* (Dana Point, California, Nov. 1992), pp. 111–134.

[8] Chao, L.-F., LaPaugh, A., and Sha, E. H.-M. Rotation scheduling: A loop pipelining algorithm. In *Proceedings of the ACM/IEEE Design Automation Conference* (June 1993).

[9] Chao, L.-F., and Sha, E. H.-M. Retiming and unfolding data-flow graphs. In *Proceedings of the International Conference on Parallel Processing* (St. Charles, Illinois, Aug. 1992), pp. II 33–40.

[10] Chao, L.-F., and Sha, E. H.-M. Unfolding and retiming data-flow DSP programs for RISC multiprocessor scheduling. In *Proceedings of the IEEE International Conference on Acoustic, Speech, and Signal Processing* (San Francisco, California, Mar. 1992), pp. V565–V568.

[11] Chao, L.-F., and Sha, E. H.-M. Static schedulings of uniform nested loops. In *Proceedings of the International Parallel Processing Symposium* (Apr. 1993), pp. 254–258.

[12] Chu, C.-M., Potkonjak, M., Thaler, M., and Rabaey, J. HYPER : an interactive synthesis environment for high performance real time applications. In *Proceedings of the International Conference on Computer Design* (1989), pp. 432–435.

[13] Cohen, E., and Megiddo, N. Strongly polynomial-time and nc algorithms for detecting cycles in dynamic graphs. In *Annual ACM Symposium on Theory of Computing* (1989), pp. 523–534.

[14] Cohen, E., and Megiddo, N. Recognizing properties of periodic graphs. In *Applied Geometry and Discrete Mathematics: The Victor Klee Festschrift*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 1991, pp. 135–146.

[15] Conway, R. W., Maxwell, W. L., and Miller, L. W. *Theory of Scheduling.* Addison-Wesley, Reading, MA, 1967.

[16] Curtis, B. A., and Madisetti, V. K. Task scheduling in the Georgia Tech digital signal processors. In *Proceedings of the IEEE International Conference on Acoustic, Speech, and Signal Processing* (1992), pp. V 589–592.

[17] Cytron, R. Doacross: beyond vectorization for multiprocessors (extended abstract). In *Proceedings of the International Conference on Parallel Processing* (1986), pp. 836–844.

[18] Davidson, S., Landskov, D., Shriver, B. D., and Mallett, P. W. Some experiments in local microcode compaction for horizontal machines. *IEEE Transactions on Computers C-30*, 7 (1981), 460–477.

[19] DAVIS, A. L., AND KELLER, R. M. Date flow program graphs. *IEEE Computer 15*, 2 (Feb. 1982), 26–41.

[20] DE MICHELI, G., KU, D., MAILHOT, F., AND TRUONG, T. The Olympus synthesis system. *IEEE Design and Test of Computers* (Oct. 1990).

[21] DEJONG, G. Data flow graphs: system specification with the most unrestricted semantics. In *Proceedings of the European Design Automation Conference* (1991).

[22] DEY, S., POTKONJAK, M., , AND ROTHWEILER, S. G. Performance optimization of sequential circuits by eliminating retiming bottlenecks. In *Proceedings of the International Conference on Computer Aided Design* (Nov. 1992), pp. 504–509.

[23] DUNCAN, P., SWAMY, S., SPROUSE, S., POTASZ, D., JAIN, R., CAMMACK, W., GAFTER, N., WONG, Y., AND GASS, W. Experiments with the Hi-PASS DSP synthesis system. In *Proceedings of the International Symposium on Circuits and Systems* (Mar. 1992), pp. 168–171.

[24] FISHBURN, J. P. Clock skew optimization. *IEEE Transactions on Computers 39*, 7 (July 1990), 945–951.

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

[26] FISHER, J. A., AND RAU, B. R. Instruction-level parallel processing. *Science 253* (Sept. 1991), 1233–1241.

[27] GAJSKI, D. D., DUTT, N. D., AND PANGRLE, B. M. Silicon compilation (tutorial). In *Proceedings of the IEEE Custom Integrated Circuits Conference* (May 1986), pp. 102–110.

[28] GAO, G. R., WONG, Y.-B., AND NING, Q. A timed petri-net model for fine-grain loop scheduling. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation* (June 1991), pp. 204–218.

[29] GAREY, M. R., AND JOHNSON, D. S. *Computers and Intractability: a Guide to the Theory of NP-Completeness*. W. H. Freeman, San Francisco, 1979.

[30] GIRCZYC, E. M. Loop winding – a data flow approach to functional pipelining. In *Proceedings of the International Symposium on Circuits and Systems* (May 1987), pp. 382–385.

[31] GOOSSENS, G., VANDEWALLE, J., AND DE MAN, H. Loop optimization in register-transfer scheduling for DSP-systems. In *Proceedings of the ACM/IEEE Design Automation Conference* (1989), pp. 826–831.

[32] HITCHCOCK, C. Y., AND THOMAS, D. E. A method of automatic data path synthesis. In *Proceedings of the ACM/IEEE Design Automation Conference* (July 1983), pp. 484–489.

[33] HWANG, C.-T., LEE, J.-H., AND HSU, Y.-C. A formal approach to the scheduling problem in high level synthesis. *IEEE Transactions on Computers* (1991), 464–475.

[34] IWANO, K. *Two-Dimensional Dynamic Graphs and their VLSI Applications.* PhD thesis, Department of Computer Science, Princeton University, 1987.

[35] IWANO, K., AND STEIGLITZ, K. Testing for cycles in infinite graphs with periodic structure. In *Annual ACM Symposium on Theory of Computing* (May 1987), pp. 46–55.

[36] IWANO, K., AND YEH, S. An efficient algorithm for optimal loop parallelization. In *Proceedings of The First International Symposium of Algorithms* (Dec. 1990).

[37] JAIN, R., MUJUMDAR, A., SHARMA, A., AND WANG, H. Empirical evaluation of some high-level synthesis scheduling heuristics. In *Proceedings of the ACM/IEEE Design Automation Conference* (1991), pp. 210–215.

[38] JENG, L.-G., AND CHEN, L.-G. A globally static rate optimal scheduling for recursive DSP algorithms. In *Proceedings of the IEEE International Conference on Acoustic, Speech, and Signal Processing* (May 1991), pp. 1005–1008.

[39] JENG, L.-G., AND CHEN, L.-G. Synthesis of rate-optimal dsp algorithms by pipeline and minimum unfolding. In *Proceedings of the ACM/IEEE International Workshop on High-Level Synthesis* (Nov. 1992), pp. 159–168.

[40] JOHNSON, D. B. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM 24*, 1 (Jan. 1977), 1–13.

[41] Karp, R. M. A characterization of the minimum cycle mean in a digraph. *Discrete Mathematics 23* (1978), 309–311.

[42] Kosaraju, S. R., and Sullivan, G. F. Detecting cycles in dynamic graphs in polynomial time. In *Annual ACM Symposium on Theory of Computing* (1988), pp. 398–406.

[43] Kung, S. *VLSI Array Processors*. Prentice Hall, 1988.

[44] Kung, S. Y., Whitehouse, H. J., and Kailath, T. *VLSI and Modern Signal Processing*. Information and Systems Sciences Series. Prentice Hall, 1985, pp. 258–264.

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

[46] Lawler, E. L. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart and Winston, New York, 1976.

[47] Lee, E. A. Consistency in dataflow graphs. *IEEE Transactions on Parallel and Distributed Systems 2*, 2 (Apr. 1991), 223–235.

[48] Lee, E. A., and Messerschmitt, D. G. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers 36*, 1 (Jan. 1987), 24–35.

[49] Lee, E. A., and Messerschmitt, D. G. Synchronous data flow. *Proceedings of the IEEE 75*, 9 (Sept. 1987), 1235–1245.

[50] Lee, T.-F., Wu, A. C.-H., Gajski, D. D., and Lin, Y.-L. An effective methodology for functional pipelining. In *Proceedings of the International Conference on Computer Aided Design* (Dec. 1992), pp. 230–233.

[51] Leiserson, C. E., Rose, F. M., and Saxe, J. B. Optimizing synchronous circuitry by retiming. In *Proceedings of the 3rd Caltech Conference on Very Large Scale Integration* (1983), pp. 87–116.

[52] Leiserson, C. E., and Saxe, J. B. Optimizing synchronous systems. *Journal of VLSI and Computer Systems 1*, 1 (Spring 1983), 41–67.

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

[54] Lobo, D., and Pangrle, B. Redundant operator creation: a scheduling optimization technique. In *Proceedings of the ACM/IEEE Design Automation Conference* (1991), pp. 775–778.

[55] Malik, S., Sentovich, E. M., et al. Retiming and resynthesis: optimizing sequential networks with combinational techniques. *IEEE Transactions on Computer-Aided Design 10*, 1 (Jan. 1991), 74–84.

[56] McFarland, M., Parker, A. C., and Camposano, R. The high-level synthesis of digital systems. *Proceedings of the IEEE 78* (Feb. 1990), 301–318.

[57] Nestor, J., and Thomas, D. E. Behavioral synthesis with interfaces. In *Proceedings of the International Conference on Computer Aided Design* (Nov. 1986), pp. 112–115.

[58] Nicolau, A., and Potasman, R. Incremental tree height reduction for high level synthesis. In *Proceedings of the ACM/IEEE Design Automation Conference* (1991), pp. 770–774.

[59] Orailogulu, A., and Gajski, D. D. Flow graph representation. In *Proceedings of the ACM/IEEE Design Automation Conference* (1986), pp. 503–509.

[60] Orline, J. B. Some problems on dynamic/periodic graphs. In *Progress in Combinatorial Optimization*, W. R. Pulleyblank, Ed. Academic Press, Orlando, Florida, 1984, pp. 273–293.

[61] Pangrle, B. M., and Gajski, D. D. Slicer: a state synthesizer for intelligent silicon compilation. In *Proceedings of the International Conference on Computer Aided Design* (Oct. 1987).

[62] Parhi, K. K. Algorithm transformation techniques for concurrent processors. *Proceedings of the IEEE 77*, 12 (Dec. 1989), 1879–1895.

[63] Parhi, K. K., and Messerschmitt, D. G. Rate-optimal fully-static multiprocessor scheduling of data-flow signal processing programs. In *Proceedings of the International Symposium on Circuits and Systems* (May 1989), pp. 1923–1928.

[64] PARHI, K. K., AND MESSERSCHMITT, D. G. Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding. In *Proceedings of the International Conference on Parallel Processing* (1989), pp. I 209–216.

[65] PARHI, K. K., AND MESSERSCHMITT, D. G. Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding. *IEEE Transactions on Computers 40*, 2 (Feb. 1991), 178–195.

[66] PARK, N., AND PARKER, A. C. Sehwa: A software package for synthesis of pipelines from behavioral specifications. *IEEE Transactions on Computer-Aided Design 7*, 3 (Mar. 1988), 356–370.

[67] PAULIN, P., AND KNIGHT, J. P. Force-directed scheduling for the behavioral synthesis of ASIC's. *IEEE Transactions on Computer-Aided Design 8*, 6 (June 1989), 661–679.

[68] POTASMAN, R., LIS, J., NICOLAU, A., AND GAJSKI, D. Percolation-based scheduling. In *Proceedings of the ACM/IEEE Design Automation Conference* (1990), pp. 444–449.

[69] POTKONJAK, M. *Algorithms for high level synthesis: resource utilization based approach*. PhD thesis, U.C. Berkeley, 1991.

[70] RAJ, V. K. DAGAR: An automatic pipelined microarchitecture synthesis system. In *Proceedings of the International Conference on Computer Design* (1989), pp. 428–431.

[71] RENFORS, M., AND NEUVO, Y. The maximum sampling rate of digital filters under hardware speed constraints. *IEEE Transactions on Circuits and Systems CAS-28*, 3 (Mar. 1981), 196–202.

[72] SHUKLA, S., LITTLE, B., AND ZAKY, A. A compile-time technique for controlling real-time execution of task-level data-flow graphs. In *Proceedings of the International Conference on Parallel Processing* (Aug. 1992), Naval Postgraduate School, pp. II 49–56.

[73] SHUKLA, S. B., AND AGRAWAL, D. P. Scheduling pipelined communication in distributed memory multiprocessors for real-time applications. In *Proceedings of the International Symposium on Computer Architecture* (May 1991).

[74] SOM, S., MIELKE, R. R., AND STOUGHTON, J. W. Strategies for predictability in real-time data-flow architectures. In *Real-Time Systems Symposium* (1990), pp. 226–237.

[75] TARJAN, R. E. *Data Structures and Network Algorithms.* SIAM, Philadelphia, Pennsylvania, 1983.

[76] TSENG, C., AND SIEWIOREK, D. P. Automated synthesis of data paths in digital systems. *IEEE Transactions on Computer-Aided Design CAD-5*, 3 (July 1986), 379–395.

[77] WANG, C.-Y., AND PARHI, K. K. High level DSP synthesis using the MARS design system. In *Proceedings of the International Symposium on Circuits and Systems* (1992), pp. 164–167.

[78] WANG, D. J., AND HU, Y. H. Optimal scheduling of linear recurrence equations on a multi-processor array. In *Proceedings of the IEEE International Conference on Acoustic, Speech, and Signal Processing* (Toronto, May 1991), pp. 1581–1584.

[79] WANG, D. J., AND HU, Y. H. Fully static multiprocessor realization for real-time recursive DSP algorithms. In *International Conference on Application-Specific Array Processors* (Aug. 1992), pp. 664–678.

[80] WOLFE, M. Loop skewing: the wavefront method revisited. *International Journal of Parallel Programming 15*, 4 (Aug. 1986), 284–294.

[81] ZAKY, A., AND SADAYAPPAN, P. Optimal static scheduling of sequential loops on multiprocessors. In *Proceedings of the International Conference on Parallel Processing* (1989), pp. III 130–137.