2003

On implementing dynamically reconfigurable architectures

Hatem Mahmoud El-Sayed El-Boghdadi
Louisiana State University and Agricultural and Mechanical College

ON IMPLEMENTING DYNAMICALLY RECONFIGURABLE ARCHITECTURES

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Electrical and Computer Engineering

by
Hatem Mahmoud El-Sayed El-Boghdadi
B.Sc., Assiut University, Egypt, 1991
M.Sc., Assiut University, Egypt, 1994
May 2003

To My Parents & Brothers

Acknowledgments

I would like to thank the members of my committee Dr. S. Rai, Dr. J. Trahan, Dr. J. Ramanujam, Dr. S. Kundu, and Dr. R. Litherland.

I would like to express my gratitude to my advisor Dr. R. Vaidyanathan for his guidance, and technical advice during the course of this work. The initial ideas for the E-SRGA and bends-cost LR-Mesh grew out of many meetings where we tried to reconcile the benefits provided by both FPGAs and R-Mesh model.

I would like also to thank Dr. M. Abdelrahman who helped me to come to the United States.

Finally, I would like to express my gratitude to my parents and brothers in Egypt. Their unwavering support has made this work eventually possible. It is to them that I dedicate the dissertation.

Table of Contents

List of Tables

List of Figures

Abstract

Dynamically reconfigurable architectures have the ability to change their structure at each step of a computation. This dissertation studies various aspects of implementing dynamic reconfiguration, ranging from hardware building blocks and low-level architectures to modeling issues and high-level algorithm design.

First we derive conditions under which classes of communication sets can be optimally scheduled on the circuit-switched tree (CST). Then we present a method to configure the CST to perform in constant time all communications scheduled for a step. This results in a constant time implementation of a step of a segmentable bus, a fundamental dynamically reconfigurable structure.

We introduce a new bus delay measure (bends-cost) and define the bends-cost LR-Mesh; the LR-Mesh is a widely used reconfigurable model. Unlike the (idealized) LR-Mesh, which ignores bus delay, the bends-cost LR-Mesh uses the number of bends in a bus to estimate its delay. We present an implementation for which the bends-cost is an accurate estimate of the actual delay. We present algorithms to simulate various LR-Mesh configuration classes on the bends-cost LR-Mesh. For "semimonotonic configurations," a $\Theta(N) \times \Theta(N)$ bends-cost LR-Mesh with bus delay at most D can simulate a step of the idealized $N \times N$ LR-Mesh in $O\left(\left(\frac{\log N}{\log D - \log \Delta}\right)^2\right)$ time (where Δ is the delay of an N-element segmentable bus), while employing about the same number of processors. For some special cases this time reduces to $O\left(\frac{\log N}{\log D - \log \Delta}\right)$. If $D = N^\epsilon$, for an arbitrarily small constant $\epsilon > 0$, then the running times of bends-cost LR-Mesh algorithms are within a constant of their idealized counterparts. We also prove that with a polynomial blowup in the number of processors and $D = N^\epsilon$, the bends-cost LR-Mesh can simulate any step of an idealized LR-Mesh in constant time, thereby establishing that these models have the same "power."

We present an implementation (in VHDL) of the "Enhanced Self Reconfigurable Gate Array" (E-SRGA) architecture and perform a cost-benefit study for different dynamic reconfiguration features. This study shows our approach to be feasible.

Chapter 1

Introduction

Advances in technology and the need for more powerful and faster devices have produced a range of computing devices varying in their efficiency and their flexibility. At one extreme are Application Specific Integrated Circuits (ASICs) that are narrowly tailored to solve a small suite of problems. At the other end of the spectrum are programmable processors that can be programmed to solve any solvable problem. This dissertation focuses on devices and models that occupy a middle ground that deals with reconfigurable computing [13].

ASICs are devices that have dedicated hardware designed for one specific task. The function of this hardware is fixed at the time of fabrication. Thus, one can expect such devices to be fast and have an efficient use of chip area and power. Once manufactured, however, these devices can only perform the tasks for which they are designed. Thus, they lack flexibility.

On the other hand, programmable processors are devices that can execute a number of different functions. The user can program these devices, after fabrication, to perform any desired task. Therefore, such devices are very flexible but only at the expense of efficiency.

Reconfigurable computing started as a new method that promised speeds not possible on traditional models of computation. This initial thrust was centered around models such as the R-Mesh that used dynamic reconfiguration, the ability to change the structure of the architecture very rapidly, possibly at each step.

Subsequently, reconfiguration moved to Field Programmable Gate Arrays (FPGAs), devices that can be configured as circuits suited to the problem at hand. Run-

time reconfiguration (RTR) deals with fast reconfiguration on FPGAs. Although RTR is not as fast as dynamic reconfiguration, it allows the device to configure at run time to suit the problem at hand.

This dissertation deals with implementing Dynamic Reconfiguration using ideas from both the R-Mesh model and FPGA-type platforms.

1.1 The State of the Art

Dynamic reconfiguration has been shown to be a very powerful computing paradigm, capable of extremely fast solutions to many problems. Models such as the R-Mesh [32] have been extensively studied and solutions developed for a wide range of problems. Nakano [34] provides an extensive bibliography of results in computing with dynamic reconfiguration. Although models such as the R-Mesh provide an abstract platform to develop reconfigurable algorithms, they are difficult to implement. This is due to the fact that most algorithms employ buses whose delay is proportional to the problem size. On such buses, the constant bus-delay assumption that is central to all R-Mesh algorithms does not hold.

Compared to the volume of results published for reconfigurable models, relatively little work has been reported on implementing these models or specific algorithms for them. One direction has been algorithmic, using an R-Mesh with restricted bus lengths (that restrict the delay). Beresford-Smith *et al.* [12] developed a sorting algorithm that runs on an $N \times N$ R-Mesh with bus delay bounded by D. This algorithm incurs a $\Theta\left(\frac{N}{D}\right)$ overhead in time. Kunde and Gurtzig [23] designed an $hN + o(hdN)$ time algorithm for h-h sorting and routing problems on a d-dimensional R-Mesh of side length N using constant delay buses. The fact that this algorithm runs at the same speed as the one using buses spanning more processors is due to the properties of the problem and its solution, and is not a general technique for implementing buses spanning a large number of processors. Bertossi and Mei [6] showed that the simulation of the Basic R-Mesh (a very restricted version of the LR-Mesh) reduces to the segmented scan problem and proved that this problem can be solved on fixed connection networks. Other approaches [3, 17] tried to scale down the size of the

R-Mesh enabling the algorithms to run on smaller size array. These approaches simulate the large size model on the smaller size model and hinge on computing the connected components for the simulated array at every step of the simulated algorithm. Murshed [33] introduced other simulation algorithms for the LR-Mesh with *monotonic configurations* and without solving the connected components problem. However, even for the smaller sized R-Mesh, the constant delay assumption for the buses is difficult to realize. Other directions for implementing dynamic reconfiguration are technology-based [28, 29]. Several prototypes were proposed [27, 39, 45] but they have not kept pace with algorithmic advances on reconfigurable models.

FPGAs (for example, see Wakerly [51]), though very different from models such as the R-Mesh, provide a practical platform that supports reconfiguration. These devices have hardware that can be configured to suit the problem at hand. Generally speaking, an FPGA consists of an array of logic blocks that can be connected using horizontal and vertical channels. Switches that can be configured are located at the intersection of the horizontal and vertical channels. By configuring the switches, different connections can be established between logic blocks. FPGAs have the flexibility of implementing different functions and the efficiency that comes from exploiting the possible parallelism in the problem. They have also proven very useful for rapid prototyping. Traditional FPGAs configure their switches using information generated outside the chip. Consequently, pin limitation is one of the biggest hurdles for run-time reconfiguration in these devices.

For FPGAs too, technological advances, coupled with ideas such as pre-loaded contexts [9, 18, 38] and partial reconfiguration [2, 52] have reduced reconfiguration time substantially. However, one cannot expect current FPGA-type devices to support the "R-Mesh-type" reconfiguration, in which connections could be altered at each step of the computation.

Perhaps one of the most promising first steps in implementing the R-Mesh-type reconfiguration was due to Sidhu *et al.* who proposed the *Self-Reconfigurable Gate Array* (SRGA) [40, 41, 42, 43]. This architecture augments an FPGA-type structure with the ability to generate reconfiguration information from within the device (self-reconfiguration). Consequently, it can change its configuration extremely fast (in a

few clock cycles). This self-reconfiguration feature has much in common with the R-Mesh in that local information is used to generate a configuration with global relevance.

1.2 Background

In this section we introduce some ideas that will assist in describing the scope and contribution of this work. Specifically, we discuss four broad ideas: (1) the Self-Reconfigurable Gate Array Architecture (SRGA), (2) the Circuit Switched (binary) Tree (CST) interconnect, (3) the segmentable bus and (4) the Reconfigurable Mesh (R-Mesh). We now discuss these topics briefly.

1.2.1 Self-Reconfigurable Gate Array Architecture

The Self-Reconfigurable Gate Array Architecture (SRGA) consists of an array of processing elements (PEs) connected by rows and columns of trees, much like a mesh-of-trees structure [24] (see Figure 1.1). Each PE consists of a flip-flop, look-up table (LUT), local memory and a small amount of control hardware. One could view a PE as a small 1-bit processor. Each PE is a leaf of a row tree and column tree that the PE uses to communicate with other PEs. As defined by Sidhu *et al.* [43], the SRGA permits a tree to only connect one pair of PEs at a time. We adopt a somewhat more general view of the tree and allow it to connect multiple processor pairs simultaneously. This general tree, called the circuit switched tree, is discussed next.

1.2.2 The Circuit Switched Tree

The Circuit Switched Tree (CST) is a balanced binary tree whose leaves are PEs and internal nodes are switches that can be configured to establish various paths among leaves. The edges of the CST represent full duplex links (that allow simultaneous communications in opposite directions.) Thus, we will view the CST as a directed graph with each tree edge replaced by two oppositely directed edges. For a communication to be performed on the CST, a directed shortest path must be established

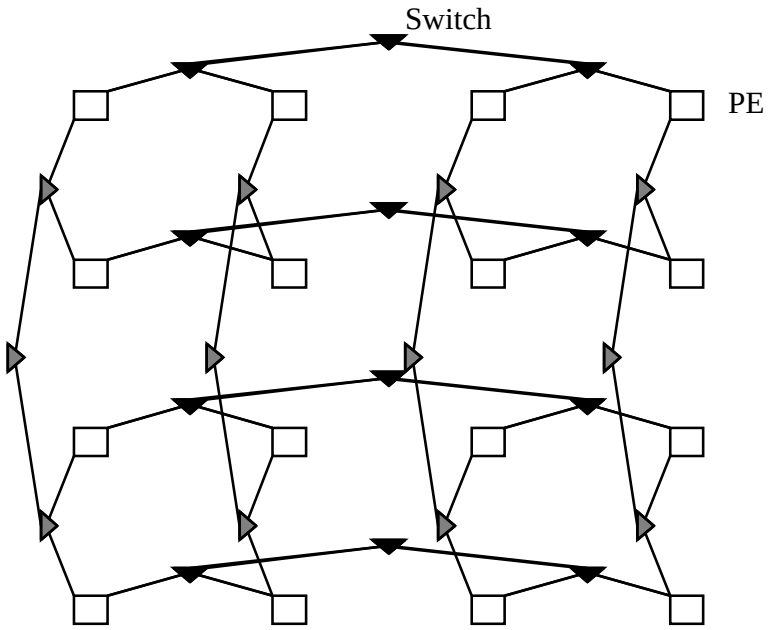
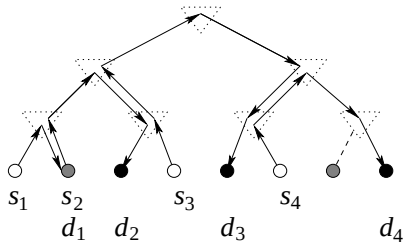


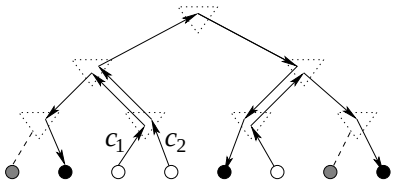
Figure 1.1: 4×4 PE array

between two leaves (PEs) (since the underlying structure is a tree, there is a unique shortest path between any pair of leaves.)

A communication that has only one source and one destination is called a one-to-one communication (see Figure 1.2 for examples of one-to-one communications on the CST). Two communications can be accommodated simultaneously on the CST if they do not use a common directed edge. The *width* of a set of one-to-one communications is the maximum number of communications that use any given directed edge (the width of the communication set of Figure 1.2(a) is 1 because no two communications use a common edge whereas the communication set of Figure 1.2(b) has width 2 because the communications labeled c_1 and c_2 share a common directed edge.) Clearly, all communications in a width-1 communication can be accommodated simultaneously on the CST. If two communications use a common directed edge, then they are said to form an *incompatible*. The incompatible is called a *source* (resp. *destination*) *incompatible* if the two communications use an edge going up (resp. down) the tree. Clearly, the size of the largest incompatible in a communication set

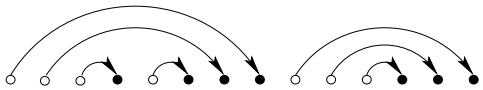


(a) A width-1 communication set

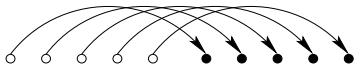


(b) A width-2 communication set

Figure 1.2: A sample set of communications. Sources, s_i, and destinations, d_i are shown as white and black circles, respectively. A PE could be both a source and a destination, or neither (shown shaded in grey).



(a) An oriented well-nested set



(b) An oriented monotonic set

Figure 1.3: Examples of oriented communication sets

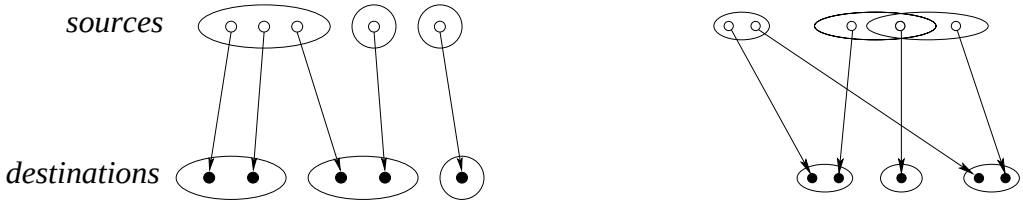
is the same as the width of the communication set. If a width-w communication set can be scheduled on the CST in w steps, then the communication set is said to be *width partitionable*.

A communication set is *oriented* if either (1) for each communication in it, the source is a leaf that is to the left of the destination on the CST, or (2) for each communication, the source is to the right of the destination (Figures 1.3(a) and (b) show oriented communication sets while those in Figures 1.4(a) and (b) are not oriented because some sources are to the right of their destinations while other sources are to the left of their destinations.)

Another way to classify communication sets is by using the pattern of communications they form, regardless of the source-destination orientation. In a well-nested communication set the communications can be represented as well nested parentheses (see Figure 1.3(a)); i.e., each communication is entirely inside another communication

(a) A non-oriented well-nested set (b) A non-oriented monotonic set

Figure 1.4: Examples of non oriented communication sets



(a) A communication set with disjoint incompatibles

(b) A communication set with non-disjoint incompatibles

Figure 1.5: Examples of communication sets with (a) disjoint incompatibles and (b) non-disjoint incompatibles

or concatenated with another well-nested communication set. A monotonic communication set forms a stride of communications (see Figure 1.3(b)). Yet another way to classify communication sets is based on the properties of its incompatibles. For example, the incompatibles of a communication set could be disjoint if no two incompatibles have a common element (see Figure 1.5(a)) or non-disjoint (see Figure 1.5(b).) The bipartite graphs shown in Figure 1.5 are called *incompatibility graphs* and represent communication sets. The source incompatibles and destination incompatibles are shown encircled. We use these ideas to derive properties related to accommodating communication sets on the CST. In particular, we will address the question of scheduling a communication set so that only a width-1 set is scheduled in a step.

The ability of the CST to accommodate a width-1 communication set does not guarantee the ability to perform these communications. Performing the communications requires configuring the switches of the CST to physically establish the communications paths. We use the term configuring the CST to refer to configuring its switches to establish the communications paths between sources and destinations.

The settings (configuration) of each switch needed to successfully establish the communications can be computed at run time or compile time. In run-time configuration, different configuration information is generated at each step of the algorithm. This information could be based on the particular input instance and on the results of the previous steps of the algorithm. On the other hand, compile-time configuration information is instance-independent and is computed before the algorithm starts. One of the derivatives of our method to accommodate width-1 communication sets on the CST is an implementation of the segmentable bus, described below.

1.2.3 Segmentable Bus

The structure of an N-processor *segmentable bus* [48] is shown in Figure 1.6. Each processor controls (opens/closes) a segment switch on the bus using local information.

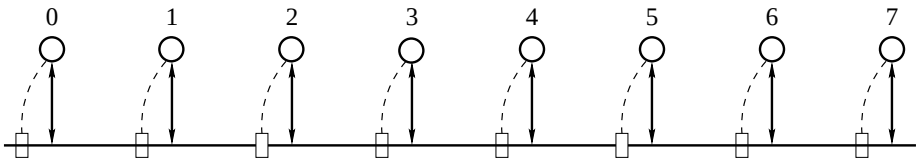
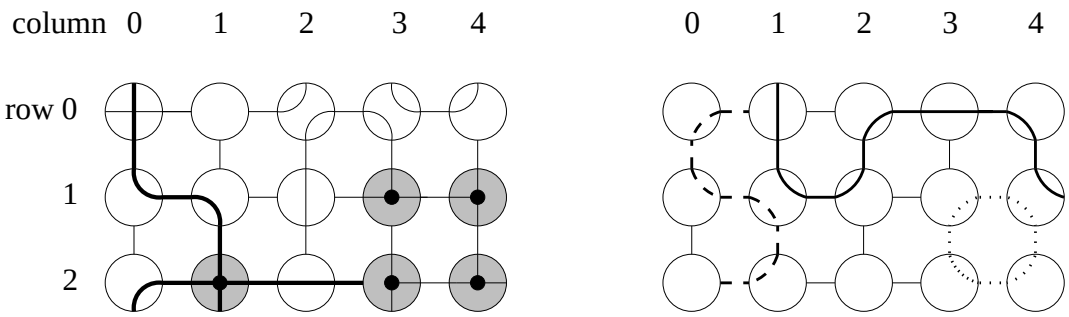


Figure 1.6: An 8-processor segmentable bus; bidirectional lines are data links between the processors and the bus; dashed lines allow processors to control their segment switches.

Opening or closing the switches transforms the segmentable bus into blocks of contiguous processors (segments); that is, local information at each processor is translated into information with global relevance. Each processor can write to its segment and all other processors incident on the segment can read the written data. A segmentable bus can also be viewed as a one-dimensional R-Mesh (see Section 1.2.4.) A segmentable bus plays a vital role in our implementations of R-Mesh-type models. We now describe these models.

1.2.4 Reconfigurable Mesh

The *Reconfigurable Mesh* (R-Mesh) is a two-dimensional array of processors connected by an underlying mesh (see Figure 1.7). Each processor has four ports (called North,



(a) R-Mesh (b) LR-Mesh

Figure 1.7: Examples of buses in a 3×5 R-Mesh and LR-Mesh

South, East, and West ports in the obvious manner, and abbreviated N, S, E, and W). Each processor can independently partition its ports so that ports in the same block of a partition are connected to each other. As shown in Figure 1.7(a), fifteen different port partitions are possible. The port partitions along with the underlying mesh connections between neighboring processors form buses connecting processors. Figure 1.7 shows buses in bold, dashed, and dotted. An R-Mesh that assumes constant bus delay, regardless of the number of ports spanned by the bus, is called a *unit-cost* R-Mesh. The Linear R-Mesh (LR-Mesh) [4]is a restricted version of the R-Mesh whose buses are not allowed to branch (see Figure 1.7(b)). Numerous R-Mesh algorithms run on the LR-Mesh without loss of speed. A Horizontal-Vertical Reconfigurable Mesh (HV-R-Mesh) [4] is another restricted version of the R-Mesh whose buses are not allowed to bend from a row to a column or vice versa. A bit-model R-Mesh [21] is a fine-grained version of the R-Mesh with processors of constant size (like PEs of the SRGA architecture).

The R-Mesh solves problems very differently than conventional models. Consider the problem of counting the number of 1's among N input bits (see Figure 1.8). The inputs are at the top row and available to the respective columns. The counting algorithm constructs buses starting at the processors of the West border of the R-Mesh and move down one row for each 1 in the input (see Figure 1.8).

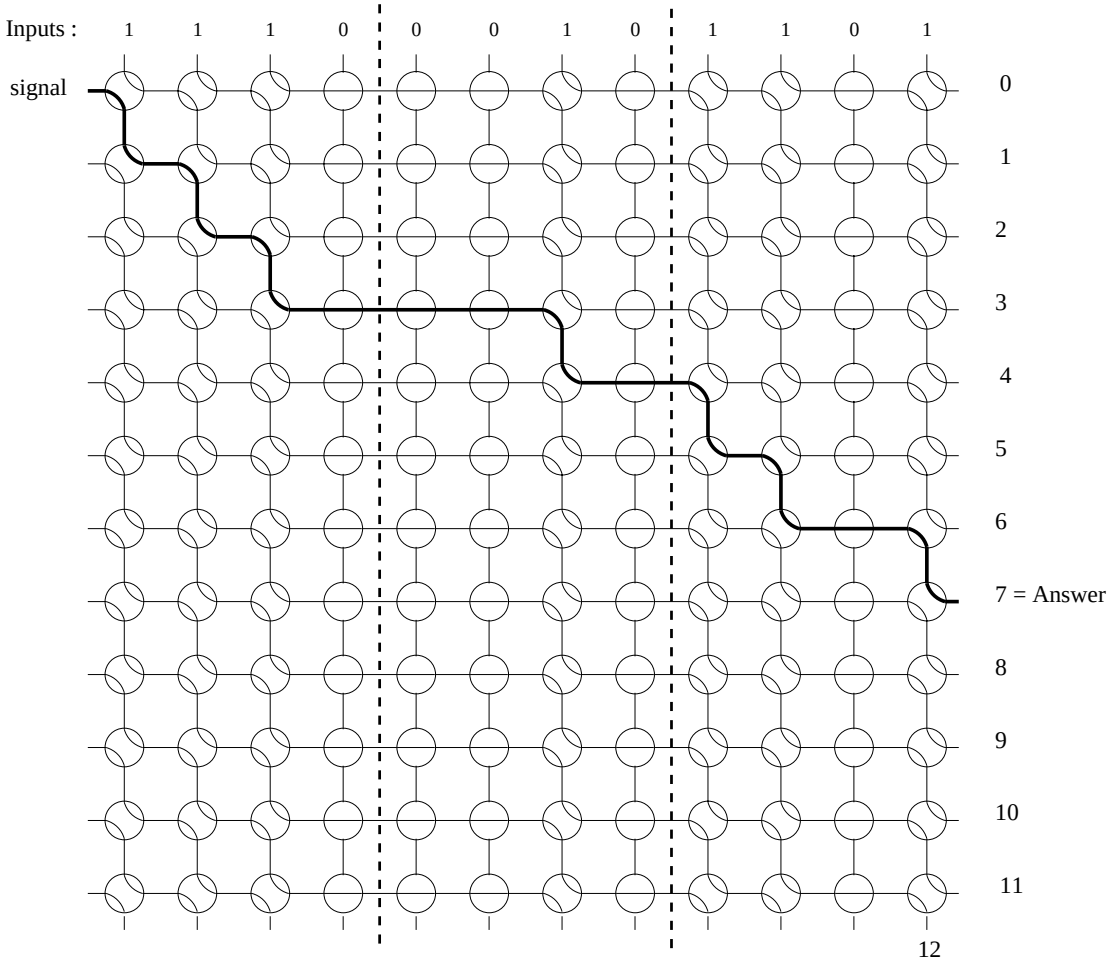


Figure 1.8: Counting N bits

The bus starting at processor $(0,0)$ (top left corner) reaches processor $(\alpha, N-1)$ (in row α and column $N-1$) iff the input bits include α 1's. If a processor $(0,0)$ sends a signal from its West port, it will reach processor $(\alpha, N-1)$ where α is the number of 1's in the input.

1.3 Scope of the Dissertation

As mentioned earlier, little work has been reported on implementing dynamic reconfiguration. This dissertation addresses various aspects of implementing dynamic

reconfiguration. The work is in four main directions. The first direction analyzes the communication capability of the CST (Section 1.2.2). The second direction examines strategies in configuring the CST to perform a set of communications. The third direction grapples with the issue of implementing dynamically reconfigurable models such as the R-Mesh and the LR-Mesh (see Section 1.2.4). The fourth direction is a practical study of the cost-benefits tradeoff of various dynamic reconfiguration features in the setting of an FPGA-like device.

1.3.1 Communication Capability of the CST

This direction studies the problem of scheduling communication sets on the CST so that each step of this schedule accommodates a width-1 communication set. In Chapter 3 we first prove that a width-w communication set requires at least w steps to schedule on the CST. In fact, this chapter deals primarily with width-w communication sets that can be scheduled in w steps (or width partitionable sets). Then we show three important classes of communication sets (namely, (a) those with "disjoint incompatibles" (see Figure 1.5(b)) (b) oriented, well-nested sets (see Figure 1.3(a)), and (c) oriented monotonic communication sets (see Figure 1.3(b))) to possess this property. As a special case of the second result, we show that the set of communications that can be performed in one step on a segmentable bus (see Section 1.2.3) can be scheduled in two steps on the CST. This result implies that the communication ability of the bit-model HV-R-Mesh [4], a special case of the bit-model R-Mesh [21], can be emulated by an SRGA-like architecture without significant overhead. Also as a special case of the third result, we show that the communications of a uniform hypercube [50] can be scheduled optimally on the CST.

Chapter 4 considers communication classes that are not necessarily width partitionable. We derive the minimum requirement for a communication set to be not width partitionable. Specifically, we prove that for a communication set to be not width partitionable, the set must be at least of width 2, have at least five communications, three source incompatibles and three destinations incompatibles. Further, we prove that only two such "simplest sets" are possible (to within isomorphism). Figure 1.5(b) shows one of these sets. This set requires one extra step (beyond its

width) to schedule on the CST. In general, we show that there exists a width-w set that is not width partitionable and which requires $\Theta(w)$ extra steps. Recall that in Chapter 3 we prove that oriented well-nested sets and oriented monotonic sets are width partitionable. If we allow these sets to be non-oriented (see Figure 1.4), then they are not nessecarily width partitionable. We establish this by constructing a non-oriented well-nested set and a non-oriented monotonic set whose incompatibility graphs are both the same as the one in Figure 1.5(b).

1.3.2 Configuring the CST

The work in Chapters 3 and 4 deals with converting a communication set into a series of width-1 communication sets. In Chapter 5 we deal with the issue of configuring the CST switches to accommodate any given communication set of width 1.

We first identify a class of communication sets (called edge-exclusive sets) for which the CST can be configured in one step (at run time). The idea for configuring the tree is to translate the local information at processors to global information that represents the connectivity of the communication set. Next we present an algorithm to decompose any width-1 communication set into at most three "edge-exclusive" sets. (In general, the decomposition algorithm works at compile time.) Thus, any width-1 communication set can be performed on the CST in at most three steps. Since an edge-exclusive set can be accommodated on a CST with half duplex links, a half duplex CST can simulate a full duplex CST in at most 3 steps.

Chapter 6 deals with a particular communication set, namely that of a segmentable bus. We give methods to dynamically configure CST switches to implement the functionality of a segmentable bus (see Section 1.2.3). As in Chapter 5, the idea is to translate the local information at processors to global information that represents the connectivity of a segmentable bus. We present two approaches. The first is suitable for large processors of word-size $\Omega(\log N)$ bits in which one "step" (cycle) can accommodate $\Omega(\log N)$ gate delays. This approach emulates each step of a segmentable bus in $O(1)$ steps. Although the main idea is similar to the one in Chapter 5, the first approach exploits features specific to the segmentable bus and does not use any decomposition algorithm. The second approach is suitable for smaller processors of

word-size $\Theta(\log k)$ bits where $\log\log N \leq \log k \leq \log N$ and emulates a segmentable bus step algorithmically using a normalized tree algorithm [24] in $O(\log_k N)$ steps.

1.3.3 Implementing R-Mesh-Type Models

As noted earlier, the main obstacle to implementing an R-Mesh (or other related models) is the bus delay, which these models assume to be constant. In Chapter 7 we introduce a new measure for the bus delay called the *bends-cost* measure. We show that there exists an LR-Mesh implementation for which the bends-cost is a faithful measure of the actual bus delay. This "bends-cost LR-Mesh" implementation uses the segmentable bus derived in Chapter 6 as a building block. Then we describe methods to use the bends-cost measure in algorithm design. Let Δ denote the delay of an N-processor segmentable bus. We prove that for any delay $D \geq \Delta$, a $\Theta(N) \times \Theta(N)$ bends-cost LR-Mesh can find the prefix sums of N bits or sort N elements in $O\left(\frac{\log N}{\log D - \log \Delta}\right)$ time. A similar result for adding $N \quad b = O(\log N)$-bit numbers runs in the same time but on a $\Theta(Nb) \times \Theta(Nb)$ bends-cost LR-Mesh. These processor resources are within a constant factor of the original (unit-cost) LR-Mesh algorithms on which they are based. In particular, if $D = N^\epsilon$, $0 < \epsilon < 1$, then our algorithms have the same time computed on the idealized LR-Mesh, but run on an implementable platform. The ideas used to achieve these results apply to a large class of algorithms that use "incremental buses" (defined formally in Section 7.1.) That is, we establish that any T step (unit-cost) LR-Mesh algorithm using incremental buses runs on the bends-cost LR-Mesh in $O\left(T\left(\frac{\log N}{\log D - \log \Delta}\right)\right)$ time using buses of delay at most D. We then further generalize this result for "semi-monotonic buses," a large class of bus configurations, and prove that any T step (unit-cost) LR-Mesh algorithm using incremental buses runs on the bends-cost LR-Mesh in $O\left(T\left(\frac{\log N}{\log D - \log \Delta}\right)^2\right)$ time using buses of delay at most D.

In Chapter 8 we consider the computational power of the bends-cost LR-Mesh. We show that an arbitrary configuration of an $N \times N$ unit-cost LR-Mesh can be simulated in $O\left(\frac{\log N}{\log D - \log \Delta}\right)$ time on an $O\left(\frac{DN^2}{\Delta}\right) \times O\left(\frac{DN^2}{\Delta}\right)$ bends-cost LR-Mesh whose buses have a delay of at most D. In other words, if $D = N^\epsilon$ (where $0 < \epsilon < 1$), then

the bends-cost LR-Mesh can emulate any step of a unit-cost LR-Mesh in constant time; that is, the bends-cost LR-Mesh is equal in power to the unit-cost LR-Mesh.

1.3.4 Cost-Benefit Tradeoff Study

In Chapter 9 we combine ideas from the bends-cost LR-Mesh (see Section 1.3.3), the CST (see Section 1.2.2), and the SRGA (see Section 1.2.1) to construct the "Enhanced SRGA" (E-SRGA) architecture. The E-SRGA adds dynamic reconfiguration features to the SRGA platform and can be viewed as a possible implementation of a bit-model bends-cost LR-Mesh. These features include the ability to connect its PEs in rows/columns as a segmentable bus using local data and the ability of each PE to configure its switches directly on the basis of local data. These features are in addition to the SRGA's ability to connect pairs of PEs.

We have coded the E-SRGA in VHDL and synthesized the architecture using a 0.5 micron library of standard cells from AMI. The Leonardo Spectrum (synthesis tool) was used for the synthesis and optimization of the architecture. A C program was written to automate the implementation of E-SRGAs of different sizes. We conducted experiments to ascertain the cost-benefit tradeoff of these dynamic reconfiguration features.

1.4 Contributions of this Work

Dynamic reconfiguration has provided platforms capable of very fast solutions to many problems. However, models of dynamic reconfiguration such as the R-Mesh are difficult to realize because of the constant delay assumption that is central to such models. This work bridges the gap between theory and practice. Significant contributions have been made towards translating theoretical algorithms to practical solutions. Many aspects of dynamic reconfiguration have been examined as explained below.

One important component of our work is the CST (see Section 1.2.2). Chapters 3 and 4 present a formal study of the communication capability of the CST and provide a better understanding of the capabilities and the limitations of this important

structure. In Chapter 3 we show that some interesting communication classes can be scheduled optimally on the CST. In particular, we show that the communications of a step of the segmentable bus can be accommodated in at most two steps on the CST. This result is significant as the segmentable bus is one of the most fundamental components of a dynamically reconfigurable architecture. Not all communication sets are width partitionable (see Figure 1.4). Chapter 4 deals with such communication sets. Our work here provides a clearer understanding of some conditions under which a communication set is not width partitionable.

The analysis of the CST introduces new concepts and methodologies whose utility extends far beyond the CST and the communication sets constructed. That is, the concepts presented here are general enough so that they can be used in analyzing other interconnection networks (not necessarily the CST) and communication sets.

In Chapter 5 we study the problem of configuring the CST. Configuring the CST switches based on local data to reflect a global context is the essence of dynamic reconfiguration. This important issue is not considered by Sidhu *et al.* [43], who originally proposed the SRGA which is based on the CST. As we noted in Section 1.3.2, we propose a method to decompose any width-1 communication set into at most three edge-exclusive sets (for which CST switches can be configured at run time).

The results of Chapters 3, 4, and 5 provide a comprehensive set of results to perform virtually any set of communications on the CST. If the communication set is width partitionable, then Chapter 3 gives means to schedule the set; i.e., breaks it up into width-1 sets. For non-width partitionable sets, Chapter 4 provides the same means. The work of Chapter 5 allows the communications of these width-1 sets to be actually performed on the CST in at most three steps. Thus, for example, communications of every width partitionable set of width w can be performed on the CST in at most $3w$ steps.

Chapter 6 uses the CST to implement segmentable buses. As noted earlier, a segmentable bus is a fundamental dynamically reconfigurable structure. We present two approaches. One is suitable for large processors of size $\Omega(\log N)$ while the other is suitable for smaller processors of size $O(\log k)$ bits, where $\log\log N \leq \log k \leq \log N$. Collectively, the two approaches allow all levels of processor granularity to adopt

the segmentable bus structure varying from an FPGA type structure to a mesh of processors.

An important contribution of Chapter 7 is the new measure of bus delay called bends-cost. This new measure allows the algorithm designer to estimate bus delay accurately, yet abstract away from hardware details. It also provides a new approach to designing R-Mesh algorithms in which the designer carefully factors in the number of bends in buses used by the algorithm.

The bends-cost LR-Mesh implementation for the LR-Mesh validates the bends-cost measure. It has independent value as an LR-Mesh realization as well. Then, we present simulation algorithms to simulate the unit-cost LR-Mesh with semimonotonic configurations on the bends-cost LR-Mesh. Our main result shows that if $D = N^{\epsilon}$ for an arbitrarily small constant $\epsilon > 0$, then the running times of the bends-cost LR-Mesh algorithms are within a constant of their ideal (unit-cost) LR-Mesh counterparts. This is the first general result that admits constant time algorithms on reconfigurable models without resorting to the use of the unit-cost measure for bus delay. Our approach also opens the door to translating the large body of fast LR-Mesh algorithms to run on a more practical platform.

Chapter 8 which deals with the computational power of the bends-cost LR-Mesh, serves to show that with delay of $D = N^{\epsilon}$ (where $0 < \epsilon < 1$) the bends-cost LR-Mesh can compute anything the idealized unit-cost LR-Mesh can compute without significant loss of speed.

The practical study of Chapter 9 brings together many of the concepts explored in previous chapters in a practical FPGA-type setting. The E-SRGA architecture that we propose uses an interconnection network that occupies only about 6% of the chip area while accommodating a rich array of communication patterns. Thus it provides a higher functional density than typical FPGAs. While the E-SRGA is well suited for algorithmic solutions to problems, it is not as nimble as an FPGA for implementing circuits. Most importantly, the results here point to the feasibility of the ideas proposed in previous chapters.

1.5 Organization of the Dissertation

In the next chapter we present some preliminary concepts and definitions. The communication capability of the CST is analyzed in Chapter 3 (for width partitionable sets) and in Chapter 4 (for non-width-partitionable sets). In Chapter 5 we address the issue of configuring the CST for any width-1 communication set. Chapter 6 deals with segmentable bus implementations. Chapter 7 presents the bends-cost LR-Mesh and its simulation of the unit-cost LR-Mesh. Chapter 8 addresses the computational power of the bends-cost LR-Mesh. Chapter 9 describes the cost-benefit tradeoff of the E-SRGA architecture. Chapter 10 summarizes our results and identifies several open problems.

Chapter 2

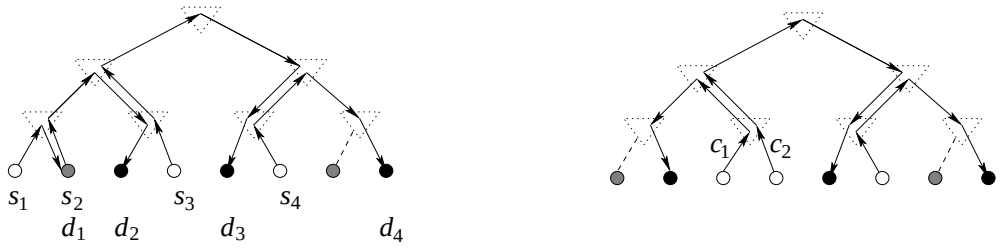
Preliminaries

In this chapter we introduce some basic ideas and definitions used throughout the dissertation. In the next section we present the circuit switched tree (CST). In Section 2.2 we describe the segmentable bus, a fundamental reconfigurable structure. Section 2.3 presents the R-Mesh model. Section 2.4 introduces the LR-Mesh, an important restriction of the R-Mesh. Finally, Section 2.5 discusses the notion of bus delay.

2.1 The Circuit Switched Tree

The circuit switched tree (CST) is a balanced binary tree whose leaves are PEs and whose internal nodes are switches (see Figure 2.1). Each switch has a full-duplex link to its parent (if any) and two children. (A full-duplex link can carry information in both directions simultaneously.) The switch can be configured to connect to its parents and children in various ways. Figure 2.2 shows representative configurations. Additional configurations can be obtained from those shown in the figure by symmetry and rotation. Some of these configurations are simple extensions of those used in the SRGA architecture to include broadcasting. Observe that while an incoming link can connect to two outgoing links (for broadcasting), two incoming links cannot both lead to the same outgoing link (concurrent writes [1] are not permitted). Also a switch cannot connect an incoming link to an outgoing link in the same "side" of the switch. This ensures that for a tree with N leaves (PEs), every communication will traverse

[1] we relax this assumption in Section 6.2.3 and allow concurrent writes.



(a) (b)

Figure 2.1: A sample set of communications. Sources, s_i, and destinations, d_i, are shown as white and black circles, respectively. A PE could be both a source and a destination, or neither (shown shaded in grey).

no more than $2 \log N$ switches. Any pair of leaves (PEs) connected by a dedicated path through the switches can communicate in one step.

In addition to the data links, there is a control line between each node and its parent that conveys control symbols from the switch to its parent. Control symbols are used to configure the switches.

Figure 2.3 shows the internal structure of the switch. The connection unit box labeled C is a combinational logic that connects the appropriate data inputs to the data outputs to achieve the configurations illustrated in Figure 2.2. The control unit has an input to the connection unit that selects one of these configurations.

It should be pointed out that the CST is very different from a traditional point-to-point tree topology, where each node is a processor that stores and forwards packets along the correct path. In contrast, the CST switches consist of combinational logic and establish dedicated paths between leaves of the tree (as in circuit switching). Two paths can be used simultaneously, only if they have no tree edges in common.

In the next few chapters we will consider two broad issues regarding the CST. The first addresses the ability of the CST to simultaneously accommodate many one-to-one communications (Chapters 3 and 4). For example, the communications of Figure 2.1(a) can be accommodated simultaneously on the CST because no two communications use the same directed edge, whereas the communications of Figure 2.1(b)

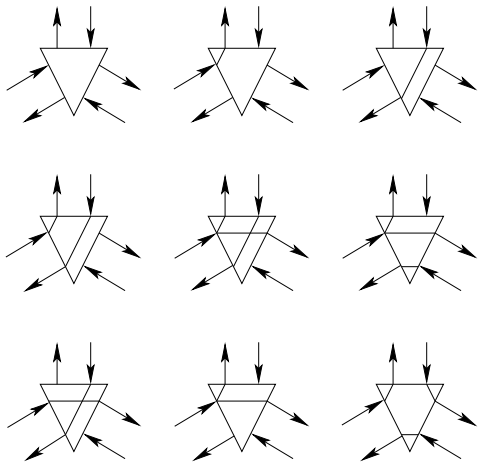


Figure 2.2: Some CST switch configurations

cannot be accommodated simultaneously on the CST because communications c_1 and c_2 use a common edge. However, the communications of Figure 2.1(b) can be accommodated in two steps (scheduled). We will use the term scheduling a communication set to refer to the need for more than one step to accommodate the communications. This issue of accommodating (scheduling) communication sets on the CST does not reflect the complexity of generating control information to configure the connection unit within each switch. The second issue addresses this point. Therefore we will make a distinction between accommodating (or scheduling) a set of communications and performing that set of communications.

2.2 Segmentable Bus

The segmentable bus is one of the most fundamental structures in reconfigurable computing. In Chapter 6 we present methods to implement segmentable buses using binary trees. Functionally speaking, an N-element *segmentable bus* has the structure shown in Figure 2.4. Each processor is connected to a bus by a bidirectional (read/write) port and controls a segment switch that is placed on the bus. Each segment switch can be in the "open" or "closed" state. When open, a segment switch cuts the bus at the point at which it is placed; otherwise, the switch is closed and the

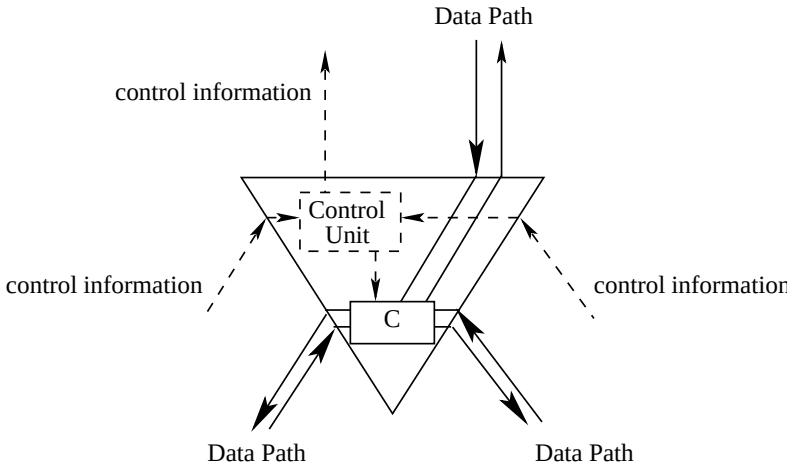


Figure 2.3: Internal structure of the CST switch

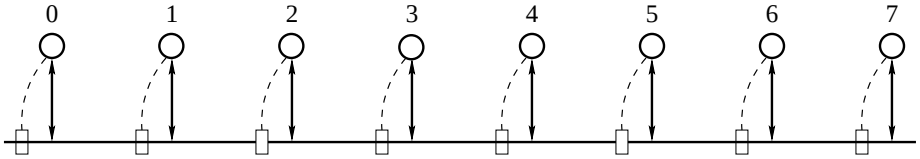


Figure 2.4: Structure of an 8-processor segmentable bus

bus passes through it seamlessly. A *bus configuration* is a set of segment switch states. Each segment switch is controlled by a processor (the one to its right in Figure 2.4). Since each segment switch can be controlled indepedently, numerous bus configurations are possible. An unsegmented portion of the bus between two open segment switches is called a *bus segment*. Figure 2.5 shows an example bus configuration of an 8-element segmentable bus with three bus segments.

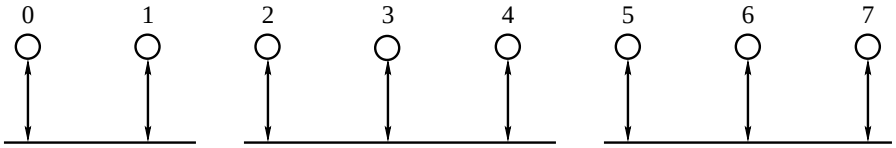


Figure 2.5: A configuration of an 8-processor segmentable bus

The segmentable bus architecture, like other reconfigurable architectures, is synchronous. At any given step each processor could perform the following actions: (1) open or close its segment switch, (2) read from or write to its bus segment, and (3) perform a local computation.

The decision to open or close the segment switch is based entirely on local information. Thus, each processor can independently control its segment switch. The reading or writing on a segment could be exclusive or concurrent. In an exclusive read (resp., write), a segment can have only one reader (resp., writer). In a concurrent read (resp., write) segmentable bus, a segment can have multiple readers (resp., writers). As in a PRAM [20], we use ER, CR, EW, and CW to denote exclusive read, concurrent read, exclusive write, and concurrent write. Thus, for example, a segmentable bus with concurrent read and exclusive write is called a CREW segmentable bus. Again as in a PRAM with concurrent writes, the segmentable bus uses a rule to determine the values written to the bus. In this dissertation we consider the COMMON, COLLISION, COLLISION$^+$, PRIORITY, and ARBITRARY rules. In the COMMON rule, all the values written to any bus segment must be the same. Under the COLLISION rule, a collision symbol is written on any segment with multiple writers. The COLLISION$^+$ rule is the same as COMMON if all the values written to the segment are the same, otherwise a collision symbol is written to the segment. The PRIORITY rule assumes a fixed priority (usually the index) to each processor and allows the highest priority writer to write its value to the segment. Finally, the ARBITRARY rule selects any one writer to write to the segment.

The segmentable bus can also be viewed as shown in Figure 2.6 where each processor has two ports (East and West) and an internal switch. By closing (resp., opening) its internal switch, each processor can connect (resp., disconnect) its two ports forming segments of ports. The structure shown in Figure 2.6 is also known as a one dimensional R-Mesh.

The segmentable bus assumes that two processors can communicate in one step. In other words, the segmentable bus assumes a unit-cost bus delay. As a reconfigurable model, it could use any of the bus delay measures described in Section 2.5. As we explained before, in Chapter 6 we present methods to implement segmentable

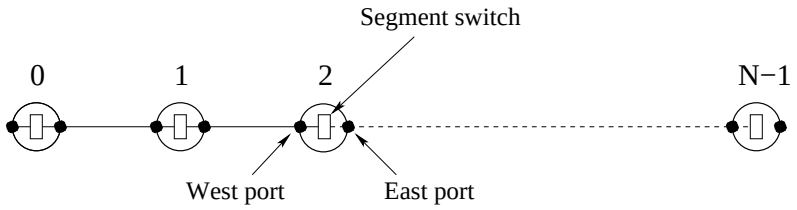
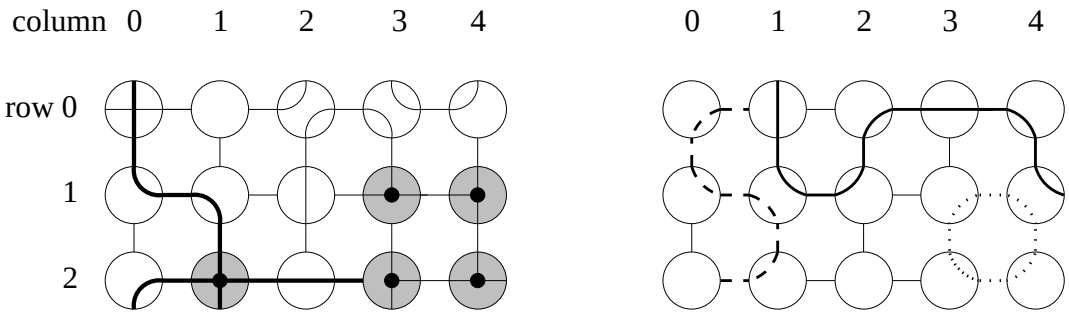


Figure 2.6: Another representation of an N-processor segmentable bus



(a) R-Mesh (b) LR-Mesh

Figure 2.7: Example of buses in a 3×5 R-Mesh and LR-Mesh

buses. If the implementation takes s steps to perform the functionality of a 1-step segmentable bus, then we say that the implementation of the segmentable bus runs in s steps.

2.3 The Reconfigurable Mesh

An $R \times C$ *reconfigurable mesh* or *R-Mesh* [32] consists of an R-row, C-column array of processors connected by an underlying mesh (see Figure 2.7). Number the rows (resp., columns) $0, 1, \cdots, R-1$ (resp., $0, 1, \cdots, C-1$). Each processor has four ports (called North, South, East, and West ports in the obvious manner, and abbreviated N, S, E, and W). Each processor can independently partition its ports to connect certain ports together leaving other ports unconnected. For example, the top left processor of Figure 2.7(a) connects its N port to its S port, and its E port to its W port. The corresponding partition is denoted by $\{\overline{N, S}\ ;\ \overline{E, W}\}$. Figure 2.7(a) shows the fifteen possible port partitions of the R-Mesh. An assignment of a port partition

to each R-Mesh processor is called a *configuration*. Figure 2.7 shows two different configurations. The port partitions along with the underlying mesh connections between neighboring processors form buses connecting processors. Figure 2.7(b) shows buses in bold, dashed, and dotted. An assumption central to all traditional R-Mesh algorithms is that buses have constant delay, regardless of the number of processors they span. An R-Mesh making this assumption is called a "unit-cost R-Mesh." While this assumption enables us to design very fast algorithms, it makes it very difficult to implement such a model.

At each step of an R-Mesh algorithm, a PE could perform the following actions: (1) configure (partition) its ports, (2) read from and write to its ports, and (3) perform a local computation. As in a segmentable bus, an R-Mesh could permit concurrent reads and writes. If more than one processor is allowed to write to a bus at the same time, then the R-Mesh has concurrent write ability and the concurrent write rules (described in the previous section) are used to resolve the values written to the bus.

2.4 The LR-Mesh

In Chapter 7 we consider a restricted version of the R-Mesh called the *Linear R-Mesh* or *LR-Mesh* [3, 17] (see Figure 2.7(b)) whose buses are linear (non-branching); that is, an LR-Mesh processor cannot use the five partitions $\{\overline{N, S, E, W}\}$, $\{\overline{N}\ ;\ \overline{S, E, W}\}$, $\{\overline{S}\ ;\ \overline{N, E, W}\}$, $\{\overline{E}\ ;\ \overline{N, S, W}\}$ and $\{\overline{W}\ ;\ \overline{N, S, E}\}$ in the shaded processors of Figure 2.7(a). Notwithstanding this restriction, the LR-Mesh can generate an exponential number of different buses among its processors and solve many problems extremely quickly. Indeed, most R-Mesh algorithms run on the LR-Mesh. The counting algorithm presented in Section 1.2 is an example of an algorithm that runs on both an R-Mesh and an LR-Mesh.

2.5 Bus Delay

A traditional bus (see Figure 2.8) is a set of wires with multiple taps connecting processors to it. Each processor incident on the bus has a read port and a write port

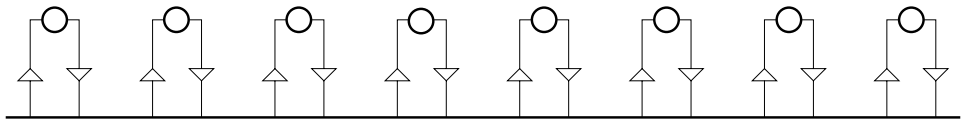


Figure 2.8: Structure of a traditional bus

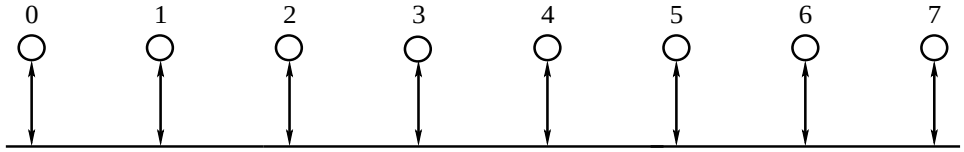


Figure 2.9: A segmentable bus with all processors connected

connecting it to the bus. Capacitive loading [44] due to a large number of taps causes the bus to use a reduced data rate.

On the other hand, a reconfigurable bus connects elements (processors) in different ways at different steps. The set of elements it connects can change at each step. The bus uses switches to change its structure to achieve the desired connectivity. Therefore, switches are located on the data path, forming a combinational circuit rather than that shown in Figure 2.8. Consider the simple case of a segmentable bus all of whose processors are connected (see Figure 2.9) and a segmentable bus all of whose processors are disconnected (see Figure 2.10). To achieve different configurations for the segmentable bus switches, have to be closed using logic gates on the data path. Figure 2.11 shows a data path between processors 0 and 5. The AND gates on the data path represent a combinational circuit. Because there are relatively few taps between two successive gates, capacitive loading is not a big problem. The primary concern is the switch delay of the longest path of this circuit. A large delay forces a reduction in the data rate as a bus should not be reconfigured before the current bus cycle has been completed. A conventional implementation of a general reconfigurable bus (see Figure 2.11) spanning N processors has a combinational path with $\Omega(N)$ gate delays.

For other technologies, for example with reconfigurable optical buses, switches are placed on the optical path. Each switch attenuates the optical signal requiring detectors to be illuminated for longer periods of time to ensure reliable operation.

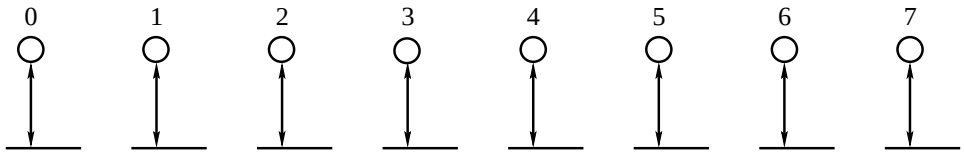


Figure 2.10: A segmentable bus with all processors disconnected

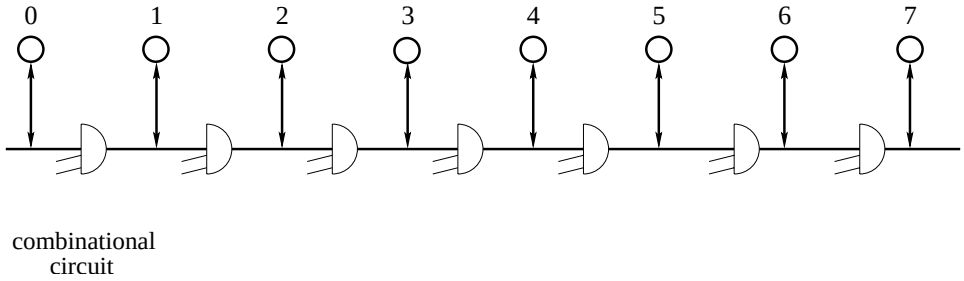


Figure 2.11: A bus represented as a combinational circuit

Once again the effect is a longer bus cycle. Thus the net effect of buses spanning a large number of processors is a reduction in data rate. We use the term bus-delay to capture this detrimental effect.

A good algorithm on a reconfigurable model should use buses with small delay. Because buses can take numerous shapes and forms, it is very difficult to accurately ascertain bus-delay. Consequently, bus-delay measures have to be used as approximations of the actual delay. These include the linear-cost measure, unit-cost measure, and logarithmic-cost measure. With the *linear-cost* measure, a bus spanning N processors has a delay of $\Theta(N)$. While this measure is quite accurate, it renders most reconfigurable models' algorithms too slow for practical use. Most work on reconfigurable models assumes the *unit-cost* measure [32] that assumes a bus to have constant delay, regardless of the number of processors it spans. Clearly, this measure does not reflect reality. A more conservative *log-cost* measure [32] assigns a $\log N$ delay to a bus spanning N processors. While this measure is reasonable for a fixed bus, it does not capture the complexities arising from the ability of a reconfigurable model to configure its buses in an exponential number of ways. In Chapter 7 we introduce a

new measure for bus delay called *bends-cost* that accurately represents the actual bus delay and yet provides the abstraction needed for convenient design.

Chapter 3

CST Communication—Width Partitionable Sets

In this chapter we study the circuit switched tree (CST) interconnect of the SRGA architecture of Sidhu *et al.* [40, 41, 42, 43] (see also Section 2.1). The CST is a balanced binary tree with PEs (or processors) at its leaves and switches at its internal nodes. These switches can be configured to establish dedicated directed paths between pairs of leaves. At most one path may use a tree edge in any given direction (child to parent or parent to child).

In this chapter, we consider sets of one-to-one communications between leaves of the CST and study properties that allow communications from a set to be accommodated on the CST (see the distinction between "performing" and "accommodating/scheduling," page 20).

We first derive a condition under which pairs of processors can communicate simultaneously on the CST. Then we introduce a quantity called the "width" of the communication set and use it to derive a necessary condition for any set of k one-to-one communications to be scheduled in t steps (where $1 \leq t \leq k$) on the CST. This necessary condition is also sufficient if the communication set has a property that we call *width partitionability*. We show that the class of communication sets with disjoint incompatibles (see Section 1.2 for an intuitive definition) possesses this property. We then identify three conditions that can be used to construct other classes of communication sets: (1) *capping*, (2) *concatenation*, and (3) *interleaving*. We use these conditions to construct two important classes of communication sets

called (a) oriented, well-nested sets, and (b) oriented, monotonic sets. We prove these sets (of communications) to be width partitionable. The set of communications that can be accommodated in one step on a segmentable bus (see Section 2.2) is a special case of the "non-oriented," well-nested sets. We apply our results on oriented, well-nested sets to show that a CST can emulate a step of a segmentable bus in two steps. Oriented monotonic sets represent a rich array of communications, including those of a uniform hypercube [50].

Although the work here is motivated by the interconnect structure of the SRGA architecture, the results and techniques could be of interest in a general FPGA-type setting in which the interconnection fabric can be configured to establish various connection patterns. It should be noted that in this chapter the analysis of the communication capability of the CST does not consider the switch configurations needed to perform these communications. Chapter 5 deals with that issue (see also Section 2.1).

In the next section we derive a lower bound on the number of steps needed to schedule a set of communications on the CST and identify a property of the communication set for which this lower bound can be met. Sections 3.2–3.5 deal with three classes of communication sets that possess the above property. Section 3.6 deals with segmentable bus communications. Section 3.7 summarizes our results in this chapter and makes some concluding remarks.

3.1 Communicating over the CST

In this section we formally define the notion of communication width and prove that the CST requires at least t steps to schedule all communications from a width-t communication set. Next, we identify a property of the communication set, called width partitionability, that allows a width-t communication set to be scheduled on the CST in t steps. That is, the lower bound imposed by the width can be achieved for width partitionable communication sets. We first introduce some definitions.

Represent a CST as an N-leaf tree with A denoting its set of leaves (PEs). To account for the full duplex links of the CST, replace each tree edge by two oppositely

directed edges. For the following definitions, let $\mathcal{T}$ denote this "directed tree." For any internal node u of $\mathcal{T}$, let $\ell level(u)$ denote its level; the leaves are at level 0 and the root is at level $\log_2 N$. For example, the node labeled v in Figure 3.1 is at level 2. For a set $S \subseteq A$ of *sources* and a set $D \subseteq A$ of *destinations*, a set of k one-to-one

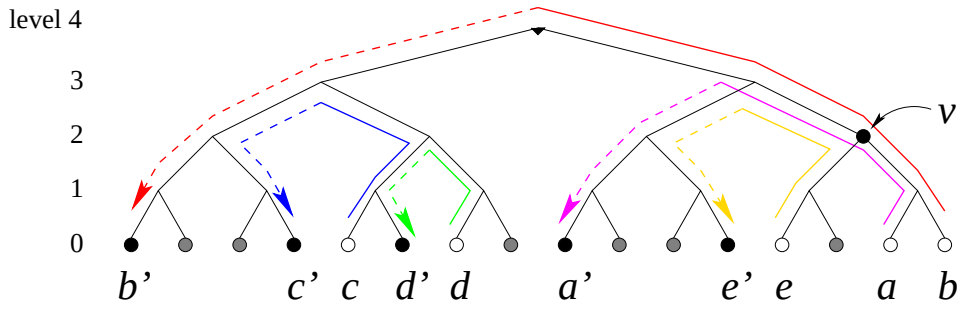


Figure 3.1: An example of a communication set. Each source-destination pair is labeled (x, x'), where $x \in \{a, b, c, d, e\}$. Sources are unshaded circles while destinations are shaded

communications, (x, x') where $x \in S$ and $x' \in D$, is simply a pairing of the elements of S and D (in Figure 3.1, $k = 5$, $S = \{a, b, c, d, e\}$ and $D = \{a', b', c', d', e'\}$). We note that a leaf of $\mathcal{T}$ could be both a source and a destination (for example, see Figure 2.1(a), page 19). Source x and destination x' of communication (x, x') are said to form a "matching" *source-destination pair*.

For a set $X \subseteq A$ of leaves of $\mathcal{T}$, let $\ell ca(X)$ denote the lowest common ancestor of all elements of X, and let $\ell(X) = \ell level(\ell ca(X))$ be the level of this lowest common ancestor. For instance in Figure 3.1, if $X_1 = \{a, b, e\}$, then $\ell ca(X_1) = v$ and $\ell(X_1) = 2$. As another example, if $X_2 = \{c', d, d'\}$, then $\ell(X_2) = 3$.

For small sets such as $\{a, b\}$, we will write $\ell ca(\{a, b\})$ and $\ell(\{a, b\})$ without braces as $\ell ca(a, b)$ and $\ell(a, b)$.

For any communication $c = (x, x')$, the edges from node x (the source of c) of the directed tree $\mathcal{T}$ to node $\ell ca(x, x')$ are called *upward edges* of c. All these edges are from a node to its parent. Similarly, the edges of $\mathcal{T}$ from node $\ell ca(x, x')$ to the destination x' are the *downward edges*. Upward (resp., downward) edges are shown solid (resp., dashed) in Figure 3.1.

Definition 3.1 Let $X \subseteq S$ be any set of sources. Set X is called a *source incompatible* if and only if $\ell(X) < \ell(x, x')$, for each communication (x, x'), $x \in X$. Similarly, $Y \subseteq D$ of destinations is a *destination incompatible* iff $\ell(Y) < \ell(y, y')$, for each communication (y, y'), $y' \in Y$. ■

In Figure 3.1 the set $\{a, b, e\}$ is a source incompatible. This is because $\ell(a, b, e) = 2$, which is smaller than $\ell(a, a') = \ell(e, e') = 3$ and $\ell(b, b') = 4$. Similarly, set $\{c', b'\}$ is a destination incompatible. On the other hand set $\{a, c\}$ is not a source incompatible as $\ell(a, c) = 4 > \ell(a, a') = \ell(c, c') = 3$. We will use the term incompatible (without the attribute source or destination) to refer to a source or a destination incompatible. Intuitively, the CST cannot accommodate two communications simultaneously if their sources and/or destinations are in the same incompatible; we prove this below in Lemma 3.1.

Remarks: We say that communications $c_x = (x, x')$ and $c_y = (y, y')$ are incompatible iff $\{x, y\}$ or $\{x', y'\}$ is an incompatible.

Definition 3.2 An incompatible I is *maximal* if no superset of I is an incompatible. A maximal incompatible I is *maximum* if no incompatible has more elements than I. ■

Note that while a maximum incompatible is always maximal, a maximal incompatible could contain as few as a single element and, therefore, need not be maximum.

Definition 3.3 The *width* of a set of communications is the size of its maximum incompatible. ■

The communication set of Figure 3.1 has source incompatibles $\{a, b, e\}$, $\{c\}$, $\{d\}$, and destination incompatibles $\{a', e'\}$, $\{b', c'\}$, $\{d'\}$. The width of this communication set is 3 because the maximum incompatible (the set $\{a, b, e\}$) is of size 3. Although the incompatibles of this example do not contain any common elements, in general incompatibles need not be disjoint.

For convenience we will represent a communication set, C, as an annotated bipartite graph called an *incompatibility graph*. The incompatibility graph $\mathcal{G} = (V, E)$ has

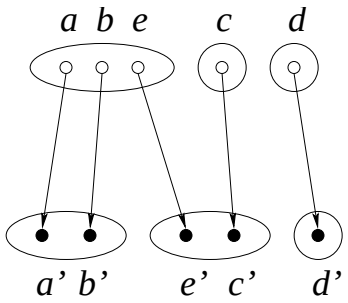


Figure 3.2: Communication set with disjoint incompatibles

the set of sources and destinations of C as its set of vertices. Note that since a leaf (PE) of the CST can be both a source as well as a destination, a leaf could appear twice in the set V. The edges of $\mathcal{G}$ connect source-destination pairs in accordance with the given communication set C. Arrange nodes of $\mathcal{G}$ as sources and destinations to form a bipartite graph and indicate incompatibles by encircling nodes in the same incompatible. For example, Figure 3.2 shows the incompatibility graph of the communication set of Figure 3.1.

Since an incompatible is a collection of sources or destinations of communications that interfere with each other, the width of a set of communications is an important factor in the amount of time required to schedule the communications. We now derive a necessary condition for the CST $\mathcal{T}$ to schedule k one-to-one communications in t steps (where $1 \leq t < k$). If $t \geq k$, then the communications can be trivially scheduled one at a time.

Lemma 3.1 *The CST requires at least t steps to schedule communications from a set that has a t-element incompatible.*

Proof: Let C denote a communication set. We prove that if C has an incompatible with t elements, then the CST cannot accommodate the communications associated with this incompatible in $t-1$ steps. Without loss of generality, let S' be a source incompatible with t elements. Since $\ell(S') < \ell(x, x')$ for each $x \in S'$, and since $\ell ca(S')$ is an ancestor of every $x \in S'$, each communication (x, x') with $x \in S'$ has to traverse the upward edge between $\ell ca(S')$ and its parent (see Figure 3.3). That is,

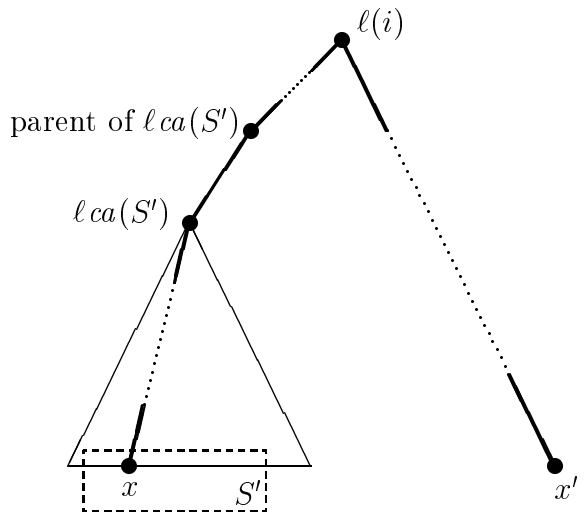


Figure 3.3: Illustration of the proof of Lemma 3.1.

all t communications with sources in S' require the link from $\ell ca(S')$ and its parent. Consequently, they require at least t steps.

In a similar manner, the existence of a destination incompatible D' with t elements implies that the link from the parent of $\ell ca(D')$ to $\ell ca(D')$ would be used at least t times. ■

Corollary 3.2 *A width-w set of communications requires at least w steps to be scheduled on a CST.* ■

Only communications from a width-1 communication set can be accommodated simultaneously on the CST. A width-w set (where $w > 1$) could be partitioned into width-1 sets $C_1, C_2, \cdots, C_\alpha$ (for some $\alpha \geq w$) so that communications from different C_i's ($1 \leq i \leq \alpha$) are accommodated in different steps. Since each C_i has width 1, all communications in C_i can be accommodated in the same step. Thus the partition corresponds to an α-step schedule for accommodating all communications in set C. Corollary 3.2 implies that $\alpha \geq w$.

Lemma 3.3 *A set S of elements is an incompatible if and only if for all $a, b \in S$, $\{a, b\}$ is an incompatible.*

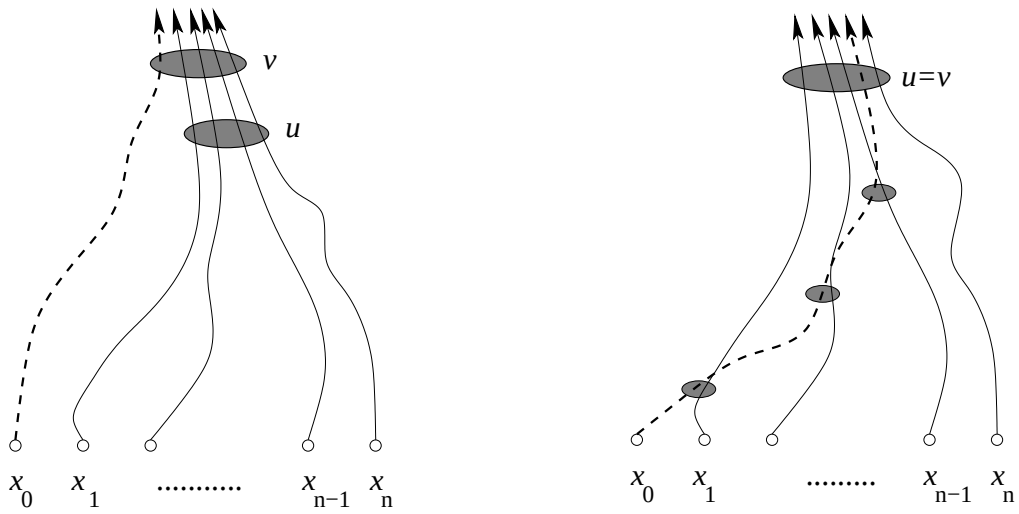


Figure 3.4: Illustration of the proof of Lemma 3.3

Proof: If S is an incompatible, then so is any subset of S. That is, for all $a, b \in S$, $\{a, b\}$ is an incompatible.

In the other direction, let $S = \{x_0, x_1, \cdots, x_n\}$. We are given that

$$\text{for all } 0 \leq i < j \leq n, \quad \{x_i, x_j\} \text{ is an incompatible.} \tag{3.1}$$

Equation 3.1 implies that $S - \{x_0\}$ is an incompatible (by the induction hypothesis). Equation 3.1 also implies that $\{x_0, x\}$ is an incompatible for all $x \in S - \{x_0\}$. Let $u = \ell ca(S - \{x_0\})$ and let $v = \ell ca(S)$. Clearly u is a descendant of v (including v itself). Since $S - \{x_0\}$ is an incompatible, each communication (x, x') (with $x \in S - \{x_0\}$) has an upward edge from u to $parent(u)$. That is, $\ell evel(u) \leq \ell evel(v)$. We consider two cases.

Case 1 [$\ell evel(u) < \ell evel(v)$]**:** The situation is shown in Figure 3.4(a). Clearly for all $0 \leq i \leq n$, communication (x_i, x_i') has upward edge $\langle v, parent(v) \rangle$. Thus, S is an incompatible (see proof of Lemma 3.1).

Case 2 [$\ell evel(u) = \ell evel(v)$]**:** The situation is shown in Figure 3.4(b). For $1 \leq i \leq n$, let $\ell ca(x_i, x_0) = w_i$ and let $\ell evel(w_i) = \ell_i$. Without loss of generality, let $\ell_1 \leq \ell_2 \leq \cdots \leq \ell_n$ (see Figure 3.4(b)). Then, $\ell ca(x_0, x_n) = u$, otherwise $\ell ca(S) \neq u$. Then, again each communication (x_i, x_i'), $1 \leq i \leq n$, has an upward edge from u to $parent(u)$. Thus in either case, S is an incompatible. ■

The necessary condition of the Corollary 3.2 applies to any set of one-to-one communications and the CST. Is this condition sufficient for all one-to-one communications? In general, the answer is "no" as it is possible for a width-w communication set to require more than w steps.

Consider the communication set $C = \{(a, a'), (b, b') \cdots, (e, e')\}$ of Figure 3.5, whose incompatibility graph is shown in Figure 3.6. Clearly, the width of C is 2.

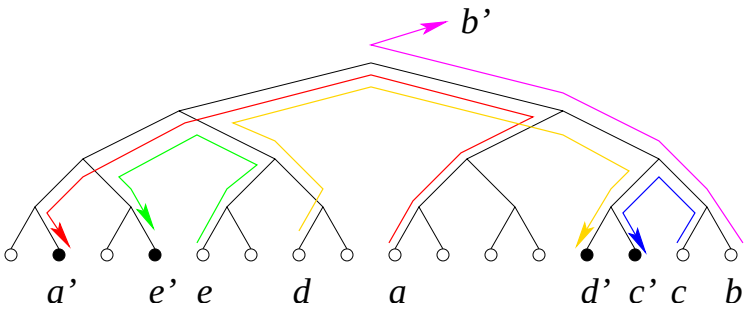


Figure 3.5: Width-2 communication set requiring three steps

The only communication that can be scheduled simultaneously with (b, b') is either (d, d') or (e, e'). Since $C - \{(b, b'), (d, d')\}$ or $C - \{(b, b'), (e, e')\}$ has width 2, it follows that C cannot be scheduled in two steps.

If a width-w communication set possesses certain properties, however, then it can be scheduled on the CST in w steps. We present one such property in the following discussion.

Lemma 3.4 *The CST can accommodate communications* $c_1 = (x, x')$ *and* $c_2 = (y, y')$ *simultaneously if and only if sets* $\{x, y\}$ *and* $\{x', y'\}$ *are not incompatibles.*

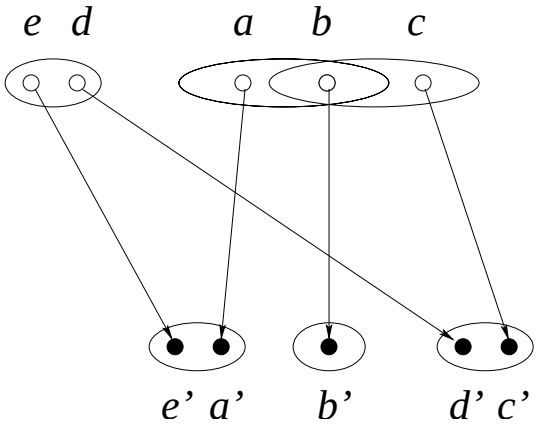


Figure 3.6: Incompatibility graph of the communication set of Figure 3.5

Proof: Clearly if either $\{x, y\}$ or $\{x', y'\}$ is an incompatible, then the width of the communication set $\{c_1, c_2\}$ is 2. Lemma 3.1 implies that they cannot be accommodated simultaneously.

If $\{x, y\}$ is not an incompatible, then both $\ell(x, x')$ and $\ell(y, y')$ cannot be strictly larger than $\ell(x, y)$. Without loss of generality, let $\ell(x, x') \leq \ell(y, y')$. This implies that $\ell(x, x') \leq \ell(x, y)$. Let $\ell ca(x, x') = u$ and $\ell(x, y) = v$. Since both u and v are ancestors of x, either v is an ancestor of u, or $u = v$. We now consider these two cases.

Case 1 $[\ell(x, x') = \ell(x, y)]$**:** Here $\ell evel(u) = \ell evel(v)$. With the observation made above, this implies that $u = v$ (see Figure 3.7(a)). Clearly, y and x' are leaves of a subtree $\mathcal{T}'$ rooted at a child w of node $u = v$. Every upward edge of communication $c_2 = (y, y')$ is either incident on a node of subtree $\mathcal{T}'$ or is on the path between w and the root of the CST $\mathcal{T}$. In contrast, every upward edge of $c_1 = (x, x')$ is on the path between x and u. Clearly, c_1 and c_2 have no common upward edges.

Case 2 $[\ell(x, x') < \ell(x, y)]$**:** Here $\ell evel(u) < \ell evel(v)$, and hence u is a descendant of v (see Figure 3.7(b)). All upward edges of $c_1 = (x, x')$ are confined to the subtree $\mathcal{T}''$ rooted at u. Since $\ell(v) > \ell(u)$, node y lies outside subtree $\mathcal{T}''$. Consequently, all

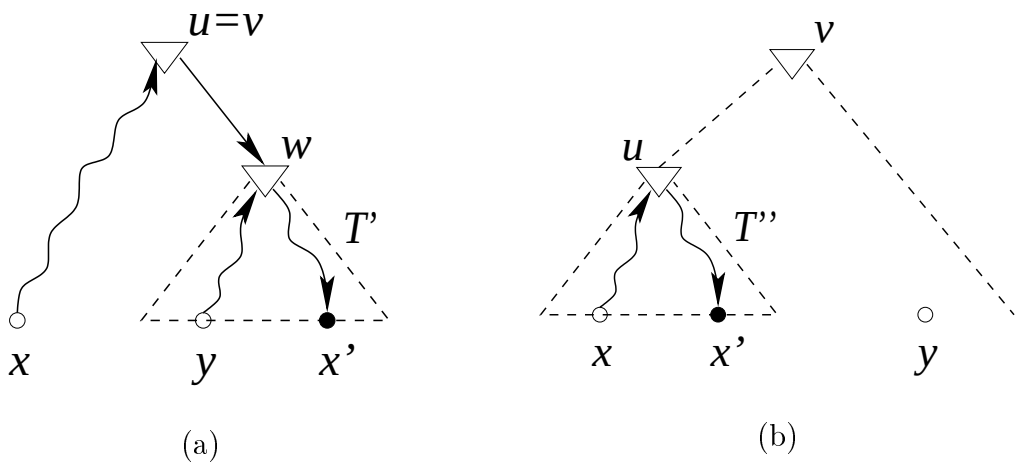


Figure 3.7: Illustration of the proof of Lemma 3.4

upward edges of (y, y') lie outside $\mathcal{T}''$. In any case, communications c_1 and c_2 have no common upward edges.

By an analogous argument we can use the fact that $\ell(x, x') \leq \ell(x', y')$ to establish that communications c_1 and c_2 have no common downward edges. Thus, the CST can accommodate communications c_1 and c_2 simultaneously. ■

An obvious consequence of Lemma 3.4 is the following result.

Corollary 3.5 *For any $k \geq 1$, the CST can simultaneously accommodate a set C of k one-to-one communications if and only if for any two communications $(x, x'), (y, y') \in C$, sets $\{x, y\}$ and $\{x', y'\}$ are not incompatibles.* ■

We now formalize the notion of scheduling a width-w communication set in w steps.

Definition 3.4 A set, C, of communications with width w is *width partitionable* if and only if

(a) C has only one communication, or

(b) C satisfies the following two conditions:

(i) There exists a set $C_1 \subseteq C$ such that C_1 has width 1.

(ii) The set $C - C_1$ has width $w - 1$ and is width partitionable. ■

Remarks: For the recursive definition, the singleton set C forms the base case. For width-w set C (where $w > 1$), the set C_1 consists of a set of communications that can be scheduled in one step such that $C - C_1$ has width $w - 1$. A similar width-1 subset $C_2 \subseteq C - C_1$ reduces the width of $C - C_1 - C_2$ to $w - 2$. Thus, C can be partitioned into w subsets $C_1, C_2, \cdots, C_w$, each of width-1. Since the width of C is w, no fewer than w blocks are possible in this partition; in that sense it is width partitionable. Implicitly, the above definition also specifies a w-step schedule for an width partitionable set of communications with width w. Note that while the schedule $C_1, C_2, \cdots, C_w$ prescribes the communications that can be performed simultaneously, the order in which communications of sets $C_1, C_2, \cdots, C_w$ are performed is not important.

Theorem 3.6 *The CST can schedule the communications from a width partitionable set in w steps if and only if the width of the set of communications is at most w.*

In the next two sections we discuss width partitionable communication sets. Chapter 4 deals with sets that are not width partitionable.

3.2 Communication Sets with Disjoint Incompatibles

In this section we consider a particular class of communication sets and prove them to be width partitionable. Consequently, Theorem 3.6 provides a tight bound on the time for scheduling communications from this class on the CST.

Definition 3.5 A set C of communications has *disjoint incompatibles* if and only if no source or destination appears in more than one incompatible. ■

Figure 3.2 shows an example of a communication set with disjoint incompatibles. Consider any width-w set, C, of communications with disjoint incompatibles. Clearly, every subset of C also has disjoint incompatibles. To prove that C is width partitionable (when the width of C is greater than 1), we only need show the existence of $C_1 \subseteq C$ that includes exactly one source or destination for each maximum incompatible. We now show that this set C_1 can be constructed.

Broadly speaking, the idea is to represent the incompatibles of C as nodes of a graph and the communications themselves as edges in this graph. The task of selecting a set of communications (edges of this graph), subject to the restrictions in the definition of a width partitionable set, will be shown to be that of finding a matching in the graph.

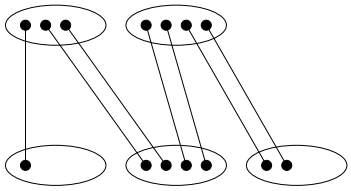
To help us along with the proof, we first add some dummy communications to C. To each source (resp., destination) incompatible, I, of C, add $w - |I|$ dummy sources (resp., destinations); $|I|$ denotes the number of elements in set I. If the number of source and destination incompatibles is different, then add dummy source or destination incompatibles, each with w dummy elements, so that the number of source and destination incompatibles is the same. Since C has an equal number of sources and destinations, the number of dummy sources and destinations added is equal. Pair each dummy source with a dummy destination to form a dummy one-to-one communication. (It is not important for the pairing to ensure that the dummy communications satisfy the membership requirements of their incompatibles.)

Figure 3.8(a) shows an example of a communication set with disjoint incompatibles. Figure 3.8(b) shows the set after adding the dummy communications.

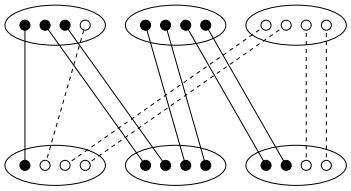
Let the augmented set of communications be $\widehat{C}$. Clearly, $\widehat{C}$ has an equal number of source and destination incompatibles, each of size w. We now represent $\widehat{C}$ as a bipartite graph and show that it has a complete matching from the source incompatibles to the destination incompatibles. Subsequently, we will use this to construct $C_1 \subseteq C$ that includes exactly one source or destination for each maximum incompatible of C.

Let $\widehat{I} = \{\alpha_1, \alpha_2, \cdots, \alpha_z\}$ and $\widehat{J} = \{\beta_1, \beta_2, \cdots, \beta_z\}$ be the sets of source and destination incompatibles of $\widehat{C}$. Construct a bipartite graph $\widehat{\mathcal{G}} = (\widehat{I} \cup \widehat{J}, E)$ with an edge (α_i, β_j) iff there is a communication whose source is in α_i and whose destination is in β_j. Figure 3.8(c) shows a bipartite graph of the communication set of Figure 3.8(b).

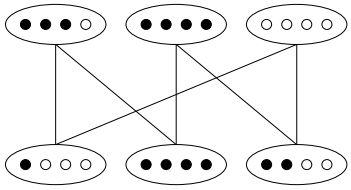
A *matching* on a bipartite graph is a subset of its edges so that no two selected edges share a common vertex. A matching is *complete* if for every vertex v of the graph, the matching includes an edge incident on v. The following result is well known.



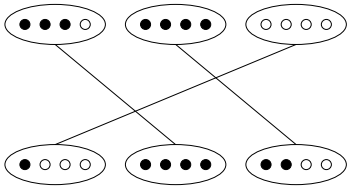
(a) Original communication set



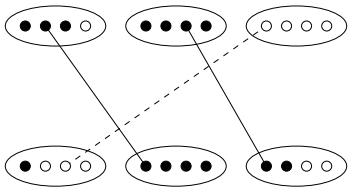
(b) Adding dummy communications



(c) A bipartite graph



(d) A complete matching



(e) The set C_1

Figure 3.8: Constructing set C_1 for a set with disjoint incompatibles

Theorem 3.7 (Hall's Theorem) [1, p. 667]: *A bipartite graph with vertex set* $V = A \cup B$ *has a complete matching if and only if for each* $Q \subseteq A$, $|Q| \leq |R(Q)|$, *where* $R(Q) \subseteq B$ *is that subset of* B *with edges to elements of* Q. ■

We now use Hall's Theorem to prove that the graph $\widehat{\mathcal{G}} = (\widehat{I} \cup \widehat{J}, E)$ has a complete matching. Select any $Q \subseteq \widehat{I}$ with $|Q| = q$. Since each of the q incompatibles of Q has w elements, Q represents wq communications (both real and dummy). These wq communications are spread over at least q destination incompatibles, as each destination incompatible holds only w elements. Therefore $|R(Q)| \geq q = |Q|$. Thus

$\widehat{\mathcal{G}}$ has a complete matching $Q \subseteq E$ (Theorem 3.7). Figure 3.8(d) shows one possible matching between nodes of Figure 3.8(c).

Let $M = \{(\alpha_i, \beta_{f(i)}) \ : \ 1 \leq i \leq z\}$ be a complete matching of graph $\widehat{\mathcal{G}}$. Construct set $C_1 \subseteq C$ as follows. If α_i is a maximum source incompatible of C, then it contains no dummy elements. Consequently, there is a (non-dummy) communication (x, x') with $x \in \alpha_i$ and $x' \in \beta_{f(i)}$. For each such α_i, select (x, x') to be in C_1. Similarly, if $\beta_{f(i)}$ is a maximum destination incompatible of C, then there is a (non-dummy) communication (y, y') with $y \in \alpha_i$ and $y' \in \beta_{f(i)}$. Again for each such $\beta_{f(i)}$, select (y, y') to be in C_1. No other communication is selected to be in C_1. Figure 3.8(e) shows the communications included in the set C_1.

Clearly, each communication of C_1 has its source and destination in a different incompatible; this is because their selection is based on a matching on a graph with incompatibles as vertices. Also for each maximum incompatible, there is a communication in C_1 whose source or destination is in the maximum incompatible. This proves that every set of communications with disjoint incompatibles is width partitionable. With Theorem 3.6 we have the following result.

Theorem 3.8 *The CST can schedule communications from a set with disjoint incompatibles in w steps if and only if the width of the set of communications is at most w.* ■

3.3 Sets with Overlapping Incompatibles

In the last section we proved that communication sets with disjoint incompatibles are width partitionable. In general, however, two incompatibles may overlap (have some common elements). In this section we consider some important classes of communication sets whose incompatibles need not be disjoint. We first define three conditions that establish building blocks for constructing complex communication sets. Subsequently, we use these conditions to construct two important classes of communication sets, that we then prove to be width partitionable.

In this section we restrict our discussions to communication sets that are "oriented". For the purpose of our discussion, number the sources and destinations

(a) (b)

Figure 3.9: Examples of oriented (a) and non-oriented (b) communication sets

(leaves of the CST) in ascending order from left to right. Thus we may say that for two leaves x, y, $\quad x < y$ to mean that x is to the left of y, or $x \leq y$ to mean that x is not to the right of y.

Definition 3.6 A communication set C is oriented if and only if either (i) for every communication $(x, x') \in C, \quad x < x'$ or (ii) for every communication $(x, x') \in C, \quad x > x'$. ■

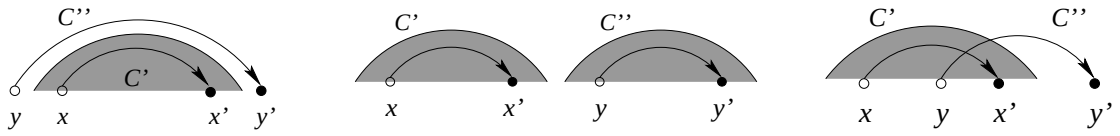
Remark: Where there is no ambiguity we will drop the attribute "oriented" for communication sets in this section.

Figure 3.9(a) shows example of an oriented communication set as each source is to the left of its destination. The communication set of Figure 3.9(b) is not oriented. Considering oriented communication sets greatly simplifies the discussion without giving up too much in the generality of the results. (This is because every non-oriented communication set is a union of at most two oriented communication sets.) Without loss of generality, we assume each communication to have its source to the left of its destination at the leaves of the CST (as in Figure 3.9(a)).

3.3.1 Combining Communication Sets

The combination of two communication sets is simply their union. When combined under certain conditions, the resulting communication set can be proved to have some useful properties. In this section we identify three such conditions termed *capping*, *concatenation*, and *interleaving*. Subsequently we use these conditions to express two useful classes of communications called well-nested sets and monotonic sets, and prove these sets to be width partitionable.

We now describe the conditions referred to above. Although we apply these conditions to communications oriented from left to right, they can be adapted to sets



(a) Capping (b) Concatenation (c) Interleaving

Figure 3.10: Illustration of conditions for combining communication sets

oriented from right to left and to non-oriented sets. For the following definitions let C' and C'' be communication sets and let $C = C' \cup C''$.

Capping: The *capping condition* (see Figure 3.10(a)) requires that

$$C'' = \{(y, y')\} \text{ and for all } (x, x') \in C', \quad y < x \text{ and } x' < y'. \tag{3.2}$$

Remark: Since $x < x'$, the capping condition implies that $y < x < x' < y'$. Intuitively, the capping condition requires the singleton set C'' to span across (or cap) the entire set C'.

To emphasize that sets C' and C'' satisfy the capping condition we will express set $C = C' \cup C''$ as $C = cap(C', C'')$. Since $C'' = \{c\}$, a singleton element set, we will write $cap(C', C'')$ simply as $cap(C', c)$ (rather than $cap(C', \{c\})$.

Concatenation: The *concatenation condition* (see Figure 3.10(b)) requires that

$$\text{for all } (x, x') \in C', (y, y') \in C'', \quad x' < y. \tag{3.3}$$

Remark: Again since $x < x'$ and $y < y'$, the concatenation condition implies that $x < x' < y < y'$. Intuitively, all communications of C' are to the left of those of C''.

Here we write $C = C' \cup C'' = concat(C', C'')$. In general, $|C''| \geq 1$. However, if $C'' = \{c\}$, a single element set, then we write $concat(C', \{c\})$ as $concat(C', c)$.

Interleaving: The *interleaving condition* (see Figure 3.10(c)) requires that

$$C'' = (y, y'), \text{ and } \forall (x, x') \in C', \quad x < y \text{ and } x' < y', \text{ and } \exists (z, z') \in C', \quad y < z'. \tag{3.4}$$

Remark: Here for some (z, z'), $\quad z < y < z' < y'$. Intuitively, C' is to the left of C'' but not entirely. At least one destination of C' is to the right of y.

Here we write $C = C' \cup C'' = inter(C', C'')$. As in the capping condition, we will only use singleton sets for C'' and use $inter(C', c)$ to mean $inter(C', \{c\})$.

The restriction that $|C''| = 1$ for the capping and interleaving conditions is necessary for properly defining the well-nested and monotonic sets later in this section. For a different setting however, our definitions of the conditions could be generalized to $|C''| \geq 1$.

We now show how the conditions identified above could be used to define classes of communication sets. We consider communication sets that satisfy a set of conditions. Although some definitions are made in a more general setting, we implicitly assume the conditions to be one of capping, concatenation, and interleaving discussed above. Again we will use the symbol $\sigma(C', C'')$ to denote that the communication set $C' \cup C''$ (obtained from sets C' and C'') emphasizing the fact that $C' \cup C''$ satisfies condition σ, where $\sigma \in \{cap, concat, inter\}$.

Definition 3.7 Inductively, define a communication set satisfying a set S of conditions as follows.

(a) A set with only one communication satisfies S.

(b) If sets C' and C'' satisfy S, then for each condition $\sigma \in S$, set $\sigma(C', C'')$ satisfies S.

(c) No other communication set satisfies S.

The above definition allows a communication set C to be constructed inductively using only the conditions of S; all conditions of S need not be used, however. The sequence of conditions applied to construct a communication set gives a (*condition*) *expression* for C. For example the expression for the communication set in Figure 3.11

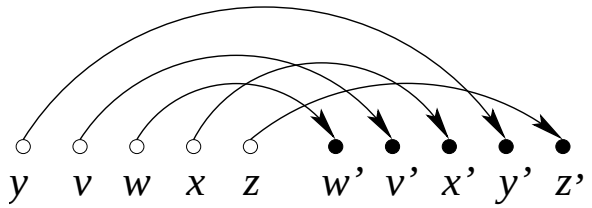


Figure 3.11: A communication set satisfying the set $\{cap, inter\}$

is $inter(cap(inter(cap(c_1, c_2), c_3), c_4), c_5)$ where $c_1 = (v, v')$, $c_2 = (w, w')$, $c_3 = (x, x')$, $c_4 = (y, y')$, $c_5 = (z, z')$. Note that this expression is not necessarily unique. (That is, an expression may have different conditions in different order but the set of conditions used is always the same.) We can think of a condition set S as inducing a class $\mathcal{C}$ of communication sets such that each $C \in \mathcal{C}$ can be constructed as described above.

Lemma 3.9 *Let S_1, S_2 be sets of conditions inducing classes $\mathcal{C}_1, \mathcal{C}_2$ of communication sets. If $S_1 \subseteq S_2$, then $\mathcal{C}_1 \subseteq \mathcal{C}_2$.*

Proof: Since every condition $\sigma \in S_1$ is also in S_2, then it is always possible to construct the class $\mathcal{C}_1$ of communications using the set of conditions S_2. Consequently, $\mathcal{C}_1 \subseteq \mathcal{C}_2$. ■

Let the *base set of conditions* of a communication set C be the smallest set of conditions that it satisfies. Observe that the three conditions described earlier in this section combine two communication sets with different relative placement of communications (see Figure 3.10). That is, one cannot replace another. For example, if C' containing (x, x') and $C'' = \{c\} = \{(y, y'\})$ satisfy the capping condition, then $y < x < x' < y'$. This implies that they cannot satisfy the concatenation condition (requiring $x < x' < y < y'$) or the interleaving condition (requiring $x < y < x' < y'$, assuming (x, x') to satisfy the last clause of Equation 3.4. Thus capping cannot be replaced by concatenation or interleaving. Similar assertions can be made about concatenation and interleaving. Thus we have the following lemma.

Lemma 3.10 *If a communication set satisfies a set of conditions, then it has a unique base set of conditions.* ■

We now consider oriented communication sets that can satisfy various subsets of the set $\{cap, concat, inter\}$ and examine whether these sets are width partitionable (see Definition 3.4).

We first prove that there exists a communication set satisfying $\{cap, inter\}$ that is not width partitionable. This implies that a set satisfying $\{cap, concat, inter\}$ is not width partitionable as well. Next in Lemma 3.12 we prove that a communication set satisfying $\{concat\}$ is width partitionable. We use this result in Sections 3.4 and 3.5 to establish that sets satisfying $\{cap, concat\}$ (called well-nested sets) or $\{concat, inter\}$ (called monotonic sets) are width partitionable. These results, with Lemma 3.12, imply a communication satisfying only one of the three conditions in $\{cap, concat, inter\}$ is width partitionable. Table 3.1 summarizes these results.

Table 3.1: Width partitionability of communication sets satisfying conditions from {*cap*, *concat*, *inter*}

Base condition set	Width Partitionable?	Remarks
$\{cap, concat, inter\}$	No	implied by Lemma 3.11
$\{cap, concat\}$	Yes	well-nested set; Section 3.4
$\{cap, inter\}$	No	implied by Lemma 3.11
$\{concat, inter\}$	Yes	monotonic set; Section 3.5
$\{cap\}$	Yes	implied by Theorem 3.15
$\{concat\}$	Yes	implied by Corollary 3.13
$\{inter\}$	Yes	implied by Theorem 3.17

Lemma 3.11 *There exists a communication set satisfying* $\{cap, inter\}$ *that is not width partitionable.*

Proof: Consider the set $C = \{c_1, c_2, c_3, c_4, c_5\}$ of communications in Figure 3.11 where $c_1 = (v, v'), c_2 = (w, w'), c_3 = (x, x'), c_4 = (y, y'), c_5 = (z, z')$. This communication set of Figure 3.11 has the condition expression $inter(cap(inter(cap(c_1, c_2), c_3), c_4), c_5)$ and its base set of conditions is $\{cap, inter\}$. By Lemma 3.10, this set of conditions is unique; that is, the communication set cannot be constructed with a different set of conditions. Figure 3.12 shows one possible mapping of these communications on

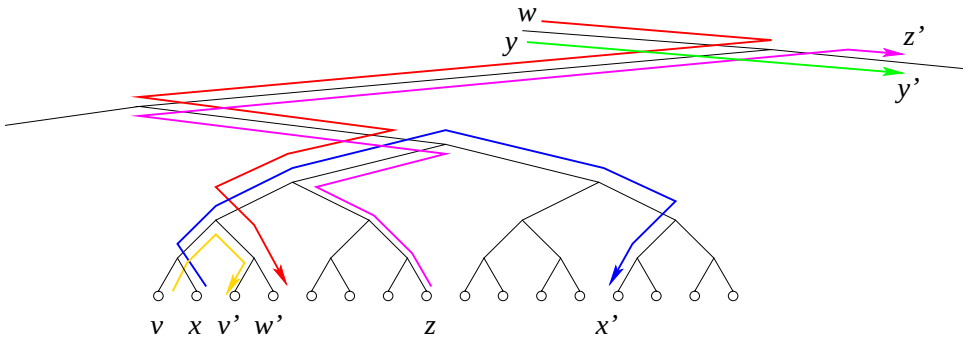


Figure 3.12: Mapping communication set of Figure 3.11 on the CST

a CST and Figure 3.13 shows the incompatibility graph of this communication set. Clearly, the communication set has width 2. As in Figure 3.6, the only communication that can be scheduled with (x, x') is either (w, w') or (y, y'). Since $C - \{(x, x'), (w, w')\}$ or $C - \{(x, x'), (y, y')\}$ has width 2, it follows that C cannot be scheduled in two steps. ■

We close this section with the following results that are used in subsequent sections.

Lemma 3.12 *For any width partitionable, oriented communication sets C' and C'' of widths w_1 and w_2, respectively, $concat(C', C'')$ is a width partitionable communication set of width $max\{w_1, w_2\}$.*

Proof: If C' and C'' share a common portion of a CST, then it must be at the right (destination) end of C' and the left (source) end of C'' (see Figure 3.10(b)). Since the CST has full duplex links, upward edges from sources will not conflict with downward edges to destinations. That is, the sets C' and C'' do not interfere with each other. Consequently, $concat(C', C'')$ is width partitionable and has width $max\{w_1, w_2\}$.

Corollary 3.13 *Every communication set that satisfies the set $\{concat\}$ is width partitionable.* ■

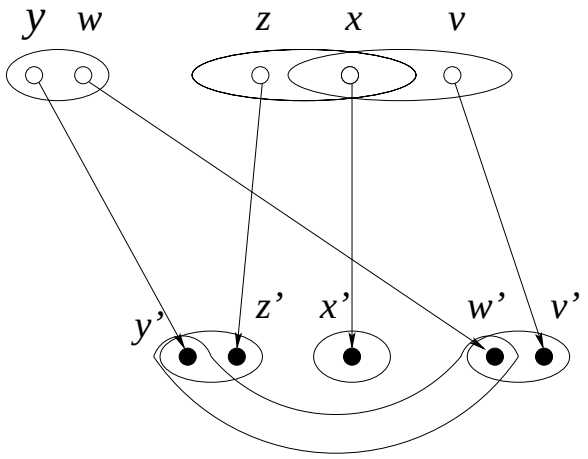


Figure 3.13: The incompatibility graph of the communication set of Figure 3.12

3.4 Well-Nested Communication Sets

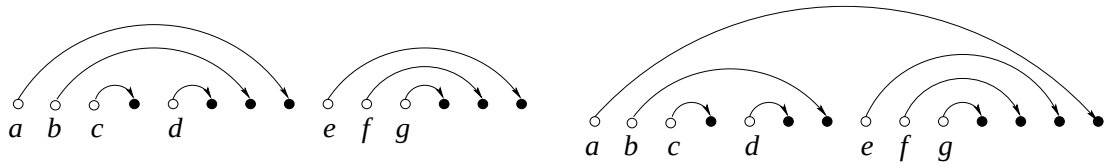
In this section we define a class of communications called "oriented well-nested (communication) sets." A special case of a well-nested set is the set of communications on a segmentable bus [48], a fundamental structure of reconfigurable computing.

Definition 3.8 An oriented well nested communication set is one that satisfies the conditions in the set $\{cap, concat\}$. ■

Figure 3.14 shows examples of oriented well-nested sets and their condition expressions. The well-nested set derives its name from its similarity to a well-nested parenthesis sequence. If each source (resp., destination) in a well-nested set is replaced by a "(" (resp., ")"), then the resulting sequence is a well-nested (parenthesis) sequence. For example the communication set of Figure 3.14(a) can be represented as the well-nested sequence ((( )( ))) ((( ))).

Definition 3.9 The *depth* of an oriented well-nested set can be defined inductively as follows:

(a) The depth of a singleton communication set is 1.



(a) $concat(cap(cap(concat(c,d),b),a), cap(cap(g,f),e))$ of depth 3

(b) $cap(cap(concat(c,d),b), cap(cap(g,f),e),a)$ of depth 4

Figure 3.14: Examples of well-nested sets, with the corresponding condition expressions and parentheses depths. The letter next to each source represents the entire communication.

(b) If C is an oriented well-nested set of depth d, then $cap(C,c)$ has depth $d+1$.

(c) If C' and C'' are well-nested sets of depths d_1 and d_2, then $concat(C',C'')$ has depth $\max\{d_1,d_2\}$. ■

For brevity, we will use the terms well-nested sequence and well-nested set interchangeably. Recall that we consider only sets of communications that are oriented from left to right. The condition expression may not be unique. For example the condition expressions $concat(a, concat(b,c))$ and $concat(concat(a,b),c)$ represent the same set of communications. However, the order in which the conditions apply is unique.

Let C be an oriented well-nested set. Let $(x,x') \in C$ be the communication with the leftmost source x; i.e. for all $(y,y') \in C$, $x \leq y$. If x' is also the rightmost destination (i.e., for all $(y,y') \in C$ $x' \geq y'$), then C is *terminally capped*. In other words, a terminally capped set has an *outermost* communication (x,x').

Lemma 3.14 *Let C_1 be a terminally capped oriented well-nested set with an outermost communication c'. Let c'' be any communication such that C_1 and c'' satisfy the capping condition. Then for any communication $c \in C_1$, the sources (or destinations) of c and c'' are in the same incompatible only if the sources (or destinations) of c' and c'' are in the same incompatible.*

Proof: If $|C_1| = 1$, then $C_1 = \{c'\}$ and there is nothing to prove. So assume that $C_1 = cap(C_2,c')$, $c = (x,x'), c' = (y,y')$ and $c'' = (z,z')$. Clearly, $z < y < x < x' <$

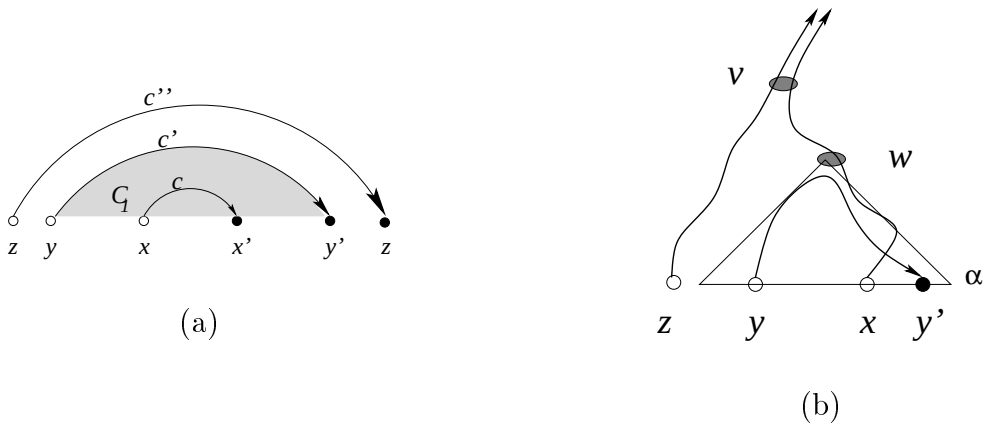


Figure 3.15: Illustration of Lemma 3.14

$y' < z'$ (see Figure 3.15(a)). Let $\ell ca(x, z) = v$ and $\ell ca(y, y') = w$. Suppose that x and z are in the same incompatible but y and z are not (see Figure 3.15(b)). This requires that $\ell evel(w) < \ell evel(v)$. Let α be the rightmost node of the subtree rooted at w. So, $y' \leq \alpha$ and $x' > \alpha$ which contradicts the fact that $x' < y'$. ■

Theorem 3.15 *Every oriented well-nested set is width partitionable.*

Proof: We proceed by induction on the depth of the well-nested set. A depth-1 well-nested set satisfies the set $\{concat\}$. By Corollary 3.13, it is width partitionable. Assume the theorem to hold for any well-nested set with depth at most δ and consider an oriented set C with depth $\delta + 1$. We have two cases corresponding to (a) $C = concat(C', C'')$ and (b) $C = cap(C', c)$, the two parts in the recursive definition of an oriented well-nested set.

Suppose that $C = concat(C', C'')$, for well-nested sets C' and C'' of widths w_1 and w_2, respectively. By Lemma 3.12, C is of width $\max\{w_1, w_2\}$ and can be scheduled on the CST in $\max\{w_1, w_2\}$ steps. That is, C is width partitionable.

Suppose now that $C = cap(C', c)$, where C' is an oriented set of width w_1. Let w denote the width of C. Observe that C' has depth δ, so the induction hypothesis applies to it. Recall that proving a width-w communication set width partitionable is tantamount to scheduling its communications in w steps. If $w = w_1 + 1$, then schedule C' in w_1 steps and communication c all by itself in another step.

Suppose $w = w_1$. Here we only need identify a width-1 set C_1 such that $C - C_1$ has width $w - 1$. Let C' be the concatenation of C'_1, C'_2, $\cdots$, C'_α, for some integer $\alpha \geq 1$ (see Figure 3.16) where for $1 \leq i \leq \alpha$, C'_i is a terminally capped well-nested set. Let the outermost communication of C'_i be $c_i = (x_i, x'_i)$. Clearly C'_i has depth $\leq \delta$ and the induction hypothesis applies to it. Thus C'_i has an optimal schedule. Let S_i be the set of all communications that are scheduled at the same step as communication c_i in this optimal schedule.

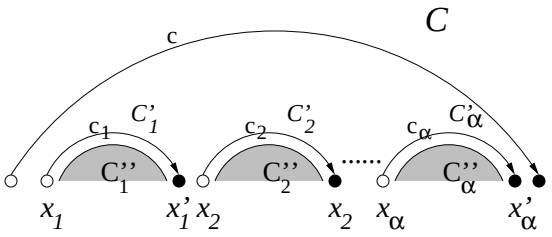


Figure 3.16: Illustration of the proof of Theorem 3.15

Define set $C_1 = \bigcup_{i=1}^{\alpha} S_i$. By the argument used in Lemma 3.12, width of C_1 = $\max\{\text{width of } S_i : 1 \leq i \leq \alpha\}$. Since each S_i has width 1, C_1 has width 1. By the induction hypothesis, the set $C' - C_1$ has width $w - 1$. Set $C - C_1$ also has width $w - 1$ for the following reason. Suppose c and c_i are in the same incompatible, then an element of this incompatible has already been included in C_1. If c is not in the same incompatible as any of the c_i's, then by Lemma 3.14, c cannot be incompatible with any communication in the set C'. ■

3.5 Monotonic Communication Sets

In this section we consider another class of communication sets called "oriented monotonic sets" (see Figure 3.17) and prove it to be width partitionable. This class has many important communication sets including those of the uniform (or normal) hypercube (in which only one dimension of the hypercube is used for communications).

Definition 3.10 An oriented monotonic communication set is one that satisfies the conditions in the set $\{concat, inter\}$. ■

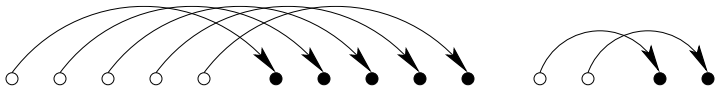


Figure 3.17: Monotonic communication set

The monotonic set derives its name from the nature of its incompatibility graph. We provide more details at the end of this section.

Definition 3.11 The *breadth* of an oriented monotonic set can be defined recursively as follows.

(a) The breadth of a singleton communication set is 1.

(b) If C is an oriented monotonic set of breadth b, then $inter(C, c)$ has breadth $b + 1$.

(c) If C' and C'' are monotonic sets of breadth b_1 and b_2, then $concat(C', C'')$ has breadth $\max\{b_1, b_2\}$. ■

As for oriented well-nested set, the condition expression of an oriented monotonic set is unique.

Lemma 3.16 *Let C be an oriented monotonic set such that either $C = \{c'\}$ or it is possible to express C as $inter(C - \{c'\}, c')$ for some monotonic set $C - \{c'\}$ and communication c'. Let c'' be any communication such that C and c'' satisfy the interleaving condition. Then for any communication $c \in C$, the sources (or destinations) of c and c'' are in the same incompatible only if the sources (or destinations) of c' and c'' are in the same incompatible.*

Proof: The proof is similar to the proof of Lemma 3.14. If $|C| = 1$, there is nothing to prove. So assume that $C = inter(C - \{c'\}, c')$, $c = (x, x')$, $c' = (y, y')$ and $c'' = (z, z')$. Clearly $x' < y'$ (see Figure 3.18(a)). Here (y, y') is the rightmost communication of C. Let $\ell ca(x, z) = v$ and $\ell ca(y, y') = w$. Suppose that z is incompatible with x but not incompatible with y (see Figure 3.18(b)). This requires that $\ell evel(w) < \ell evel(v)$. Let α be the rightmost node of the subtree rooted at w. So, $y' \leq \alpha$ and $x' > \alpha$ which contradicts the fact that $x' < y'$. ■

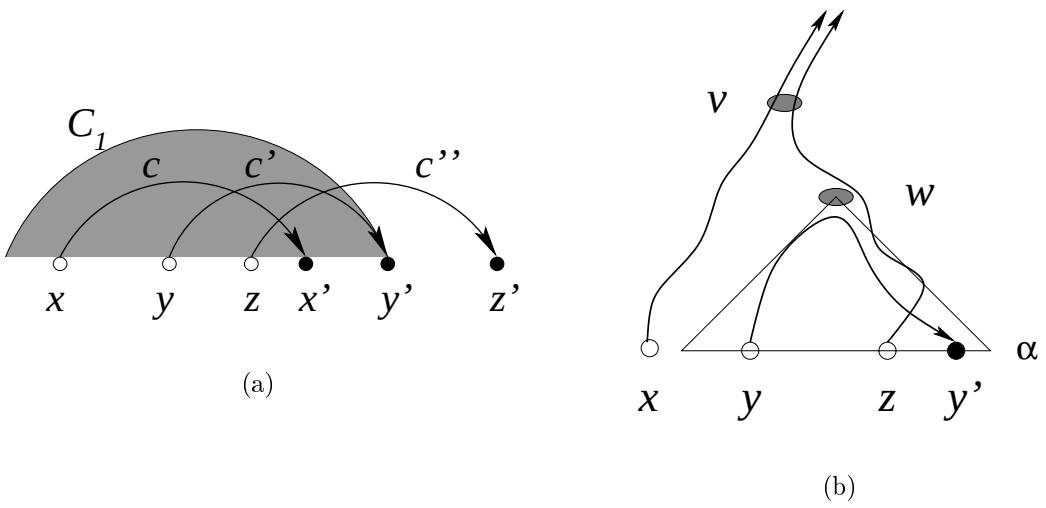


Figure 3.18: Illustration of the proof of Lemma 3.16

Theorem 3.17 *Every oriented monotonic set is width partitionable.*

Proof: The proof follows the same lines as the proof of Theorem 3.15. We proceed by induction on the breadth of the oriented monotonic set. One step suffices for a breadth-1 monotonic set. Assume the theorem to hold for a monotonic set with breadth at most β and consider a monotonic set C with breadth $\beta + 1$. We have two cases corresponding to: (a) C can be expressed as $concat(C', C'')$ and (b) C cannot be expressed as $concat(C', C'')$.

As described before, if $C = concat(C', C'')$, then C can be scheduled in $\max\{w', w''\}$ steps and it is width partitionable (where w', w'' are the widths of C', C'' respectively).

If C cannot be expressed as $concat(C', C'')$ then $C = inter(C', c)$, where C' is a monotonic set of width w' (see Figure 3.19). Let w denote the width of C. Observe that C' has breadth β, so the induction hypothesis applies to it.

If $w = w' + 1$, then schedule C' in w' steps and then schedule the communication c all by itself in another step. Suppose $w = w'$. Here we only need identify a width-1 set C_1 such that $C - C_1$ has width $w - 1$. Let the rightmost communication of C' be $c' = (y, y')$. Clearly C' has breadth at most β and the induction hypothesis applies to it. Thus, C' has an optimal schedule. Let S' be the set of all communications of C' that are scheduled at the same time as communication c'.

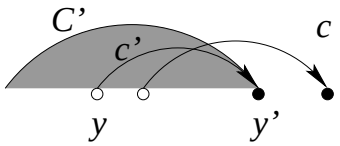


Figure 3.19: Illustration of the proof of Theorem 3.17

Define set $C_1 = S'$. Clearly, C_1 has width 1. By the induction hypothesis the set $C' - C_1$ has width $w - 1$. Set $C - C_1$ also has width $w - 1$ for the following reason. Suppose c and c' are in the same incompatible, then an element of this incompatible has already been included in C_1. If not, then, by Lemma 3.16 c cannot be in the same incompatible as any communication in the entire set C'. ■

Now we show that the incompatibility graph of an oriented monotonic communication set has a special property from which its name derives.

Definition 3.12 An *ordered incompatibility graph* of an oriented communication set is one in which the sources and destinations are arranged in increasing order of their indices. ■

For example, Figure 3.6 (page 36) shows an ordered incompatibility graph if $e < d < a < b < c$ and $e' < a' < b' < d' < c'$.

Definition 3.13 An ordered incompatibility graph of an oriented communication set is *parallel* if and only if, for all communications $(x, x'), (y, y'),$ if $x < y$, then $x' < y'$. ■

Intuitively, if the ordered incompatibility graph of an oriented communication set is not parallel, then it has edges that intersect (see Figure 3.20).

Theorem 3.18 *A communication set is oriented monotonic if and only if its ordered incompatibility graph is parallel.*

Proof: Let C be an oriented monotonic set and let $\mathcal{G}$ be its incompatibility graph. We now prove that $\mathcal{G}$ is parallel.

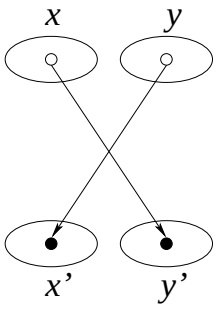


Figure 3.20: An ordered incompatibility graph that is not parallel

We proceed by induction on $|C|$. Clearly, if $|C| = 1$ then $\mathcal{G}$ is parallel. Assume the lemma to hold for any oriented monotonic communication set of size at most n and consider the case where $|C| = n + 1$. Let $C = \sigma(C', C'')$, where $\sigma \in \{concat, inter\}$. Both C' and C'' are of size at most n and the induction hypothesis applies to them (i.e., their incompatibility graphs are parallel). Regardless of the identity of σ, each source of C' is to the left of all source(s) of C'' and each destination of C' is to the left of all destination(s) of C''. Therefore, edges of the ordered incompatibility graphs of C' and C'' (when placed next to each other) do not intersect. That is, C is parallel (see Figure 3.21).

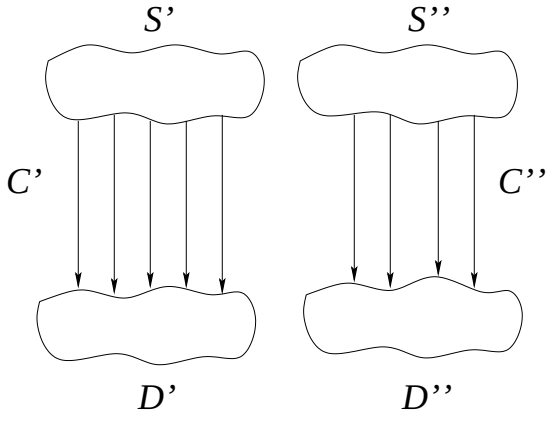


Figure 3.21: Illustration of the proof of the "if" part of Theorem 3.18

Now let $\mathcal{G}$ be a parallel incompatibility graph of an oriented communication set C. We prove that C is monotonic.

We proceed by induction on number of communications in $\mathcal{G}$. Clearly, if $\mathcal{G}$ has one communication, then (by Definition 3.11) C is monotonic. Assume the lemma to hold for any parallel incompatibility graph with at most n communications, and consider the case where the incompatibility graph $\mathcal{G}$ of set C has $n+1$ communications. Let

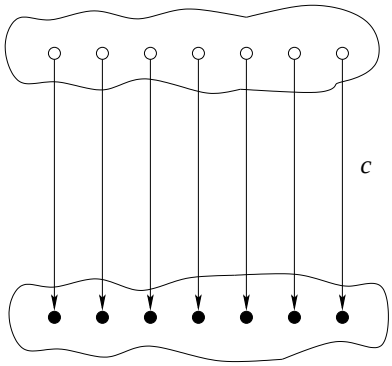


Figure 3.22: Illustration of the proof of the "only if" part of Theorem 3.18

communication c be the rightmost communication of $\mathcal{G}$. The incompatibility graph of set $C - \{c\}$ (see Figure 3.21) has n communications and the induction hypothesis applies to it (i.e. $C - \{c\}$ is monotonic). Since the source of communication c is to the right of all sources of $C - \{c\}$ and the destination of communication c is to the right of all destinations of $C - \{c\}$, then C could be constructed as $\sigma(C - \{c\}, \{c\})$ where $\sigma \in \{concat, inter\}$. Since the set $\{concat, inter\}$ is used to construct oriented monotonic sets, then C is monotonic. ■

3.6 Segmentable Bus

Recall that, functionally, a segmentable bus has the structure shown in Figure 2.4 (page 21). Each processor controls (opens or closes) a segment switch on the bus using local information, creating bus segments connecting consecutive processors. Section 2.2 provides more details. For now assume that at most one processor writes on any bus segment and at most one processor reads from a segment (exclusive

read, exclusive write model). Thus, every pair $(x, x'), (y, y')$ of communications is on a different bus segment; that is, $x, x' < y, y'$. Consider a configuration of the segmentable bus with k segments (numbered $1, 2, \cdots, k$). Let (x_i, x_i') denote the communication (if any) in segment i. Since each communication is confined to a segment of contiguous processors, the set of communications on a segmentable bus form a depth-1, well-nested set. If for each i, $x_i < x_i'$ (or for each i, $x_i > x_i'$), then the well-nested set is oriented and its width is 1. That is, the CST can accommodate these oriented communications in one step (Corollary 3.5). If a well-nested set is not oriented (the set has width at most 2), then we partition it down into two oriented sets, each of width 1, and schedule them in two steps. The width cannot exceed 2 for a well-nested set of depth 1.

So far, we have considered only one-to-one communications. The segmentable bus permits broadcasting on its segments, however. If the source of a broadcast is the leftmost (or rightmost) processor of a segment, then the CST simply connects all switches in the tree that are "below" the path between the source and destination to receive information from one "side" of the switch and transmit it out of the remaining two sides (see Figure 3.23). If the source is in the middle of the segment, then simply schedule two oriented broadcasts, one to the left and the other to the right. Thus we have the following result.

Theorem 3.19 *A CST can schedule the communications of a segmentable bus in at most two steps.* ■

3.7 Concluding Remarks

In this chapter, we have derived the idea of communication width which provides a lower bound on the time to schedule any set of one-to-one communications on the CST. We have identified a property of the communication set, called width partitionability, for which the above lower bound is tight. Then we showed two classes of communication sets to possess this property. As a special case of one of these results, we showed that the set of communications that can be accommodated in one step on a segmentable bus [48] can be scheduled in two steps on the SRGA architecture.

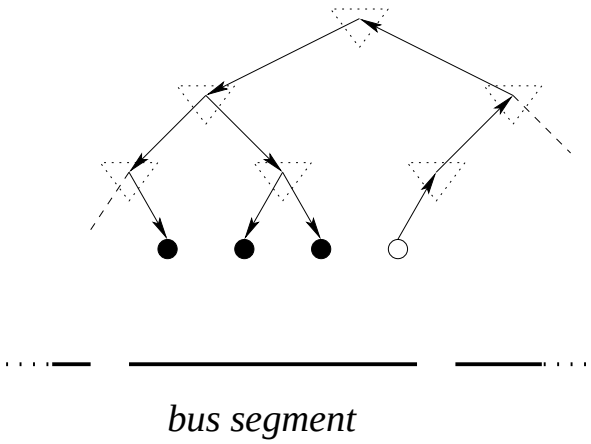


Figure 3.23: Broadcasting on the CST with the source at the right end of a bus segment

The results developed here have a simple generalization to CSTs whose edges correspond to multiple full duplex links (we used only one full duplex link per edge so far). Suppose that there are k_i full duplex links at each edge e_i of the tree, then the "effective width" of a communication set of (actual) width w is max $\{\lceil \frac{w_i}{k_i} \rceil\}$; the quantity w_i is the number of communications traversing edge e_i in any one direction. One interesting case is to use a fat tree [14, 26] where edges between levels l and $l+1$ (where $0 \leq l \leq \log N$) have 2^l full duplex links. Then any set of one-to-one communications can be scheduled in one step.

Another interesting case is when $k_i = k$ for all i, then the effective width=$\{\lceil \frac{w_i}{k} \rceil\}$. For segmentable bus communications setting $k = 2$ makes the effective width 1. That is, if each CST edge has two full duplex links, then a step of the segmentable bus can be accommodated in one step on the CST. As we noted at the start of this chapter, we have only considered the issue of accommodating communications from a set in the CST. Chapter 6 addresses the issue of setting CST switches to actually establish the dedicated paths needed to accomplish the communications. It turns out that

for a segmentable bus the switches can be set at run-time with local information (as required by the functional description of the segmentable bus).

Chapter 4

CST Communication—Sets That Are Not Width Partitionable

In Chapter 3 we showed that some communication classes (oriented well-nested sets and oriented monotonic sets) are width partitionable. In this chapter, we consider communication classes that, in general, are not width partitionable. This study provides a better understanding of the conditions under which a communication set is not width partitionable. First, we show that the incompatibility graph of Figure 4.1 represents one of the simplest communication sets that is not width partitionable. As we explained in Section 3.1, this incompatibility graph is of width 2, but requires three steps on the CST; in other words, the communication set requires one "extra" step beyond its width. We show that the number of extra steps for scheduling a width-w communication set can be as large as $\lceil \frac{w}{4} \rceil$. In Chapter 3 we proved that oriented well-nested sets and oriented monotonic sets are width partitionable. Here, we show that the non-oriented counterparts of these communication sets are, in general, not width partitionable. However, with some restrictions (that still keep the sets non-oriented), these sets are width partitionable.

On the whole, this chapter provides a better understanding of sets that are not width partitionable. Subsequent chapters build only on width partitionable sets (such as those of a segmentable bus.) Consequently, this chapter could be skipped by the reader without loss of continuity.

In the next section we identify a "simplest" communication set that is not width partitionable. Section 4.2 uses the idea of such a simplest set to find an upper bound

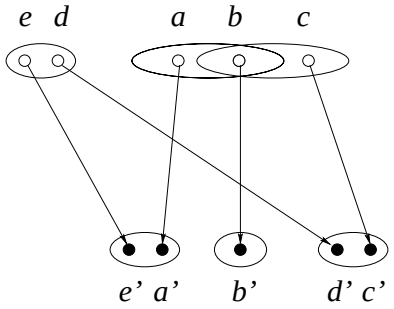


Figure 4.1: Width-2 communication set requiring three steps

on the number of extra steps. Sections 4.3 and 4.4 consider non-oriented well-nested and non-oriented monotonic communication sets.

4.1 The Simplest Communication Sets That Are Not Width Partitionable

In this section we explore the simplest set of requirements that a communication set must have so that it is not width partitionable. The set of requirements we consider includes size (number of communications), width, and number of incompatibles for the communication set. We show that every communication set that is not width partitionable must have at least five communications, at least a width of two, and at least three source incompatibles and three destination incompatibles. This result makes the communication set of Figure 4.1 a "simplest" set that is not width partitionable. Further, we show that there are only two choices (to within isomorphism) for such a simplest set. Figure 4.1 shows one of the two choices.

4.1.1 Requirement of the Simplest Set

We first derive a series of intermediate results that lead to the main result of Theorem 4.11 (page 75). In particular, we prove that a communication set with at most four communications is always width partitionable (see Lemma 4.4 and Theorem 4.6). Theorem 4.10 shows that the communication set must have at least three source in-

compatibles and three destination incompatibles to be not width partitionable. This sequence of arguments establishes the basic requirements for a simplest set that is not width partitionable.

Clearly, reversing the direction of each communication in a communication set C produces a "dual" communication set $\widehat{C}$ and vice versa. This relationship between C and $\widehat{C}$ also carries into the incompatibles. Each source incompatible of C is a destination incompatible of $\widehat{C}$ and vice versa. For brevity, we will derive intermediate results for either C or $\widehat{C}$ but not both. However, it should be understood that they apply to both. For example, Lemma 4.1 talks of overlapping source incompatibles I_1 and I_2 and disjoint destination incompatibles J_1 and J_2. Clearly, the result applies to overlapping destination incompatibles J_1 and J_2 and disjoint source incompatibles I_1 and I_2. To indicate that we are referring to this "dual" result, we will cite the "dual of Lemma 4.1." Similar conventions are adopted for other results.

We organize this section as follows. The first part (Section 4.1.1.1) considers some general results. Section 4.1.1.2 derives the simplest set from the number of communications point of view. Section 4.1.1.3 derives the simplest set from the number of incompatibles point of view. The simplest set from the width point of view is straight forward, so no separate section is devoted to that. We put all these results together in Theorem 4.11.

4.1.1.1 Preliminary Results

In this section we derive some general results that find use in later sections.

Lemma 4.1 *Let (x, x') and (y, y') be two communications in any communication set. Let I_1, I_2 (resp., J_1, J_2) be source (resp., destination) incompatibles such that $x \in I_1 - I_2$, $y \in I_2 - I_1$, $x' \in J_1$ and $y' \in J_2$. If $I_1 \cap I_2 \neq \emptyset$, then $J_1 \cap J_2 = \emptyset$.*

Proof: Let $v \in I_1 \cap I_2$ (see Figure 4.2(a)). Let $\ell ca(x, v) = p$ and let $\ell_1 = \ell evel(p)$. The directed CST link $\langle p, parent(p)\rangle$ is used by communications (x, x') and (v, v') (see Figure 4.2(b)); note that since $x, v \in I_1$ (an incompatible), p cannot be the root of the CST. Let $\ell ca(v, y) = m$ and let $\ell_2 = \ell evel(m)$. Again, the link $\langle m, parent(m)\rangle$ is used by (v, v') and (y, y'). Since $\{x, y\}$ is not an incompatible, communication

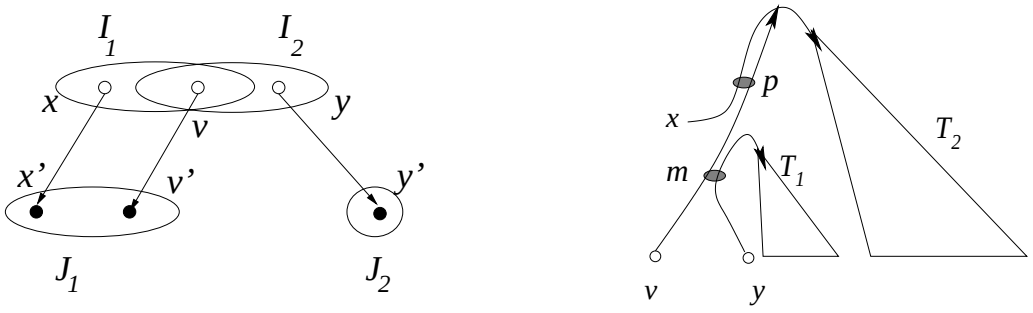


Figure 4.2: Illustration of the proof of Lemma 4.1

(y, y') cannot traverse p and (x, x') cannot traverse m; that is $\ell_1 \neq \ell_2$. Without loss of generality, let $\ell_1 > \ell_2$. This guarantees that the communication originating at y has no upward edges at level ℓ_1 or higher (otherwise $y \in I_1$) (see Figure 4.2(b)). This implies that $\{x', y'\}$ is not an incompatible. Also each destination z' such that $\{y', z'\}$ is an incompatible has to be a leaf of subtree $\mathcal{T}_1$ (see Figure 4.2(b)), whereas each destination w' such that $\{x', w'\}$ is an incompatible is a leaf of $\mathcal{T}_2$. Since $\mathcal{T}_1$ and $\mathcal{T}_2$ have disjoint sets of leaves, then $J_1 \cap J_2 = \emptyset$. ■

Remarks: Intuitively, Lemma 4.1 shows that if two source incompatibles overlap, then their exclusive elements (elements that are not common to both) must have destinations in disjoint destination incompatibles (see Figure 4.2). In effect, a destination incompatible can have edges to only one source incompatible (i.e., the size of a destination incompatible is no more than the size of the source incompatible that has edges to) and we have the following result.

Lemma 4.2 *Let I_1, I_2 be two source incompatibles such that $|I_1 \cap I_2| \geq 2$. For any communications (x, x') and (y, y') such that $x, y \in I_1 \cap I_2$, $x' \in J_1$, $y' \in J_2$, then there are no communications (u, u') and (v, v') such that $u \in I_1 - I_2$, $v \in I_2 - I_1$, $u' \in J_1 - J_2$ and $v' \in J_2 - J_1$.*

Proof outline: If the situation described by the lemma is possible, then Figure 4.3(a) shows that situation. Figure 4.3(b) shows that it is impossible for the situation in Figure 4.3(a) to occur on the CST. ■

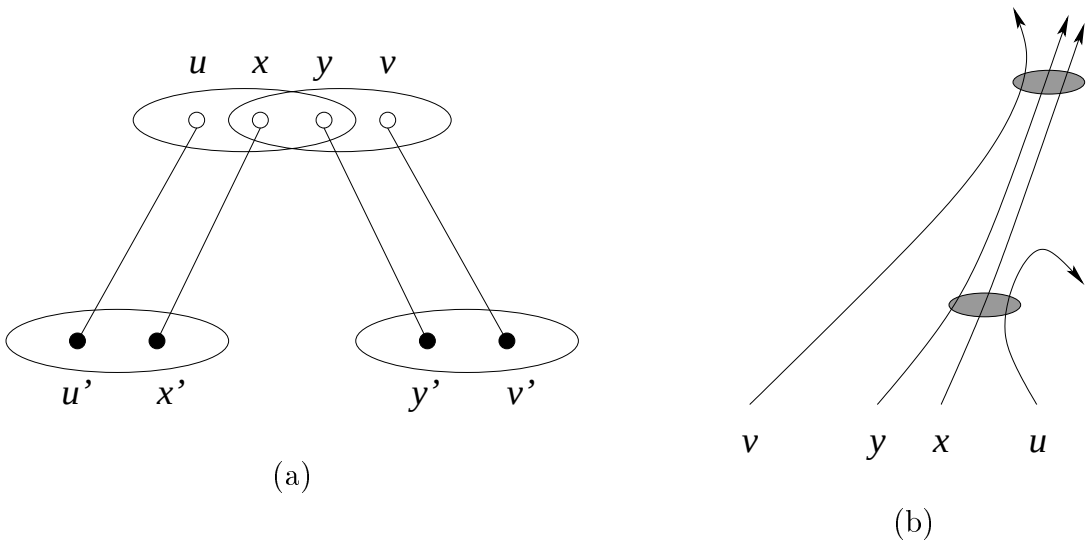


Figure 4.3: Illustration of the proof outline of Lemma 4.2

Lemma 4.3 *Let C be a communication set with at most two maximal source incompatibles I_1 and I_2. If $I_1 \cap I_2 \neq \emptyset$, then C is width partitionable.*

Proof: If C has only one source incompatible, then the width of C is $|C|$; simply schedule the communications one by one.

Let C have two distinct maximal source incompatibles I_1 and I_2 with $I_1 \cap I_2 \neq \emptyset$. Let $x \in I_1 - I_2$, $y \in I_1 \cap I_2$ and $z \in I_2 - I_1$ (see Figure 4.4). Since C has exactly two source incompatibles, $\{x, z\}$ cannot be an incompatible, as this would make $\{x, y, z\}$ an incompatible (Lemma 3.3, page 33), and hence I_1 and I_2 would not be maximal as assumed.

Let x' and z' be the destinations corresponding to sources x and z, respectively. By Lemma 4.1, x' and z' must be in disjoint destination incompatibles J_1 and J_2, respectively (say). This statement holds for any $x \in I_1 - I_2$ and $z \in I_2 - I_1$. Thus, every element of destination incompatible J_1 (resp., J_2) must have its source in I_1 (resp., I_2). That is, if J_1 (resp., J_2) is a maximum incompatible, then so is I_1 (resp., I_2). This, in turn, implies that at least one of I_1 and I_2 must be maximum.

This observation gives us a simple method to schedule C. Simply schedule all the elements in $I_1 \cap I_2$ first. After this we have a communication set with disjoint incom-

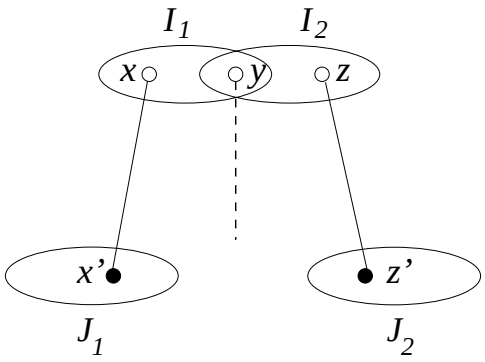


Figure 4.4: Illustration of the proof of Lemma 4.3

patibles, which by Theorem 3.8 is width partitionable. Thus, C is width partitionable. ■

4.1.1.2 Number of Communications in a Simplest Set

In this section we show that a communication set with at most four communications is width partitionable.

Lemma 4.4 *Every set of three communications is width partitionable.*

Proof: For the set $C = \{(x, x'), (y, y'), (z, z')\}$ to be not width partitionable, there must be overlapping source incompatibles and/or overlapping destination incompatibles (Theorem 3.8). Because C has only three communications, either there are only two overlapping source incompatibles or there are only two overlapping destination incompatibles. Without loss of generality, let $\{x, y\}$ and $\{y, z\}$ be incompatibles. By Lemma 3.3, page 33, $\{x, z\}$ can not be an incompatible. By Lemma 4.3 and its dual, C is width partitionable. ■

Lemma 4.5 *Let $C = \{(w, w'), (x, x'), (y, y'), (z, z')\}$ be a communication set. If $\{w, x\}, \{x, y\}$, $\{y, z\}$ and $\{w, z\}$ are source incompatibles, then C has at most two maximal source incompatibles.*

Proof: If $\{w, x, y, z\}$ is an incompatible, there is nothing to prove. So by Lemma 3.3 (page 33) $\{w, y\}$ or $\{x, z\}$ is not an incompatible.

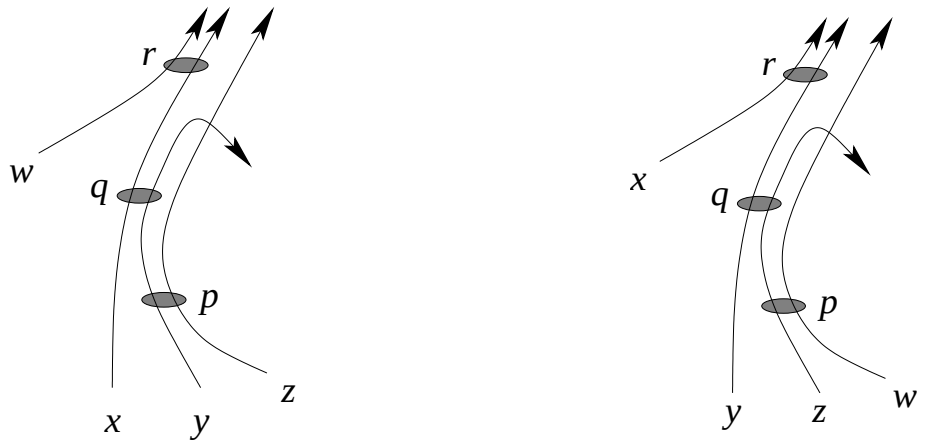


Figure 4.5: Illustration of the proof of Lemma 4.5

Case 1 $\{w, y\}$ is not an incompatible (see Figure 4.5(a)). Since $\{w, y\}$ is not an incompatible, $\ell ca(w, x) = r$ must be at a higher level than $\ell ca(x, y) = q$. Since $\{y, z\}$, $\{w, z\}$ are incompatibles, so must $\{x, y, z\}$ and $\{w, x, z\}$ (see Figure 4.5(a)) of which $\{w, x\}, \{x, y\}, \{y, z\}$ and $\{w, z\}$ are subsets.

Case 2 $\{x, z\}$ is not an incompatible (see Figure 4.5(b)). Since $\{x, z\}$ is not an incompatible, $\ell ca(x, y) = q$ must be at a higher level than $\ell ca(y, z) = p$. Since $\{w, x\}$, $\{w, z\}$ are incompatibles, so must $\{w, y, z\}$ and $\{w, x, y\}$ (see Figure 4.5(b)) of which $\{w, x\}, \{x, y\}, \{y, z\}$ and $\{w, z\}$ are subsets. ■

Theorem 4.6 *Every set of four communications is width partitionable.*

Proof: Consider a communication set, $C = \{(x, x'), (y, y'), (z, z'), (w, w')\}$, that has four communications. Recall that an overlap must exist for the set to be not width partitionable (Theorem 3.8). We have several cases. Subcases within a case are appropriately indented.

Case 1 If C has width 4, then four steps suffice for scheduling (schedule one communication at each step.)

Case 2 If C has width 3, then there exists a 3-element source incompatible and/or a 3-element destination incompatible. We have the following cases.

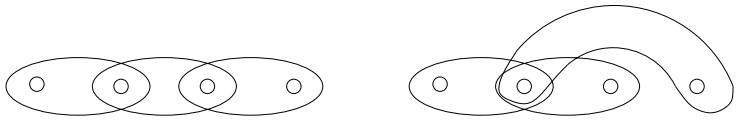


Figure 4.6: Source incompatibles for Subcase 2.3.1 of Theorem 4.6

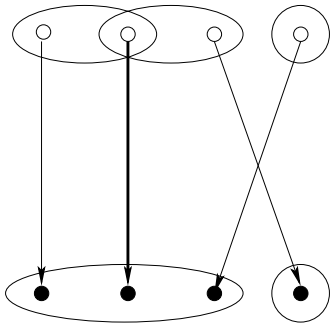


Figure 4.7: Incompatibility graph for Subcase 2.3.2 of Theorem 4.6

Subcase 2.1 Suppose there exists a 3-element source incompatible that overlaps with another maximal incompatible. This implies that C has at most two source incompatibles, and by Lemma 4.3, C is width partitionable. The argument is similar if there exists a size 3 destination incompatible that overlaps with another maximal destination incompatible.

Subcase 2.2 Suppose there is a 3-element destination incompatible that does not overlap with any maximal incompatible, then assume that there are overlapping source incompatibles of size at most 2; if this is not true, then Theorem 3.8 suffices to complete the proof. There are two cases.

Subcase 2.3.1 Let all sources be included in some overlapping incompatible (see Figure 4.6). If this is the situation at the source side, then, by Lemma 4.1, the destination side cannot have disjoint 3-element destination incompatibles, as assumed.

Subcase 2.3.2 Let one source be not included in an overlapping incompatible. Figure 4.7 shows the only possible case (within isomorphism and duality). If the communication shown in bold in the figure is scheduled in the first step, then the remaining communications form

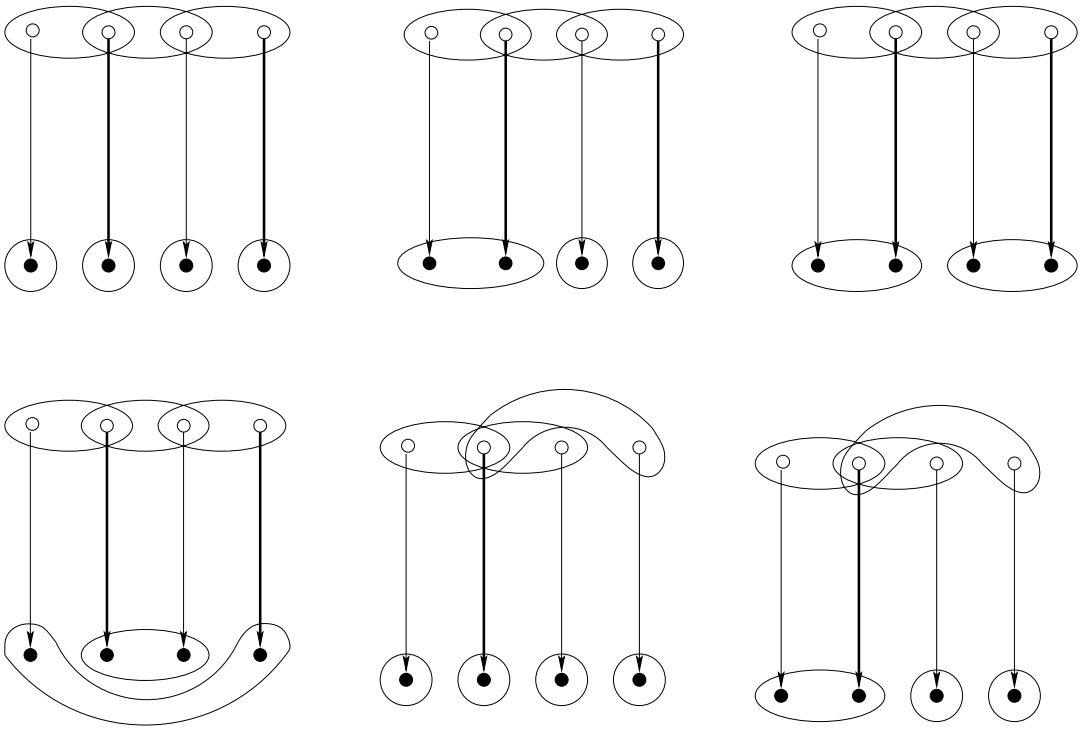


Figure 4.8: Possibilities for Subcase 3.1 of Theorem 4.6

a set of width 2 and the incompatibles are disjoint. Therefore, set C is width partitionable.

The argument is similar if there exists a 3-element source incompatible.

Case 3 If C has width 2, then we have the following cases.

Subcase 3.1 Let all sources be included in some overlapping incompatible (see Figure 4.6). Figure 4.8 shows all possible cases for the incompatibility graph (within isomorphism and duality). If the communications shown bold in the figure (for all cases) are scheduled in the first step, then the remaining communications form a set of width 1. Therefore, set C is width partitionable.

Subcase 3.2 Let one source be not included in an overlapping incompatible. Figure 4.9 shows all possible cases here (within isomorphism and duality). If the communications shown bold in the figure (for all cases) are scheduled

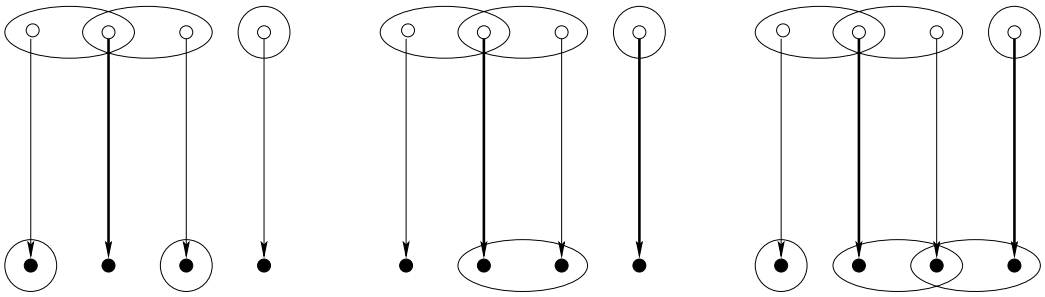


Figure 4.9: Possibilities for Subcase 3.2 of Theorem 4.6; the "uncircled" destinations of the first two graphs could be together or separate.

in the first step, then the remaining communications form a width-1 set. Therefore, set C is width partitionable. ■

4.1.1.3 Number of Incompatibles in a Simplest Set

Here we examine the minimum number of source and destination incompatibles in a simplest set that is not width partitionable. First we develop some intermediate results.

Lemma 4.7 *Let communication set C have only two disjoint source incompatibles I_1 and I_2. Let G_1 and G_2 be the sets of destinations corresponding to sources in incompatibles I_1 and I_2, respectively. Then each destination of C could be in at most two maximal destination incompatibles.*

Proof: Let destination x' be in three distinct maximal destination incompatibles. Since $x \in J_1 \cap J_2$ and $J_1 \neq J_2$, there are destinations $y' \in J_1 - J_2$ and $z' \in J_2 - J_1$. Let $(y, y'), (z, z') \in C$. By the dual of Lemma 4.1, y and z are in different source incompatibles. Without loss of generality, let $y \in I_1$ and $z \in I_2$. Recall that $x' \in J_3$. We now consider two cases.

Case 1 Suppose there is a destination $w' \in J_3$ such that $w' \notin J_1 \cup J_2$. Let $(w, w') \in C$ and without loss of generality let $w \in I_1$. The fact that $w' \in J_3 - J_1$ and $y' \in J_1 - J_3$ while $w, y \in I_1$ is a contradiction of Lemma 4.1.

Case 2 Suppose there is no destination $w' \in J$ such that $w' \notin J_1 \cup J_2$. That is, either $w' \notin J_1$ or $w' \notin J_2$. Without loss of generality, let $w' \notin J_2$. Therefore we have

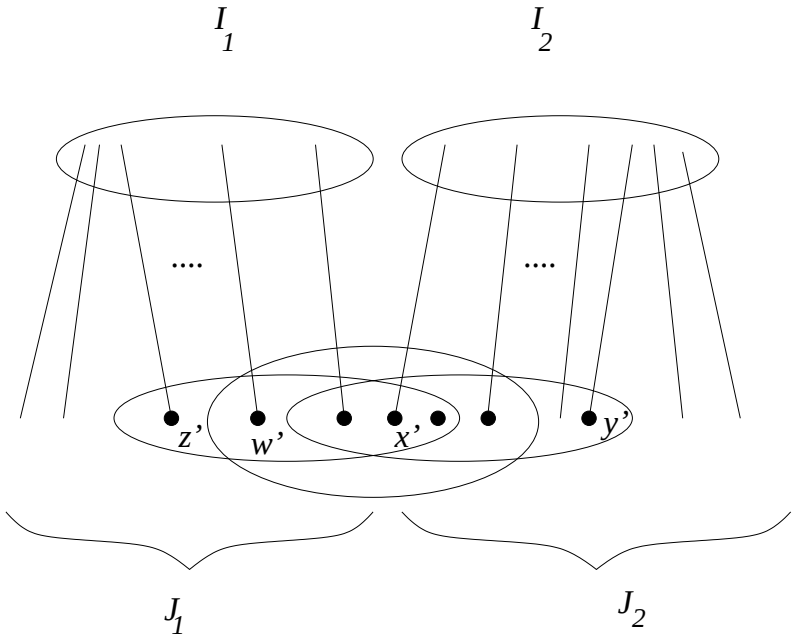


Figure 4.10: Illustration of the proof of Lemma 4.7

the situation in Figure 4.10. By Lemma 4.2, this situation cannot happen; that is, w and x cannot be in I_1 and I_2, respectively. ■

Now we consider a special case of the communication set C of Lemma 4.7, which has two disjoint source incompatibles. This represents one case in the proof of Theorem 4.11. For some communication $(x, x') \in C$, let $C' = C - \{(x, x')\}$. In the next two lemmas we prove that if C' is width partitionable, then C is also width partitionable. We consider two cases. The first (Lemma 4.8) considers the situation where x' is in only one maximal destination incompatible. The second case (Lemma 4.9) considers the situation where x' is in two maximal destination incompatibles. Recall that Lemma 4.7 has established that x' cannot be in more than two maximal incompatibles. We now consider the first situation.

Lemma 4.8 *Let communication set C have two disjoint source incompatibles. For some communication $(x, x') \in C$, let $C' = C - \{(x, x')\}$. Let x' be in only one maximal destination incompatible. If C' is width partitionable, then C is width partitionable.*

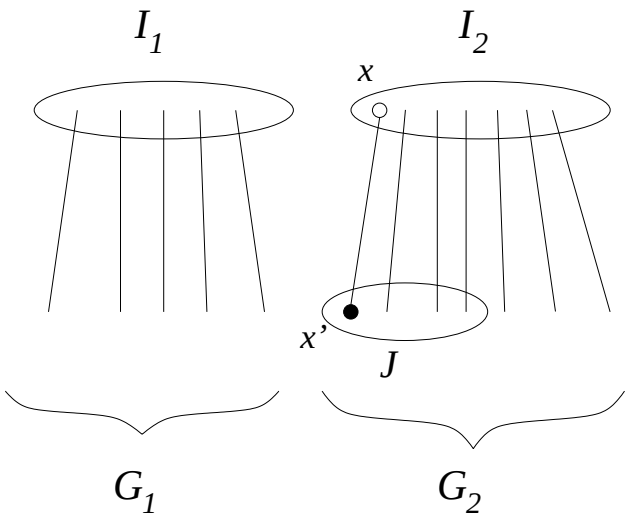


Figure 4.11: Illustration of the proof of Case 1 Lemma 4.8

Proof: Let widths of C and C' be w and w', respectively. Clearly $w = w'$ or $w = w' + 1$. Note that C' is width partitionable. If the width of $w = w' + 1$, then C is width partitionable; simply schedule (x, x') after all communications of C'. If $w = w'$, then we only need to show the existence of a width-1 set $C_1 \subseteq C$ such that $C - C_1$ has width $w - 1$ (set C_1 has only two communications, one from each source incompatible of C). Let I_1 and I_2 be the two source incompatibles of C. Let G_1 and G_2 be the sets of destinations corresponding to sources in incompatibles I_1 and I_2 respectively. We have the following cases.

Case 1 Let x' be in only one maximal incompatible, J, all of whose elements are from G_2 (see Figure 4.11). Here choose C_1 as the communications of any step of the schedule of C' that contains a communication (y, y') with source $y \in I_2 - \{x\}$. Since C_1 is a step of the schedule for C', its width is 1. Since $|J| \leq |G_2| = |I_2|$, incompatible J can be maximum only if I_2 is. Therefore scheduling communication (y, y') with source $y \in I_2$ suffices to guarantee that destination incompatibles are taken care of.

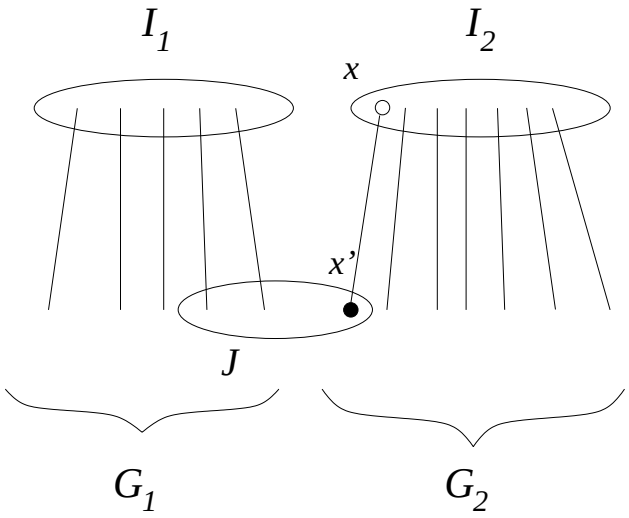


Figure 4.12: Illustration of the proof of Case 2 Lemma 4.8

Case 2 Let x' be in only one maximal incompatible, J, all of whose elements (except x') are from G_1 (see Figure 4.12). Here by Lemma 4.1, no destination in $J-\{x'\}$ is in the same incompatible as any destination in $G_2 - \{x'\}$ or any destination in $G_1 - J$. Choose C_1 as the communications of any step of the schedule of C' that includes a communication (y, y') such that $y \in I_1$ and $y' \in J$. If this step does not contain any communication with source in I_2, then add any one (other than (x, x')).

Case 3 Let x' be in only one incompatible, J, whose elements are from both G_1 and G_2 (see Figure 4.13). Choose C_1 as any step of the schedule of C' that contains a communication $(y, y') \neq (x, x')$ such that $y \in I_2$ and $y' \in J$. As before this will guarantees that $C - C_1$ has width one less that C. ■

We now consider the situation where x' is in two destination incompatibles.

Lemma 4.9 *Let a communication set, C, have two disjoint source incompatibles. For some communication $(x, x') \in C$, let $C' = C - \{(x, x')\}$. Let x' be in two destination incompatibles. If C' is width partitionable, then C is width partitionable.*

Proof outline: Let I_1 and I_2 be the two source incompatibles of C. Let G_1 and G_2 be the sets of destinations corresponding to sources in incompatibles I_1 and I_2,

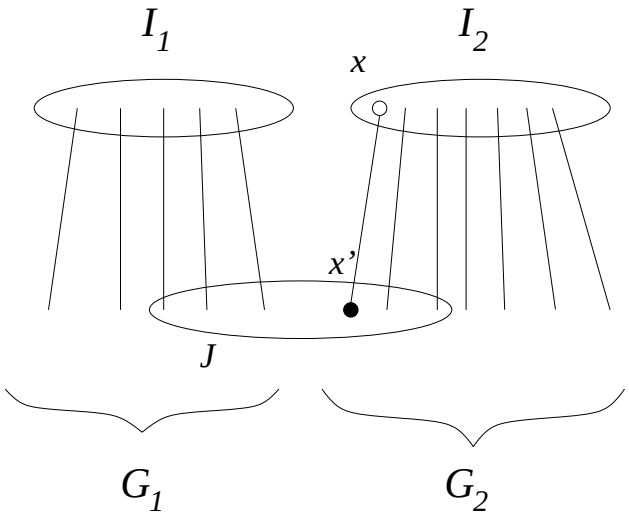


Figure 4.13: Illustration of the proof of Case 3 Lemma 4.8

respectively. Since the main arguments here mirror those of Lemma 4.8, we only outline the proof. Again let the widths of C and C' be w and w', respectively. As before in the proof of Lemma 4.8, if $w = w' + 1$, then C is width partitionable; therefore, let $w = w'$. Once again, we only need to show the existence of a width-1 set $C_1 \subseteq C$ such that $C - C_1$ has width $w - 1$. Let x' be in two overlapping incompatibles, J_1 and J_2; that is, $x \in J_1 \cap J_2$ (see Figure 4.14). Choose C_1 as follows; the argument for why this choice works is as in Lemma 4.8.

Case 1 If there exists a communication $(y, y') \neq (x, x')$ such that $y \in I_2$, $y' \in J_1 \cap J_2$, then choose C_1 as the set of communications of any step of the schedule of C' that contains the communication (y, y').

Case 2 If there exists a communication (y, y') such that $y \in I_1$, $y' \in J_1 \cap J_2$, then choose C_1 as the communications of any step of the schedule of C' that contains the communication (y, y'). If this step does not contain any communication with source in I_2, then add a communication (z, z') such that $z \in I_2 - \{x\}$.

Case 3 If Case 1 and Case 2 do not apply, then choose C_1 as the communications of any step of the schedule of C' that includes a communication (y, y') such that

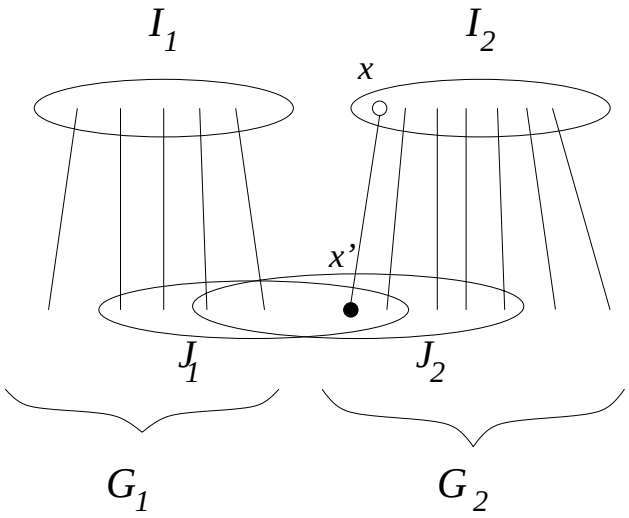


Figure 4.14: Illustration of the proof Lemma 4.9

$y \in I_1$ and $y' \in J_1$. If this step does not contain any communication with source in I_2, then add one (other than (x, x')). ■

Now we show that a communication set must have three source incompatibles and at least three destination incompatibles to be not width partitionable.

Theorem 4.10 *Every communication set with less than three source incompatibles or less than three destination incompatibles is width partitionable.*

Proof: We consider several cases.

Case 1 If the communication set C has only one source incompatible or only one destination incompatible, then it is easy to see that C is width partitionable (schedule one communication at each step).

Case 2 If C has two source incompatibles and two or more destination incompatibles, then we have the following cases.

Subcase 2.1 If the two source incompatibles overlap, then by Lemma 4.3, C is width partitionable.

Subcase 2.2 If the two source incompatibles are disjoint, then we have the following cases.

Subcase 2.3.1 If the destination incompatibles are disjoint, then C is width partitionable (Theorem 3.8).

Subcase 2.3.2 If the destination incompatibles overlap, then we proceed by induction on the number of communications, $n \geq 3$, in the communication set C. If $n = 3$, then by Lemma 4.4, C is width partitionable. Assume the lemma to hold for any set with n communications and consider an $(n+1)$-element communication set, C, with two disjoint source incompatibles. Let C have width w. For some communication $(x, x') \in C$, let $C' = C - \{(x, x')\}$. Clearly, $|C'| = n$ and by the induction hypothesis C' is width partitionable.

The destination x' could be in one destination incompatible, or in two overlapping destination incompatibles. By Lemma 4.7, it cannot be in three or more incompatibles. If x' is in only one destination incompatible, then by Lemma 4.8, C is width partitionable. If x' is two overlapping destination incompatibles, then by Lemma 4.9, C is width partitionable. ■

Theorem 4.11 *Let C be a communication set that is not width partitionable. The following statements hold.*

(*i*) *The width of C is at least two.*

(*ii*) *C has at least five communications.*

(*iii*) *C has at least three source incompatibles and at least three destination incompatibles.*

Proof: Clearly, a communication set with width 1 is width partitionable since the set has disjoint incompatibles. For part (2), Lemma 4.4 and Theorem 4.6 show that the number of communications must be at least five for the set C to be not width partitionable. For part (3), Theorem 4.10 shows that if the number of source (or destination) incompatibles is less than three, then the set C is width partitionable.

■

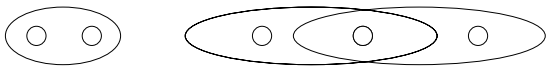


Figure 4.15: Relationship between source incompatibles for a simplest set

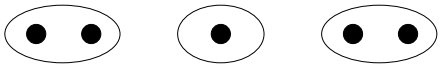


Figure 4.16: Relationships between disjoint incompatibles of a simplest set

4.1.2 Choices of the Simplest Sets

In this section we show that there are only two sets (to within isomorphism and source/destination duality) that satisfy the requirements of a simplest set in Theorem 4.11. Without loss of generality assume that source incompatibles overlap. (For a communication set to be not width partitionable, an overlap between source incompatibles and/or destination incompatibles must exist.) Destination incompatibles may or may not overlap. We now examine the relationships between source incompatibles and between destination incompatibles. Recall that the simplest set that is not width partitionable has at least five communications, a width of two, three source incompatibles, and three destination incompatibles.

Relationship between Source Incompatibles: Within the given constraints, the only possibility for source incompatibles is as shown in Figure 4.15.

Relationships between Destination Incompatibles: Destination incompatibles may or may not overlap. If destination incompatibles do not overlap, then the only possibility between destination incompatibles, while satisfying the simplest set conditions, is as shown in Figure 4.16. If destination incompatibles overlap, then by duality the only possibility is as shown in Figure 4.15.

The Simplest Set that is Not Width Partitionable: From the above discussion, the simplest set that is not width partitionable can have one of two general forms

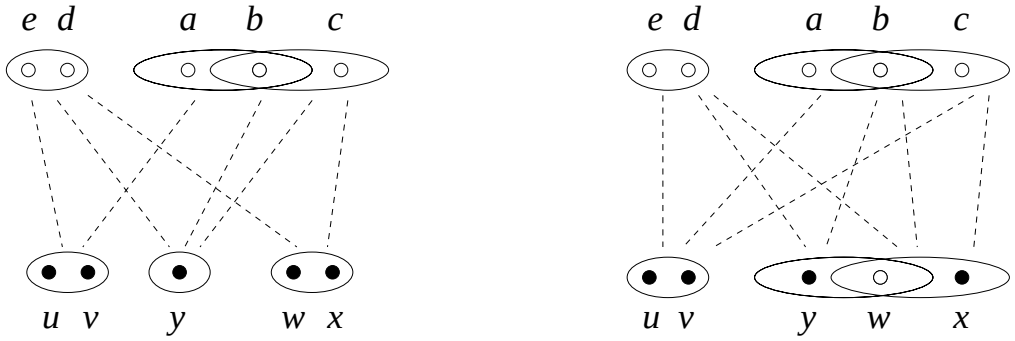


Figure 4.17: The two forms of a smallest set

shown in Figure 4.17. At this stage we have not yet paired sources and destinations. The dashed lines indicate this situation. We now prove that only two pairings are possible (to within isomorphism and duality) so that the communication set is not width partitionable. Let the sources be a, b, c, d, e and the destinations be u, v, w, x, y (see Figure 4.17).

By Lemma 4.1 and its dual, if two source (resp., destination) incompatibles overlap, then their exclusive elements (elements that are not common to both) must have destinations (resp., sources) in disjoint destination (resp., source) incompatibles. Also note that for any schedule of a communication set, every communication must be scheduled at some step. There is no loss of generality in assuming that a communication of our choosing is scheduled in the first step. We use this fact in ascertaining whether or not a set of communications is width partitionable. We have the following cases.

Case 1 Here we consider the case where destination incompatibles are disjoint (see Figure 4.17(a)). We have two subcases.

Subcase 1.1 If (b, y) is a communication, then Figure 4.18(a) shows the only possible mapping between sources and destinations. By Lemma 4.1, sources a and c cannot have destinations in the same incompatible. Without loss of generality, let (a, v) and (c, x) be communications. This implies that (e, u) and (d, w) (or (e, w) and (d, u)) must be communications. As ex-

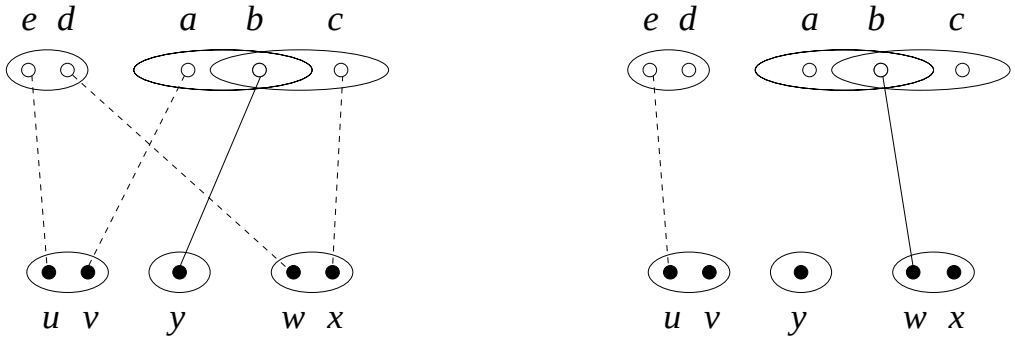


Figure 4.18: Simplest sets with disjoint destination incompatibles

plained for Figure 3.6 (page 36), the communication set of Figure 4.18(a) is not width partitionable.

Subcase 1.2 Suppose (b, y) is not a communication. Without loss of generality, let (b, w) be a communication (see Figure 4.18(b)). By Lemma 4.1, a communication must exist between a source in incompatible $\{e, d\}$ and a destination in incompatible $\{u, v\}$ (say communication (e, u)). The communication set C of the incompatibility graph of Figure 4.18(b) is width partitionable because communications (b, w) and (e, u) can be scheduled at the same step. Since $C - \{(b, w), (e, u)\}$ has width 1, it follows that C can be scheduled in two steps.

Case 2 Here we consider the case where destination incompatibles overlap (see Figure 4.17(b)). We have three subcases. The first case consider the situation where the overlapped source, b, is mapped to the overlapped destination w. The second case is such that the overlapped source, b, is mapped to a destination in incompatible $\{u, v\}$, and a source in the incompatible $\{e, d\}$ is mapped to the overlapped destination w. The third case examines the situation where b and w are mapped to elements that belong to an overlapping incompatible. The case where b is mapped to either y or x and w is mapped to an element in incompatible $\{e, d\}$ is not possible by Lemma 4.1.

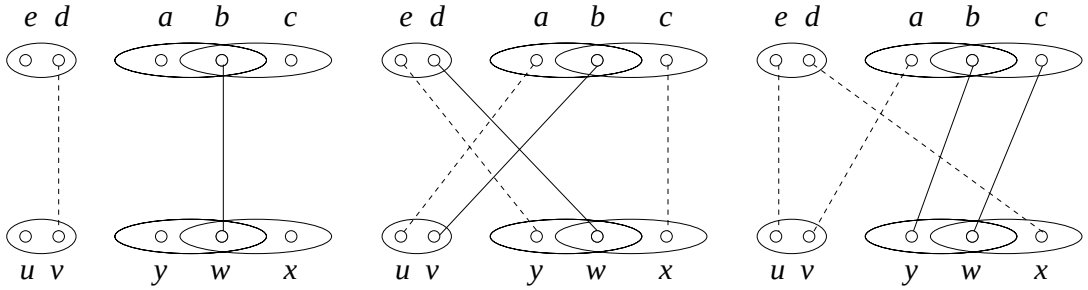


Figure 4.19: Simplest sets with overlapping destination incompatibles

Subcase 2.1 If (b, w) is a communication, then by Lemma 4.1 there must be a communication (say (d, v)) between source incompatible $\{e, d\}$ and destination incompatible $\{u, v\}$ (see Figure 4.19(a)). As in Subcase 1.2, $C - \{(b, w), (d, v)\}$ has width 1, it follows that C is width partitionable.

Subcase 2.2 Suppose that source b correspond to a destination in incompatible $\{u, v\}$ (say communication (b, v)), and that a communication exists (say communication (d, w)) between a source in $\{e, d\}$ and destination w (see Figure 4.19(b).) Again communication set C of the incompatibility graph of Figure 4.19(b) is width partitionable as $C - \{(b, v), (d, w)\}$ has width 1, it follows that C can be scheduled in two steps.

Subcase 2.3 If (b, y) (or (b, x)) is a communication and (c, w) (or (a, w)) is a communication, then the communication set C of the incompatibility graph of Figure 4.19(c) is not width partitionable because the only communication that can be scheduled simultaneously with (b, y) is either (e, u) or (d, x). Since $C - \{(b, y), (e, u)\}$ or $C - \{(b, y), (d, x)\}$ has width 2, it follows that C cannot be scheduled in two steps.

In summary, Subcase 1.1 and Subcase 2.3 are the only possibilities for the smimplest sets that are not width partitionable.

Theorem 4.12 *The simplest set of communication that is not width partitionable has an incompatibility graph whose form is (to within isomorphism and duality) one of the two shown in Figure 4.18(a) and Figure 4.19(c).* ■

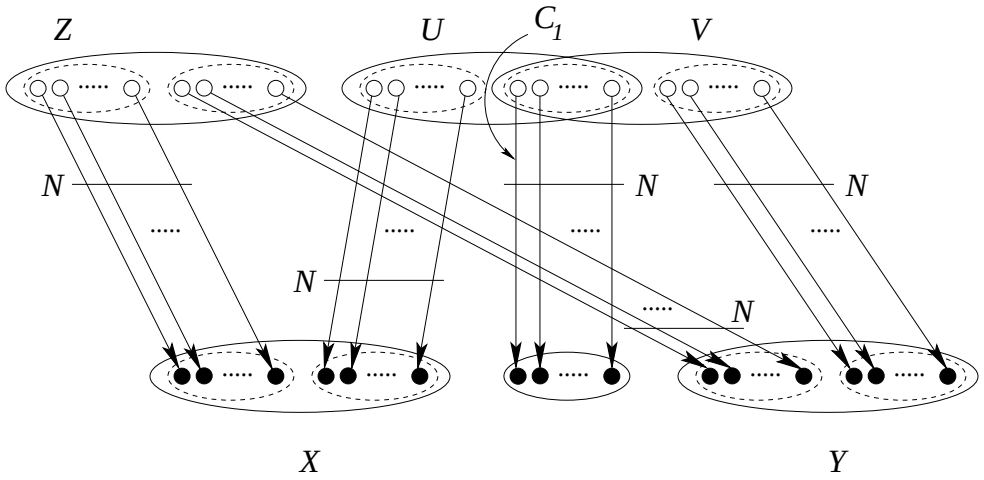


Figure 4.20: An N-extension of an incompatibility graph

Call the communication sets corresponding to these two graphs the *basic simplest sets*.

4.2 A Bound on the Number of Extra Steps

The width-2 communication set of Figure 4.18(a) requires three steps for scheduling on the CST. In other words, it requires one extra "step" beyond its width. In this section we prove that for any $w \geq 2$, there exists a communication set of width w that requires $\lceil \frac{w}{4} \rceil$ extra steps.

Consider the incompatibility graph of Figure 4.20. It is identical to the graph of Figure 4.18(a) except that each communication is replaced with group of N communications. If communication set C and $C(N)$ denote the sets corresponding to the graphs in Figures 4.18(a) and 4.20 respectively, then $C(N)$ is called an *N-extension* of C.

Theorem 4.13 *An N-extension of a basic simplest graph has width $w = 2N$ and requires a schedule of $w + \lceil \frac{w}{4} \rceil$ steps on the CST.*

Proof: We prove the theorem for an N-extension of the graph of Figure 4.18(a). The proof for the other basic simplest graph (Figure 4.19(c)) is similar. Let C_1 be

the communications corresponding to the overlapped sources of $C(N)$. Any schedule of $C(N)$ must have each communication of C_1 in different steps. Without loss of generality, assume that the first N steps of this schedule include communications of C_1. During these steps, let the schedule include α communications with destinations in X and β communications with destinations in Y (see Figure 4.20). Clearly, their sources must be from Z. Therefore, any given step can include a communication with a destination in X or with a destination in Y (but not both).

Thus $0 \leq \alpha + \beta \leq N$. Without loss of generality, let $\alpha \geq \beta$. Then at the end of the first N steps, we have the following situation. Incompatibles U and V have N sources each. Incompatibles X and Y have $2N - \alpha$ and $2N - \beta$ destinations respectively. Since $\alpha \geq \beta$, $2N - \beta \geq 2N - \alpha$ and since $2N - \beta \geq N$, Y is a maximum incompatible. Thus communication set $C(N)$ needs at least $N + 2N - \beta = 3N - \beta$ steps. The maximum value of β minimizes the number of steps in the schedule. This maximum value is $\beta = \lfloor \frac{N}{2} \rfloor$ and therefore the minimum number of steps is $w + \lceil \frac{w}{4} \rceil$.

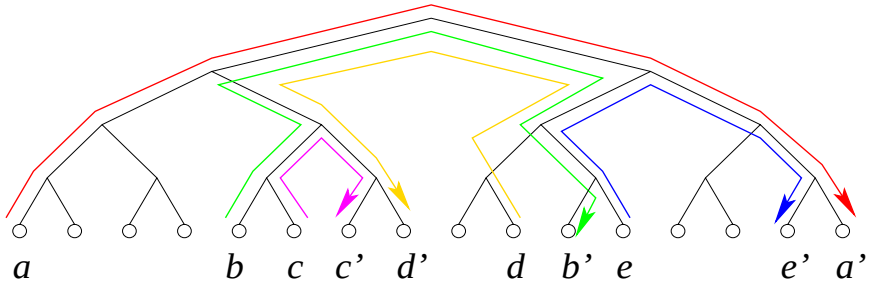
■

4.3 Non-Oriented, Well-Nested Sets

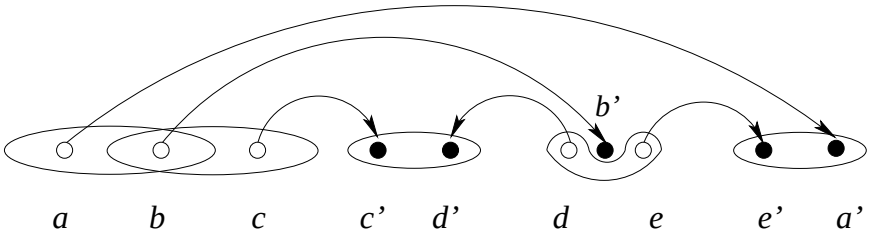
In Section 3.4 we proved that all oriented, well-nested communication sets are width partitionable. In this section we consider non-oriented, well-nested sets. In an oriented, well-nested set if some communications changed orientation, we call such set a *non-oriented, well-nested set*. In general, such sets are not width partitionable. Figure 4.21(a) and (b) show an example of a width-2, non-oriented, well-nested set that requires three steps on the CST. To see that tree steps are required, note that the incompatibility graph of Figure 4.21(b) is the same as that in Figure 4.1.

Even though non-oriented well-nested sets are not width partitionable, we identify a class of non-oriented well-nested sets that are.

Define a *level-1* oriented well-nested set to have one of the two forms shown in Figure 4.22. That is, every level-1 oriented well-nested set has a condition expression that uses only *cap* (Figure 4.22(a)) or only *concat* (Figure 4.22(b).) Call a level-1 set that uses only *cap* (resp., only *concat*) as a level-1 cap set (resp., level-1 concat set).



(a) Communications on the CST



(b) Incompatibility graph

Figure 4.21: Width-2, non-oriented monotonic set requiring three steps. In part (b), the incompatibility graph has been drawn differently to show the communications clearly.

A *level-2* oriented well-nested set has the forms shown in Figure 4.23. That is, a level-2 set is either a concatenation of several level-1 cap sets or repeated capping of a level-1 concat set. All oriented sets other than level-1 or level-2 sets described above are said to have level ≥ 3.

Let C be a non-oriented set. Construct the oriented counterpart $\vec{C}$ of C by replacing each communication $(x, x') \in C$ such that $x > x'$ by communication (x', x). That is, $\vec{C}$ contains the same communicating pairs as C, except that all sources are to the left of their destinations. For example, Figure 4.24(b) shows the oriented counterpart of the communication set of Figure 4.24(a).

The level (1, 2, or ≥ 3) of a non-oriented well-nested set is the same as its oriented counterpart. The level of a non-oriented, well-nested set appears to be an important factor in determining its width partitionability.

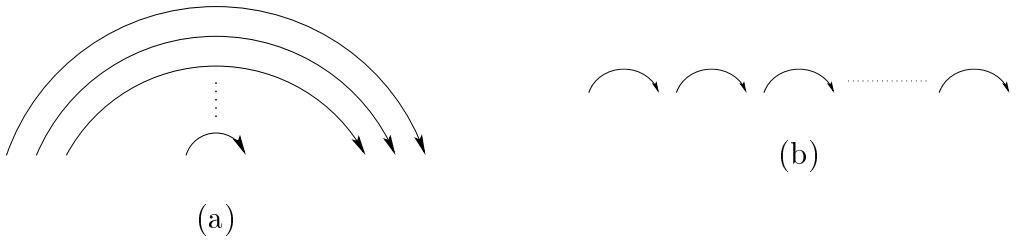


Figure 4.22: Level-1 oriented well nested sets

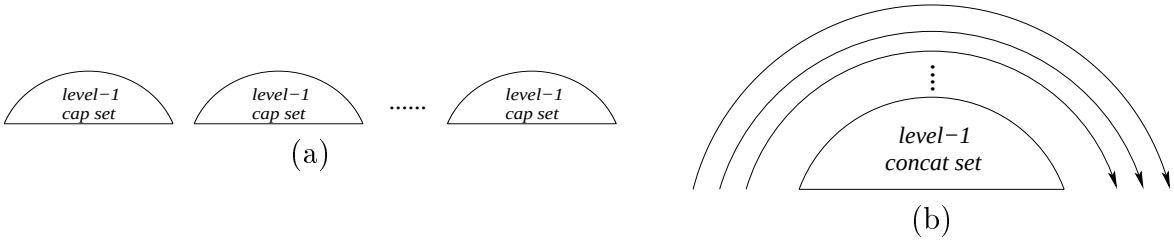
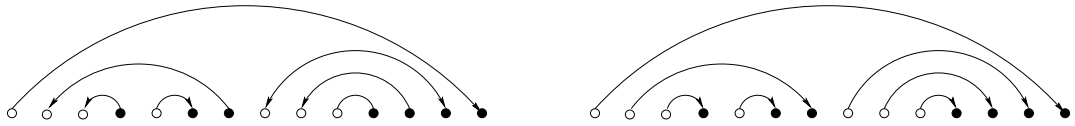


Figure 4.23: Level-2 oriented well nested sets

Lemma 4.14 *Let C be a level-2 oriented well-nested communication set formed by repeated capping of a level-1 concat set. Then every optimal schedule of C can schedule all communications of the level-1 concat set in the same step.*

Proof: Since C is oriented, its optimal schedule has w steps (where w is the width of C). The only possible form for C is shown in Figure 4.25. Let $C' = \{c_1, c_2, \cdots, c_k\}$ be the level-1 concat set and let $\gamma_1, \gamma_2, \cdots, \gamma_q$ be the capping communications in the order of their proximity to elements of C' (see Figure 4.25). Let $w_1, w_2, \cdots, w_k$ be the widths of the incompatibles that include $c_1, c_2, \cdots, c_k$, respectively. By Lemma 3.12 (page 47), the source incompatibles are all different (that is, $\{c_i, c_j\}$ is not an incompatible for any $1 \leq i < j \leq k$). Observe that by Lemma 3.14 (page 49), for any $1 \leq i \leq k$, if c_i is incompatible with γ_h $(1 \leq h \leq q)$, then c_i is also incompatible with every γ_g (where $1 \leq g \leq h$). Let $w_m = \max(w_1, w_2, \cdots, w_x)$ and consider the communication set $C'' = C - C' + c_m$ (see Figure 4.26 where the communications of C'' are in bold). Set C'' is an oriented well-nested set and therefore is width partitionable. Consider the step s (say) in the optimal schedule of C'' that includes communication c_m. In step s, none of the $w_m - 1$ communications that are incompatible with c_m are



(a) (b)

Figure 4.24: Unoriented set and its oriented counterpart

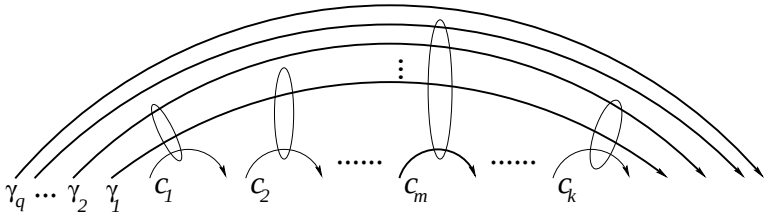


Figure 4.25: Illustration of the proof of Lemma 4.14

scheduled. This also implies that none of the communications that are incompatible with any of the c_i's ($1 \leq i \leq k$) is also scheduled in step s (from our earlier observation based on Lemma 3.12). Thus, all communications in C' can be scheduled in the step s. ■

Now we return to non-oriented sets.

Theorem 4.15 *Every level-*1 *or level-*2 *non-oriented, well-nested communication set is width partitionable.*

Proof: First we consider level-1 non-oriented, well-nested sets (see Figure 4.27(a) and (b)). Figure 4.27(a) uses only the cap condition and Figure 4.27(b) uses only the concat condition.

Consider any level-1 capped set C. This set can be partitioned into two oriented sets C', C'' (one in each direction). Let w' and w'' be their widths. Clearly, the width of C must be at least $\max(w', w'')$. Any $c' \in C'$ and $c'' \in C''$ are not incompatible. Let $c' = (x, x')$ and $c'' = (y, y')$. Without loss of generality, let $x < y' < y < x'$. Thus, $\ell(x, y)$, $\ell(x', y') \geq \ell(y, y')$, and hence $\{x, y\}$ and $\{x', y'\}$ are not incompatible. Thus,

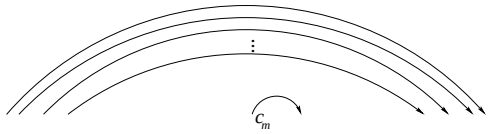


Figure 4.26: The communication set C''

(a) (b)

Figure 4.27: Level-1, non-oriented well-nested sets

the communications of C' and C'' are not restricted in any way by each other (only by themselves). Since C' and C'' are width partitionable (Theorem 3.15), so is C.

A level-1 concat set can only be of width 1 or 2. If the width is 1, then no communication is incompatible with another; schedule in one step. If the width is 2, partition the set into two oriented width-1 sets and schedule the two directions in two steps.

We now consider level-2 sets (see Figure 4.28). We consider two cases corresponding to Figure 4.28(a) and (b).

Case 1 Communication set C has the form shown in Figure 4.28(a). Let C have width w. We proceed by induction on the width of C. Clearly, a width-1 communication set is width partitionable. Assume the assertion to hold for a non-oriented set with width at most $w-1$ and consider a set C with width w. We only need show the existence of a set $C_1 \subseteq C$ of width 1 such that $C - C_1$ is of width $w-1$.

Let C be the concatenation of level-1 cap sets $P_1, P_2, \cdots, P_\alpha$, for some integer $\alpha \geq 1$ (see Figure 4.28(a)). For $1 \leq i \leq \alpha$, let $P_i = L_i \cup R_i$, where L_i (resp., R_i) is the set of communications of P_i oriented to the left (resp., to the right). Define $C_L = \bigcup_{i=1}^{\alpha} L_i$ and $C_R = \bigcup_{i=1}^{\alpha} R_i$. Individually, $L_1, L_2 \cdots L_x$, $R_1, R_2 \cdots R_\alpha$ are

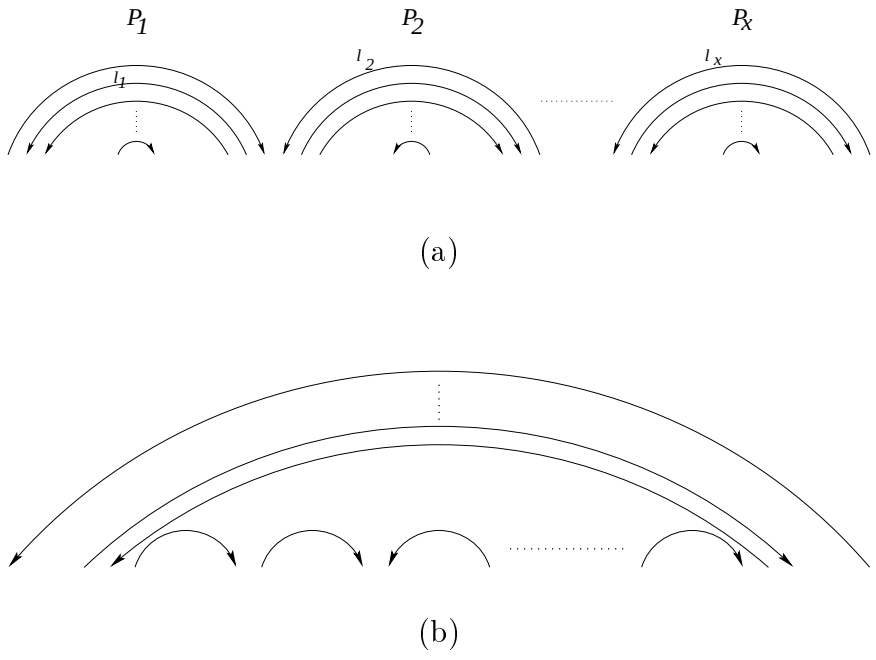


Figure 4.28: Level-2, non-oriented well nested sets

width partitionable (Theorem 3.15). First we select elements of C_L that will be in C_1 and then will deal with C_R. Let l_i be the outermost communication of L_i. Let $S(i)$ be the set of communications that are scheduled at the same step as the communication l_i. Let $S = \bigcup_{i=1}^{\alpha} S(i)$. By Lemma 3.12, S has width 1. Include S in C_1. Clearly the width of C_L has been reduced by 1. Some of the communications of C_R (oriented to the right) may be incompatible with some communications of S, while others may not.

Let r_i be the outermost communication of R_i that is not incompatible with any communication of S. Let $T(i)$ be the set of communications of R_i that are scheduled in the same step as r_i (in the optimal schedule of R_i). If $T(i)$ contains any communication that are incompatible with communication in S; then simply exclude them from $T(i)$. Let $T = \bigcup_{i=1}^{\alpha} T(i)$.

Clearly, T has width 1 and $C_R - T$ has width one less than C_R. Let $C_1 = S \cup T$. Clearly, $S \cup T$ has width 1. To see that $C - C_1$ has width $w - 1$ observe that the only communications excluded from $T(i)$'s above are those incompatible

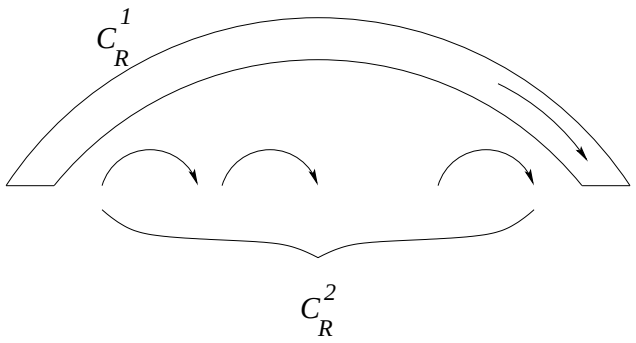


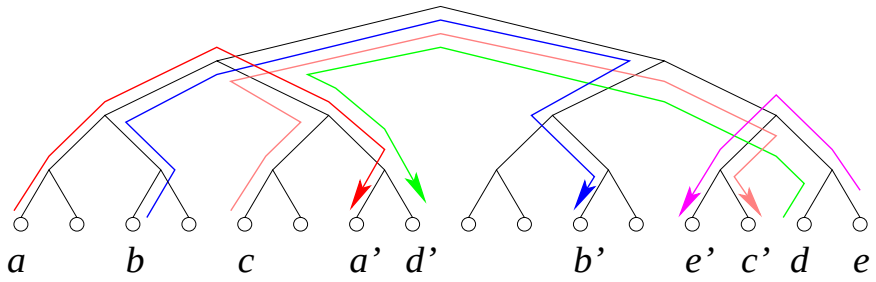
Figure 4.29: The set C_R

with communications of S. These incompatibles are clearly represented by communications of S.

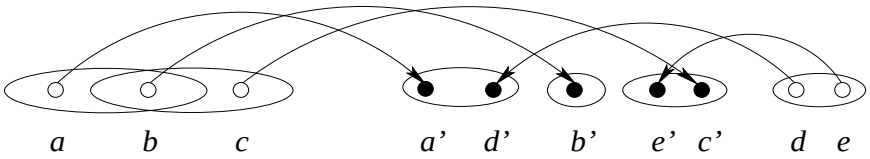
Case 2 Communication set C has the form shown in Figure 4.28(b). Partition C into two oriented communications $C_L = C_L^1 \cup C_L^2$, and $C_R = C_R^1 \cup C_R^2$ (see Figure 4.29) oriented towards the left and right, respectively. Each C_L and C_R is width partitionable, and all communications in the level-1 concat set C_L^2 or C_R^2 can be scheduled in one step (Lemma 4.14).

Note that for any $c' \in C_R^1$ and $c'' \in C_L^1$, c' and c'' are not incompatible (as described for level-1 sets). For any $c' \in C_R^2$ and $c'' \in C_L^2$, if c' and c'' are not incompatible, then the same can be said for the entire sets C_L and C_R. Thus C_L and C_R can be scheduled together, and hence C is width partitionable.

If there is some $c' \in C_R^2$ and $c'' \in C_L^2$ such that c' and c'' are incompatible, then the width of level-1 concat set, $C_R^2 \cup C_L^2$, is 2. Schedule C_L such that all the communications of C_L^2 are in one step (Lemma 4.14) and call this step s_L. Similarly, schedule C_R such that all the communications of C_R^2 are in one step s_R. Now, schedule C_L and C_R together such that $s_L \neq s_R$; permuting the steps of an optimal schedule gives an optimal schedule. ■



(a) Communications on CST



(b) Incompatibility graph

Figure 4.30: Width-2, non-oriented, monotonic set requiring three steps; In part (b) the incompatibility graph has been drawn differently to show the communications clearly.

4.4 Non-Oriented, Monotonic Sets

The definition of the oriented, monotonic set in (Section 3.5) requires the communications to be directed from left to right or vice versa (oriented). In an oriented, monotonic communication set, if some of the communications changed orientation, we call such a set *non-oriented, monotonic set*. In other words, a non-oriented monotonic set is a set that has communications in different directions, but its oriented counterpart is monotonic.

In general, a non-oriented monotonic communication set is not width partitionable (see Figure 4.30(a) and (b)). The communication set of Figure 4.30(b) has the same incompatibility graph as Figure 4.1 and hence it is not width partitionable. As in the non-oriented, well-nested case, with some restrictions a non-oriented monotonic set is width partitionable. Consider the monotonic set of Figure 4.31(a) where each source

(a) (b)

Figure 4.31: Separable monotonic sets

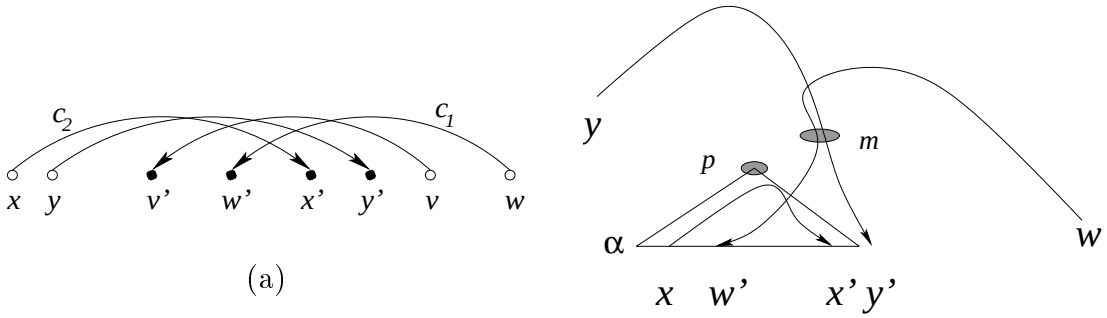


Figure 4.32: Illustration of the proof of Lemma 4.16

is to the left of all destinations. We call such set as a *separable set*. A non-oriented monotonic set is separable if its oriented counterpart is separable (see Figure 4.31(b)).

Lemma 4.16 *Let C be a separable, non-oriented, monotonic set. Let C_L (resp., C_R) be the set of communications in C that are oriented to the left (resp., right). Let c_1 (resp., c_2) be the rightmost (resp., leftmost) communication of C_L (resp., C_R). The communication c_1 is incompatible with any communication in C_R if and only if c_1 is incompatible with c_2.*

Proof: If c_1 and c_2 are incompatible, then obviously C_1 is incompatible with some communication of C_R. We now proceed in the only if direction. Let $(x, x'), (y, y') \in C_R$ and $(w, w'), (v, v') \in C_L$. Let $c_1 = (w, w')$ and $c_2 = (x, x')$ (see Figure 4.32(a)). Let $\ell ca(x, x') = p$ and $\ell ca(y', w') = m$. Suppose that $\{w', y'\}$ is incompatible but $\{w', x'\}$ is not (see Figure 4.32(b)). This requires that $\ell evel(p) < \ell evel(m)$. Let α be the leftmost node of the subtree rooted at p. It follows that $x \geq \alpha$ and $y < \alpha$ which contradicts the fact that $x < y$. ■

Remarks: Similarly, the communication c_2 is incompatible with any communication in C_L if and only if c_2 is incompatible with c_1.

Intuitively, Lemma 4.16 shows that the destinations of the oriented sets C_L and C_R are not incompatible unless the destination, w', of the rightmost communication of C_L, and the destination, x', of the leftmost communication of C_R are incompatible.

By the same argument, a similar assertion could be made about the possible interaction between the sources of C_L and C_R. Thus, we have the following result.

Lemma 4.17 *Let C be a separable, non-oriented, monotonic set. Let C_L (resp., C_R) be the set of communications in C that is oriented to the left (resp., right). Let c_3 (resp., c_4) be the leftmost (resp., rightmost) communication of C_L (resp., C_R). The communication c_3 is incompatible with any communication in C_R if and only if c_3 is incompatible with c_4.* ■

Remarks: Similarly, the communication c_4 is incompatible with any communication in C_L if and only if c_4 is incompatible with c_3.

Theorem 4.18 *Let C be a separable, non-oriented, monotonic set. Let C_L (resp., C_R) be the set of communications in C that is oriented to the left (resp., right). Let c_1 (resp., c_3) be the rightmost (resp., leftmost) communication of C_L. Let c_2 (resp., c_4) be the leftmost (resp., rightmost) communication of C_R. If at most one of $\{c_1, c_2\}$ and $\{c_3, c_4\}$ is an incompatible, then C is width partitionable.*

Proof: If neither $\{c_1, c_2\}$ nor $\{c_3, c_4\}$ is incompatible, then C_L and C_R can be scheduled independently. Without loss of generality, let c_1, c_2 be incompatible (see Figure 4.33). We proceed as in the proof of Theorem 4.15. Let step s in the schedule for C_L include communication c_1. Let S be the set of all communications in step s. Let c' be the leftmost communication of C_R such that c' and c_1 are not incompatible. Clearly, $c_2 \neq c'$. Let t be the step of the schedule of C_R that includes c'. Let T be the set of all communications of C_R that are scheduled in step t. Exclude from T any communication that is incompatible with communications of S. If $C_1 = S \cup T$, then C_1 has width 1 and $C - C_1$ has width $w - 1$, where w is the width of C. The reasoning for this assertion is the same as that in the proof of Theorem 4.15. ■

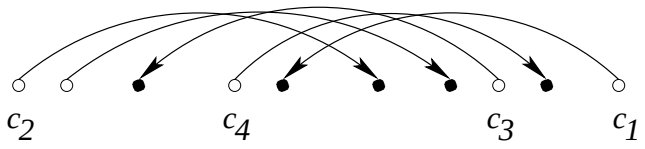


Figure 4.33: A separable monotonic communication set. Letters next to sources represent the communication.

4.5 Concluding Remarks

In this chapter, we showed that any communication set that is not width partitionable has a width of at least 2, it has at least five communications, at least three source incompatibles, and at least 3 destination incompatibles. We presented a "simplest set" that have exactly these minimum requirements. Further we showed these simplest sets are the only onespossible (to within isomorphism). We showed that there exists a width-w set requiring $w + \lceil \frac{w}{4} \rceil$ steps to be scheduled on the CST. We also showed that while non-oriented, well-nested and monotonic sets are not width partitionable, in general, they are under some restrictions.

Chapter 5

Configuring the CST

The communication capability of the CST (see also Section 2.1) has been analyzed in Chapters 3 and 4 and methods have been devised to schedule many interesting communication classes. Such a schedule partitions the communications into several "width-1" communication sets; all communications from a width-1 set can be simultaneously accommodated on the CST. In this chapter we consider only width-1 communication sets. The ability of the CST to accommodate communications of a width-1 set does not mean that it can actually establish in one step the dedicated paths between communicating pairs. Here we address the issue of configuring the CST to perform any width-1 communication set. In other words, we discuss how the CST generates information to configure switches (at its internal nodes) to establish the required paths. Henceforth, "configuring the CST" refers to configuring its switches (see Section 2.1). Once the CST is configured, it is straightforward to perform the communications of a width-1 set. In this chapter we only discuss the configuration, with the understanding that the communications follow in a straightforward manner.

Before we proceed, we note some assumptions used in this chapter and the next. As described in Section 2.1, each pair of communicating PEs of the CST uses a shortest path through the tree. Thus, each path can traverse $O(\log N)$ switches (where N is the number of leaves in the CST). We assume these $O(\log N)$ switch delays to be a basic time unit and allow a "step" to have $O(\log N)$ switch delays. This assumption has been justified by Sidhu *et al.* [43] and independently by us in Chapter 7.

In this chapter we first present a general approach for configuring the CST in one step. The basic idea of configuring the switches is to somehow reflect the global information of connections among PEs based on limited local knowledge. We show that this can be accomplished in one step if the communications possess certain properties. Next we show a class of communications called an "edge-exclusive set" that possesses these properties. This implies that for an edge-exclusive set, the CST can be configured in one step. Then, we present a method to decompose any width-1 communication set into at most three edge-exclusive sets. This, in effect, proves that any width-1 set of communications can be performed on the CST in at most three steps.

Considering that Chapters 3 and 4 provide means to convert a set of communication requirements into a sequence of width-1 sets, the results of Chapters 3, 4 and 5 provide a comprehensive solution to communicating on the CST. In Chapter 6 we apply our techniques to communications on a segmentable bus.

In the next section we outline a general technique for configuring the CST. In Section 5.2 we detail our technique and show that it can be applied to the edge-exclusive sets. Section 5.3 proves that every width-1 set can be decomposed into three edge-exclusive sets. In Section 5.4 we summarize our results.

5.1 CST Configuration—A Broad Outline

In this section, we describe the CST switches, the information flow through them, and outline an approach to a 1-step configuration of the CST to establish the paths of any given width-1 communication set. This configuration approach is based on the idea of Sidhu *et al.* [43]; we extend it to include multiple communications.

The key aspect of this approach is that it uses only information locally available to PEs and knowledge of the pattern of communications to be performed to appropriately configure the CST. For some communication patterns, such as the oriented well-nested sets, the local information is the source/destination status of each PE. In other communication patterns such as those of a segmentable bus, the local information is knowledge of whether a PE is a writer and if it cuts the bus. In general, the

information needed to configure a CST switch could come from the local information for any set of PEs. Our approach restricts this switch information to come from only the leaves of the subtree rooted at the switch. (Though this appears quite restricted, we show (Section 5.3) that our approach works for virtually all width-1 sets.) Thus, our approach only requires that local information from the PEs (leaves of the CST) be fanned-in through their ancestors. The tree structure of the CST provides an ideal platform to accomplish this.

We now describe the approach in detail. Recall that each CST switch has a full-duplex data link to its parent (if any) and two children (see Section 2.1). In addition to the data links, the switch has a control line from each node to its parent. These control lines are used to carry control symbols (holding local information) from a switch (or leaf) to its parent. The CST switch has two main blocks (a) the communication unit (labeled C in Figure 5.1) and (b) the control unit. The communication unit establishes data paths between the three data inputs and the three data outputs

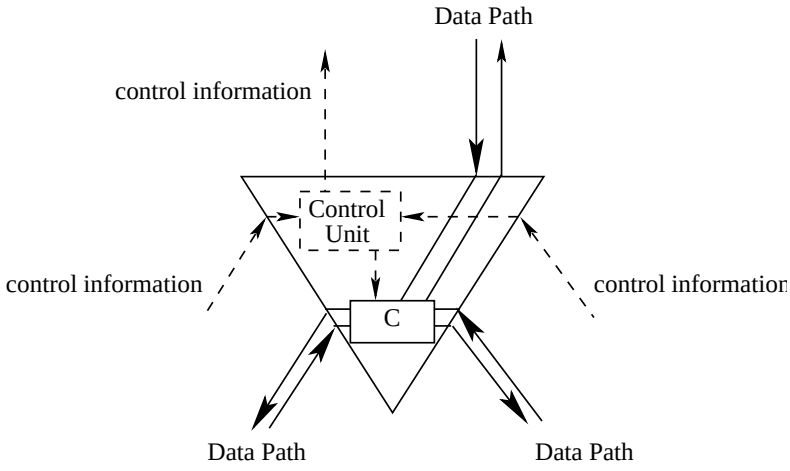


Figure 5.1: Internal Structure of the Switch

of the switch. (Figure 2.2, page 20, shows a sample of data path configurations of a switch.) The control unit accumulates information from the descendants of the switch (through two control input lines) and generates (i) accumulated information to pass

on to its parent (if any) and (ii) information to select the data configuration of the communication unit of the switch.

Putting these ideas together, we now describe the actions performed during a "CST configuration cycle." The leaves use local information to generate control symbols and send these symbols to their parents using control lines. Based on the symbols received, the control logic (combinational logic) in each switch decides on the appropriate data configuration for the switch and passes a symbol (control information) to its parent. This way, the control symbols flow up through the tree, setting switches level by level until they reach the root of the CST. This process involves information flow through $O(\log N)$ switches and, as explained earlier, runs in one step. If the PE can obtain the required local information at run-time, then note that this procedure configures each switch also at run-time.

Based on the discussion so far, our approach imposes the following constraints.

1. Each switch configures its communication unit based on the control symbols it receives from its children.

2. Each switch can generate a control symbol (that captures all relevant information from its descendants) to send to its parent.

Further, to ensure that each control unit is of constant size, we impose the following additional restriction.

3. The CST configuration algorithm must use a constant number of control symbols.

It should be noted that different control logic may be required for different communication classes. For example, a width-1 segmentable bus may need a different control logic compared to an edge-exclusive set (described in Section 5.2).

5.2 Edge-Exclusive Communication Sets

In this section we show that it is possible to configure the CST in one step for an important class of communications called edge-exclusive communications.

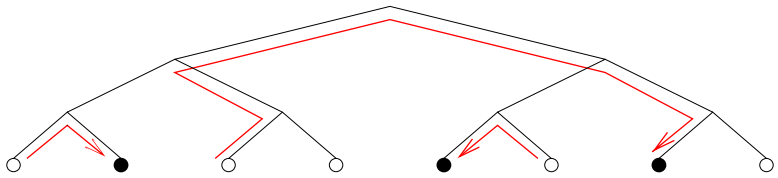


Figure 5.2: Edge-exclusive communication set

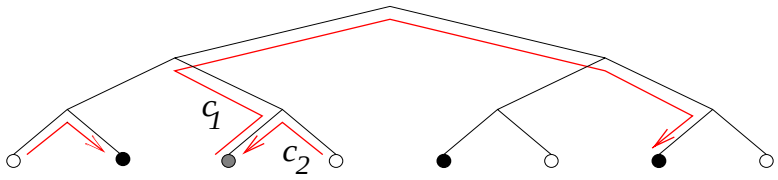


Figure 5.3: A communication set that is not edge-exclusive

Definition 5.1 A set C of communications is *edge exclusive* if and only if no two communications of C use the same CST edge (even in opposite directions). ■

For example, Figure 5.2 shows an edge-exclusive set, whereas the width-1 communication set of Figure 5.3 is not edge exclusive because communications c_1 and c_2 share a common edge. Recall that the width of a set C of communications is the maximum number of communications requiring the use of any one directed edge. Clearly, the width of an edge-exclusive set is 1.

Intuitively, our approach works for an edge-exclusive set because of the following reason. Control information flows up the CST until information from a source meets information from a destination at their lowest common ancestor. The fact that the communication set is edge exclusive guarantees that this source-destination pair is a matching pair (see Lemma 5.1). We now detail a method to configure the CST in one step for any edge-exclusive set.

Assume each PE to only know whether it is a source, destination or neither. If a PE (leaf of CST) is a source (resp., destination), then it passes control symbol `s` (resp., `d`) to its parent (a CST switch). If the PE is neither a source nor a destination, it passes symbol `n` to its parent. For a similar approach involving one source and

f_s	s	d	n
s	–	n	s
d	n	–	d
n	s	d	n

Figure 5.4: The function f_s for edge-exclusive sets

one destination (a single element communication set), Sidhu *et al.* [43] used 2-bit quantities with s = 01, d = 10, and n = 00.

Each switch (internal node) receives control symbols (from the set $S = \{\mathsf{s}, \mathsf{d}, \mathsf{n}\}$) from its children. It uses these symbols to produce a symbol (again from the set S) for its parent. Let $\mathcal{C}$ be the set of configurations of the communication unit of a switch. Then each CST switch can be viewed as two functions, $f_s : S \times S \rightarrow S$ providing a symbol (see Figure 5.4) and $f_c : S \times S \rightarrow \mathcal{C}$ providing a configuration (see Figure 5.5).

Let u be any node of the CST. Let $\mathcal{T}_u$ denote the subtree rooted at u.

Lemma 5.1 *For the algorithm explained above,*

1. *If u is an internal node, then it cannot receive symbols* s, s *from its children or* d, d *from its children.*

2. *If u receives symbols* s, d *from its children, then these symbols correspond to a matching source-destination pair.*

3. *Node u sends symbol* s *to its parent if and only if there is a source in $\mathcal{T}_u$ and the corresponding destination is outside $\mathcal{T}_u$.*

4. *Node u sends symbol* d *to its parent if and only if there is a destination in $\mathcal{T}_u$ and the corresponding source is outside $\mathcal{T}_u$.*

5. *Node u sends symbol* n *to its parent if and only if each source (if any) in $\mathcal{T}_u$ has its corresponding destination in $\mathcal{T}_u$.*

Proof: Let the CST have 2^n leaves. Its internal nodes are arranged in n levels numbered $1, 2, \cdots, n$ (with the root at level n). We proceed by induction on the level

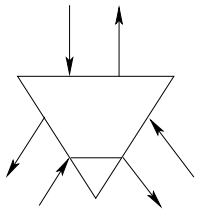
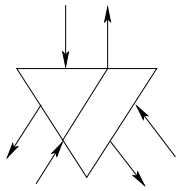
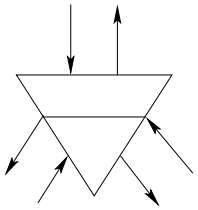
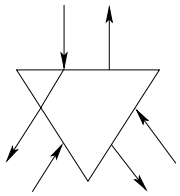
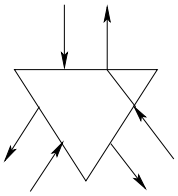
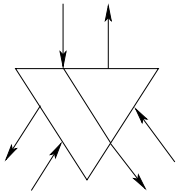
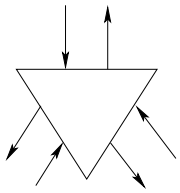
f_c	s	d	n
s	not possible		
d		not possible	
n			

Figure 5.5: The function f_c for edge-exclusive sets

l $(1 \leq l \leq n)$ of an internal node u of the CST. If $l = 1$, then u is a parent of two leaves (PEs). Suppose that u receives symbols s, s from its children (see Figure 5.6). Clearly both sources must use the link from u to its parent to communicate with their corresponding destinations. This is not possible in an edge-exclusive set. Similarly, u cannot receive symbols d, d from its children. For part 2, if u receives symbols s, d from its children, then they form a matching pair. This is because, if they do not form a matching pair, then they have to use the link from u to its parent (even though in opposite directions) which does not meet the requirement of an edge-exclusive set. For part 3, if u sends symbol s to its parent, then it must have received symbols s, n from its children (leaves); see Figure 5.5. So part 3 holds for the base case. Part 4 holds similarly. For part 5, if u sends symbol n to its parent, then it must have received symbols n, n from both its children or it must have received symbols s, d from its children (see Figure 5.5).

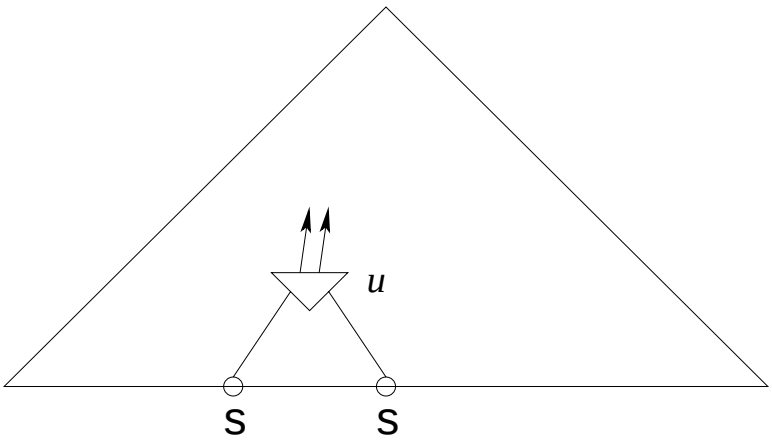


Figure 5.6: Illustration of the proof of Lemma 5.1

Now assume the lemma to hold for any node at level l (where $1 \leq l < n$) and consider node u at level $l+1$. Let v and w be children of u. Let $\mathcal{T}_v$ and $\mathcal{T}_w$ denote the subtrees rooted at v and w, respectively. Nodes v and w are at level l and the induction hypothesis applies to them. Suppose u receives symbols `s`, `s` from its children (see Figure 5.7). By the induction hypothesis, $\mathcal{T}_v$ and $\mathcal{T}_w$ contain sources whose destinations are outside $\mathcal{T}_v$ and $\mathcal{T}_w$. As before, both sources must use the link from u to its parent, and this is not possible for an edge-exclusive set. The only difference between the proof for part 1 in the base case and the induction step is that the induction hypothesis permits us to treat v and w as we did with the leaves in the base case. The remaining parts also use the induction hypothesis and use the same argument employed by the base case. ■

Theorem 5.2 *If each CST switch is configured using functions f_s and f_c, then the CST establishes the paths corresponding to the communications of the given edge-exclusive set.*

Proof: We prove the lemma by considering the following cases for any internal node u of the CST. Let $\mathcal{T}_u$ be the subtree rooted at any internal node u.

We consider three cases based on the symbol u sends to its parent.

Case 1 Suppose node u sends `s` to its parent. By part 2 of Lemma 5.1, $\mathcal{T}_u$ contains a source x such that its corresponding destination x' is outside $\mathcal{T}_u$. Consider

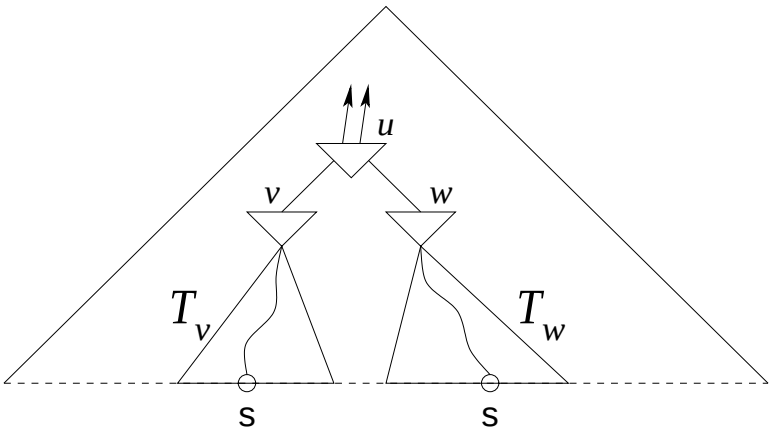


Figure 5.7: Illustration of the proof of Lemma 5.1

any internal node v on the path from x to u. Since x is in $\mathcal{T}_v$ and x' is not, by Lemma 5.1, part 2, v sends an s to its parent. Let w and z be the children of v, and let $\sigma_w = \mathsf{s}$ or $\sigma_z = \mathsf{s}$. Without loss of generality, let $\sigma_w = \mathsf{s}$. Clearly, $\sigma_z \neq \mathsf{s}$, Lemma 5.1, part 1. If $\sigma_z = \mathsf{d}$, then v sends an n to its parent, which contradicts our assumption. So, $\sigma_z = \mathsf{n}$. Thus every node in the path from x to u receives symbols s, n (with the s from the subtree containing x). From Figure 5.5, entries $f_c(\mathsf{s},\mathsf{n})$ and $f_c(\mathsf{n},\mathsf{s})$, it is clear that the CST establishes a physical path from x to the parent of u.

Case 2 Suppose node u sends symbol d to its parent. This is the dual of Case 1. An identical argument proves that there is a path in the CST from the parent of u to the destination x' in $\mathcal{T}_u$.

Case 3 Suppose u sends symbol n to its parent. Let w and z be the children of u. From Figure 5.4, two subcases are possible.

Subcase 3.1 Suppose both w and z send symbol n to u. By part 2 of Lemma 5.1, each source in $\mathcal{T}_w$ or $\mathcal{T}_z$ has its destination within the same tree; that is, there are no sources or destinations in $\mathcal{T}_w$ (or $\mathcal{T}_z$) left to match. Node u, therefore, correctly does nothing (see $f_c(\mathsf{n},\mathsf{n})$ in Figure 5.5).

Subcase 3.2 Suppose the children of u send symbols s, d to it. Without loss of generality, let w send s and z send d. By Lemma 5.1, parts 2, 3, and 4, there is a source x in $\mathcal{T}_w$ that matches a destination x' in $\mathcal{T}_z$. By Case 1 and Case 2, the CST correctly connects x to u and sets up a path from u to x'. In Figure 5.5, $f_c(\mathsf{s},\mathsf{d})$ shows that node u correctly connects these paths. ■

Note that edge-exclusive sets are also width-1 sets of a CST using half duplex links. Therefore, the results of this section may hold independent interest for such CSTs.

5.3 Edge-Exclusive Decomposition

In this section we present an algorithm to partition any communication set C into at most three edge-exclusive sets C_1, C_2, C_3 such that each of C_1, C_2, C_3 is edge-exclusive. We will call such a partitioning an *edge-exclusive decomposition* of C. Since communications from any edge-exclusive set can be established on the CST in one step, it follows that all communications from any width-1 communication set can be established on the CST in at most three steps.

Broadly speaking, the algorithm assigns a color to each communication in the set C such that no two communications with the same color share an edge of the tree. Clearly, all communications of any one color form an edge-exclusive set. We show that C can be colored with three colors.

Each communication of a width-1 set corresponds to a directed path between a source-destination pair (leaves) of the CST. Thus coloring the set of communications amounts to assigning colors to the directed edges of the CST. A *correct coloring* must:

- assign different colors to directed CST edges with the same end points (corresponding to the same undirected edge),
- assign the same color to all directed edges of a communication.

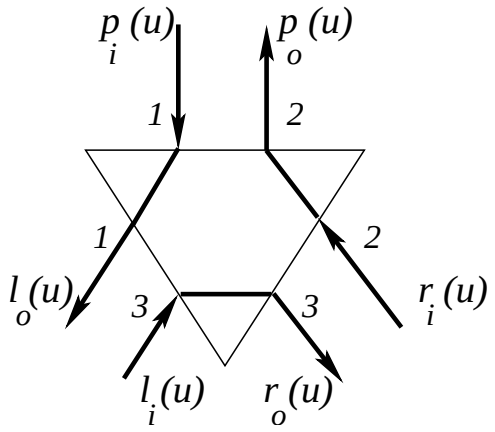


Figure 5.8: Incoming and outgoing edges of a switch

We will use a palette $\{1, 2, 3\}$ of three "colors" to color the communications. Edges that do not correspond to any communication will be assigned "color" 0 to indicate their status. We assume that the correspondence between communications and directed edges of the CST is known *a priori*. That is, a switch can match each incoming edge with an outgoing edge in accordance with a communication (if any). For ease of explanation we will use the following notation to distinguish between the incoming and outgoing edges of an internal node u of the CST (see Figure 5.8).

- $p_o(u)$ is an outgoing edge from u to the parent of u.
- $p_i(u)$ is an incoming edge from the parent of u to u.
- $l_o(u)$ is an outgoing edge from u to its left child.
- $l_i(u)$ is an incoming edge from the left child of u to u.
- $r_o(u)$ is an outgoing edge from u to its right child.
- $r_i(u)$ is an incoming edge from the right child of u to u.

We now detail the steps of the procedure and establish its correctness. Consider the procedure in Figure 5.9.

Step 1: This step defines the termination of the recursion. If u is a leaf, then its only edges are to its parent; these have already been correctly colored.

Procedure COLOR($\mathcal{T}, u, x, y$)
/* The procedure assigns a correct coloring to the subtree of $\mathcal{T}$ rooted at node u, given that the incoming edge $p_i(u)$ and outgoing edge $p_o(u)$ have been assigned colors x and y, respectively, where $x, y \in \{0, 1, 2, 3\}$ */
begin

1. If u is a leaf then return
   /* let v and w be the left and the right children, respectively of u */
2. Assign correct colors x_1 to $l_o(u)$, y_1 to $l_i(u)$, x_2 to $r_o(u)$, y_2 to $r_i(u)$
3. COLOR($\mathcal{T}, v, x_1, y_1$)
4. COLOR($\mathcal{T}, w, x_2, y_2$)

end.

Figure 5.9: Edge-Exclusive Decomposition Procedure

Step 2: The assignment of colors in this step depends on x, y and the correspondence between incoming and outgoing edges of u (as dictated by communications traversing u). For example, if $x = 1$ and $y = 2$ and the correspondence of u is shown in Figure 5.8, then clearly $l_o(u)$ has to be colored 1 and $r_i(u)$ has to be colored 2. This also implies that $l_i(u)$ and $r_o(u)$ have to be colored 3.

In general no more than three communications can traverse switch u (with three incoming and three outgoing edges) and at most two of these three communications can traverse the edge between u and its parent. Therefore, the edges of the third communication can always be correctly colored with a color from $\{1, 2, 3\}$. If an edge does not correspond to a communication, simply assign 0 to it.

Steps 3 and 4: These steps respectively color the subtrees at the children of u, given a correct coloring for the edges between them and u.

Theorem 5.3 *Every width-1 set of communications can be decomposed into at most three edge-exclusive sets.*

Proof: Call procedure COLOR($\mathcal{T}, root, 0, 0$), where *root* is the root of the CST $\mathcal{T}$. This colors the entire tree correctly in accordance with the given communication set C. Partition C into sets C_1, C_2 and C_3 such that for $i = 1, 2, 3$, C_i contains only

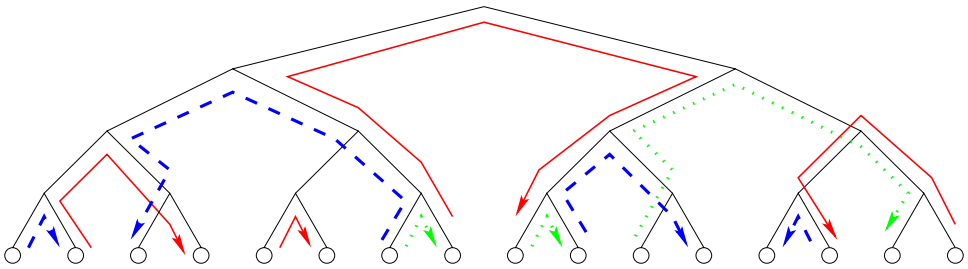


Figure 5.10: Decomposition of width-1 communication set into edge-exclusive sets. Sets are shown in solid, dashed, and dotted

these communications of C whose edges in $\mathcal{T}$ have been colored i. By virtue of the two conditions that make a coloring correct (see page 101), C_i is edge-exclusive. ■

Figure 5.10 shows an example of decomposing a width-1 communication set into three edge-exclusive sets (the sets are represented using solid, dashed, and dotted lines).

5.4 Concluding Remarks

In this chapter, we presented a one-step method to configure the CST to establish the communication paths of a width-1 communication set. We identified a class of communication sets called edge-exclusive sets for which the above method applies. Then, we showed that every width-1 communication set can be decomposed into at most three edge-exclusive sets.

Theorem 5.3 shows that every width-1 set of communications can be performed (including configuration of switches under local control) in at most three batches. Together with the schedules implied by Theorems 3.8, 3.15, and 3.17, these results provide a comprehensive approach to perform communications on the CST.

Chapter 6

Segmentable Bus Implementation

In Section 2.2 we described the segmentable bus and explained the importance of implementing it with small bus delay. In this chapter, we build on the techniques of Chapters 3 and 5 to derive a segmentable bus implementation with small bus delay. In Chapter 3 we presented a segmentable bus implementation as a special case of oriented well-nested sets. The treatment here addresses many issues (such as concurrent writes and processor word-size) not considered in that result.

A good segmentable bus implementation immediately translates to a good implementation of the HVR-Mesh [4], Basic R-Mesh [35] and the polymorphic processor array [30], models on which many algorithms have been designed. We also show (in Chapter 7) that a segmentable bus can be used as a building block for implementing an LR-Mesh (see Section 7.2).

We present two approaches for implementing segmentable buses; one is based on a hardware solution that builds on the CST, while the other uses an algorithmic approach. The problem of implementing a segmentable bus allows each processor, i, to assume only the answers to the following questions: (a) How does processor i want to configure its segment switch (open or closed—see Section 2.2)? (b) Does processor i want to write to its bus segment? This ensures that its solution matches the functional description of the segmentable bus in Section 2.2.

In the next section we briefly describe the two approaches to implementing the segmentable bus. Sections 6.2 and 6.3 detail the approaches. In Section 6.4 we summarize our results and make some concluding remarks.

6.1 Our Approaches

We present two approaches for implementing a segmentable bus, both employing a balanced tree. The first approach, based on the CST, is suitable for large processors of word-size $\Omega(\log N)$ bits (where N is the number of processors on the segmentable bus). In such a processor, one step can accommodate $\Omega(\log N)$ gate delays[1]. This approach draws upon the techniques presented in Chapter 5, at the same time exploiting properties of communication patterns possible on a segmentable bus. Here the main idea is to configure the CST to establish a dedicated path from each writer to all readers of a bus segment. Once these paths are established, data communication is seamless. Therefore we only detail the configuration phase. In general, we will admit concurrent writes to a bus segment. But if the application guarantees exclusive writes, then further improvement in cost and performance is possible.

The second approach is suitable for smaller processors of word-size $\Theta(w)$ bits where $\log\log N \leq w \leq \log N$. This approach uses a normal 2^w-ary tree algorithm [24] and runs in $O\left(\frac{\log N}{w}\right)$ steps, each of $\Theta(w)$ delay. A normal tree algorithm proceeds level by level in the tree. Here we use a normal tree algorithm to translate local information from processors to a configuration of global relevance. We permit our algorithm to use any implementation of a 2^w-processor segmentable bus. (The first approach provides one such implementation. A class of structures that is capable of implementing normal tree algorithms efficiently are multiple bus networks (MBNs) [15].)

The two methods collectively allow $\Theta(\log\log N)$-bit processors to use $\Theta\left(\frac{\log N}{\log\log N}\right)$ steps, each of $\Theta(\log\log N)$-delay, or larger $\Theta(\log N)$-bit processors to use a constant number of $\Theta(\log N)$-delay steps, or all shades in between. In both approaches, the idea is to translate the local information at processors to global information that represent the connectivity of the segmentable bus.

[1] A processor of word-size w can usually address a $2^{\Theta(w)}$-location memory in one step. A decoder for such an addressing requires $\Theta(w)$ gate delays.

6.2 Methods for Large Processors

In this section we use the CST to implement a segmentable bus. We first present a method to implement a special case of a segmentable bus (called the *right-oriented segmentable bus*), in which we assume that for each bus segment only the leftmost processor of each segment writes and all other processors read. A left-oriented segmentable bus is similar with the rightmost processor of each segment as its only writer. Next we use oriented segmentable buses to derive an implementation of a (general) segmentable bus with only exclusive writes. Finally, we augment this implementation to support concurrent writes.

The main result of this section is that an N-processor segmentable bus can be implemented on a CST to run in $\Theta(1)$ steps. The difference between the exclusive writes and concurrent write implementations is in the complexity of the hardware and constants in the running time.

As noted earlier, the problem boils down to configuring the CST (using local information at leaves) to establish communication paths. Our approach to configuring the CST builds on the technique in Chapter 5. As in Chapter 5, the CST operates as follows.

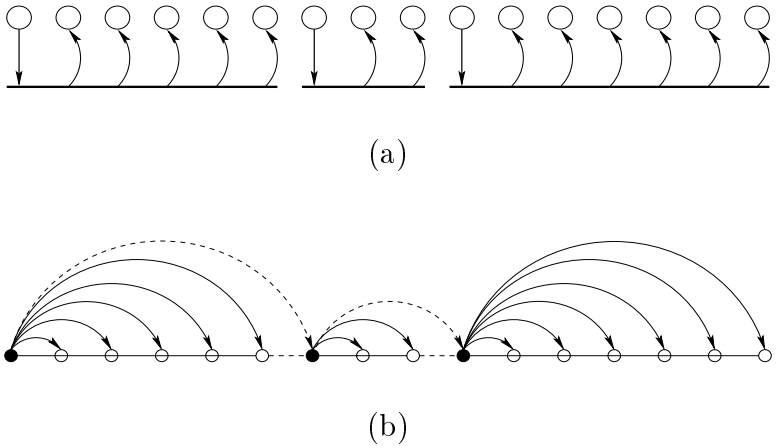
1. Each switch configures its communication unit based on the control symbols it receives from its children.

2. Each switch generates a control symbol (that captures all relevant information from its descendants) to send to its parent.

3. The CST configuration algorithm uses a constant number of control symbols.

6.2.1 Implementing an Oriented Segmentable Bus

Without loss of generality, let the N-processor segmentable bus be right oriented. Therefore we assume that the leftmost processor of each bus segment writes to the segment and all other processors read (see Figure 6.1(a)). Consider a configuration of the segmentable bus with k segments.



(a)

(b)

Figure 6.1: Right oriented segmentable bus

For $1 \leq i \leq k$, let the i^{th} segment be $S_i = (w_i, e_i)$ where w_i is the index of the leftmost processor (writer) of the segment and e_i is the index of the rightmost processor of the segment. Clearly, $w_i \leq e_i$ for all i. Also observe that if w_{i+1} exists, then $e_i \leq w_{i+1}$. Since all communications on segment S_i are from w_i toward e_i, we will say that the segment S_i "starts at w_i" and "ends at e_i". For technical reasons, we assume that for all $1 \leq i \leq k$, $e_i = w_{i+1}$; that is segment S_i ends at the same processor as the one at which S_{i+1} starts. Consequently, the writer w_i of segment S_i writes also to processor w_{i+1} (see dashed communications in Figure 6.1(b)), which simply ignores the value read. Note that this makes it possible for a processor to be a writer of one segment and a reader of the previous segment.

Leaves (processors) of the CST generate control symbols from the set $S = \{\mathsf{w}, \mathsf{r}\}$ (the symbol w corresponds to a writer processor whereas symbol r corresponds to a non writing processor). These symbols propagate up the tree, configuring switches level-by-level. Each switch (internal node) receives control symbols (from set $S = \{\mathsf{w}, \mathsf{r}\}$) from its children. It uses these symbols to produce a symbol (again from the set S) for its parent. Let $\mathcal{C}$ be the set of configurations of the communication unit of a switch. Then each CST switch can be viewed as two functions, $g_s : S \times S \to S$ providing a symbol (see Figure 6.2) and $g_c : S \times S \to \mathcal{C}$ providing a configuration (see Figure 6.3).

g_s	r	w
r	r	w
w	w	w

Figure 6.2: The function g_s for segmentable bus

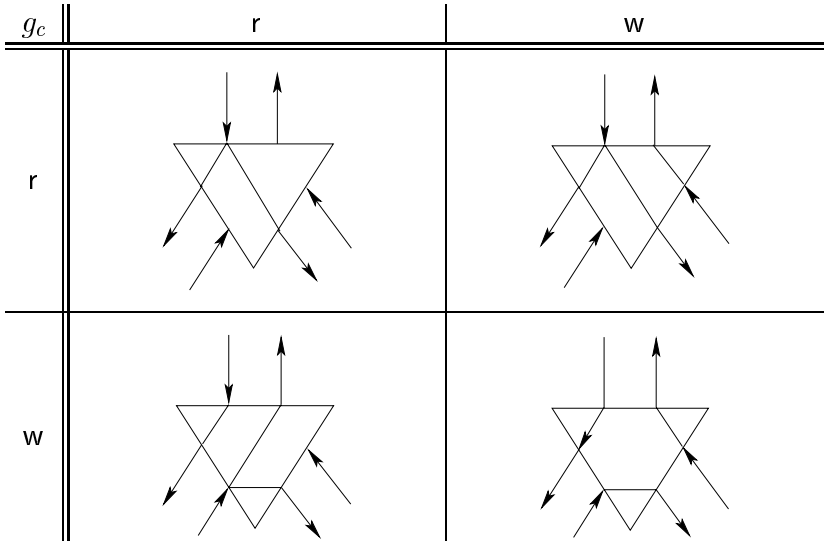


Figure 6.3: The function g_c for segmentable buses

Recall that we assume the CST leaves to be labeled in increasing order from left to right. Thus a statement such as $u \leq v < w$ is to be interpreted as leaf u is not to the right of v and leaf v is to the left of w. Let u be any node of the CST. Let $\mathcal{T}_u$ denote the subtree rooted at u. If u is an internal node, then its left (resp., right) subtree is the subtree of $\mathcal{T}_u$ rooted at the left (resp., right) child of u.

Lemma 6.1 *For any internal node u of the CST, the functions g_s and g_c establish paths as follows.*

1. *If u sends symbol* r *to its parent, then the algorithm connects the incoming edge from the parent of u to all leaves of $\mathcal{T}_u$.*

2. *If u sends symbol* w *to its parent, then let the writers in $\mathcal{T}_u$ be w_1, w_2, $\cdots$, w_α such that $w_1 \leq w_2 \leq \cdots \leq w_\alpha$.*

   (a) *The incoming edge from the parent of u is connected to all leaves $z \leq w_1$ of $\mathcal{T}_u$.*

   (b) *For $1 < i < \alpha$, writer w_i is connected to each leaf z such that $w_i < z \leq w_{i+1}$.*

   (c) *The last writer w_α is connected to each leaf $z > w_\alpha$ of $\mathcal{T}_u$ and to the outgoing edge from u to its parent.*

Proof: We proceed by induction on the level $l \geq 1$ (where leaves of the CST are at level 0) of an internal node u. If $l = 1$, then u is the parent of two leaves. If u sends an r to its parent, then both children of u are readers who send r, r to u. From Figure 6.3, it is clear that part 1 holds for the base case.

If u sends w to its parent, then the three cases correspond to $g_c(\mathsf{r},\mathsf{w})$, $g_c(\mathsf{w},\mathsf{r})$, and $g_c(\mathsf{w},\mathsf{w})$ in Figure 6.3. In the first two cases there is only one writer ($\alpha = 1$). So parts 2a and 2c apply to this writer. It is simple to verify that the case holds for parts 2a, 2b and 2c.

Assume the lemma to hold for any node at level $l \geq 1$, and consider node u at level $l + 1$. Let v and w be the left and right children of u.

If u sends r to its parent, then both v and w send r to u (see Figure 6.2). By the induction hypothesis, the CST establishes a path from u to all leaves of v and w. The switch configuration of $g_c(\mathsf{r},\mathsf{r})$ (Figure 6.4(a)) ensures that the parent of u is connected to all leaves of u.

Suppose u sends symbol w to its parent, then at least one of v or w must send w to u. We now consider three cases.

Case 1 Suppose v sends r to u and w sends w to u. By the induction hypothesis, we have the situation depicted in Figure 6.4(b). We only need observe that all leaves $z \leq w_1$, of $\mathcal{T}_u$ are indeed connected to, from the parent of u as required, and that w_α is connected to the parent of u.

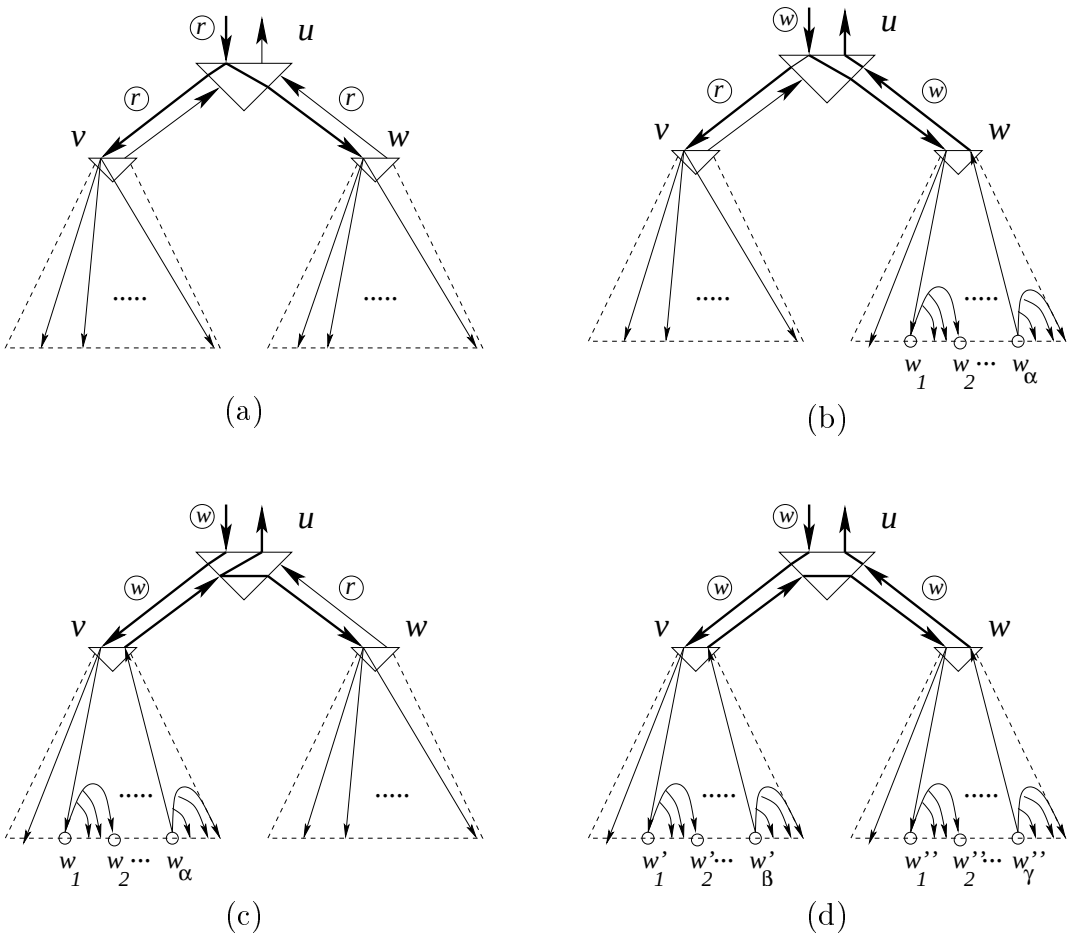


Figure 6.4: Illustration of the proof of Lemma 6.1

Case 2 Suppose v sends w to u and w sends r (see Figure 6.4(c)). Observe that the parent is connected to every leaf $z \leq w_1$, and that w_α is connected to the parent and every leaf $z > w_\alpha$, as required.

Case 3 Suppose both v and w send w to u (see Figure 6.4(d)). Let the writers of $\mathcal{T}_v$ and $\mathcal{T}_w$ be w'_1, w'_2, $\cdots$, w'_β, and w''_1, w''_2, $\cdots$, w''_γ enumerated from left to right. Then for $\mathcal{T}_u$, $w_1 = w'_1$ and $w_\alpha = w''_\gamma$. Notice (especially between w'_β and w''_1) that all required connections are connected. ■

Theorem 6.2 *A CST configured by the functions g_s and g_c can perform all communications of a right-oriented segmentable bus in one step.* ■

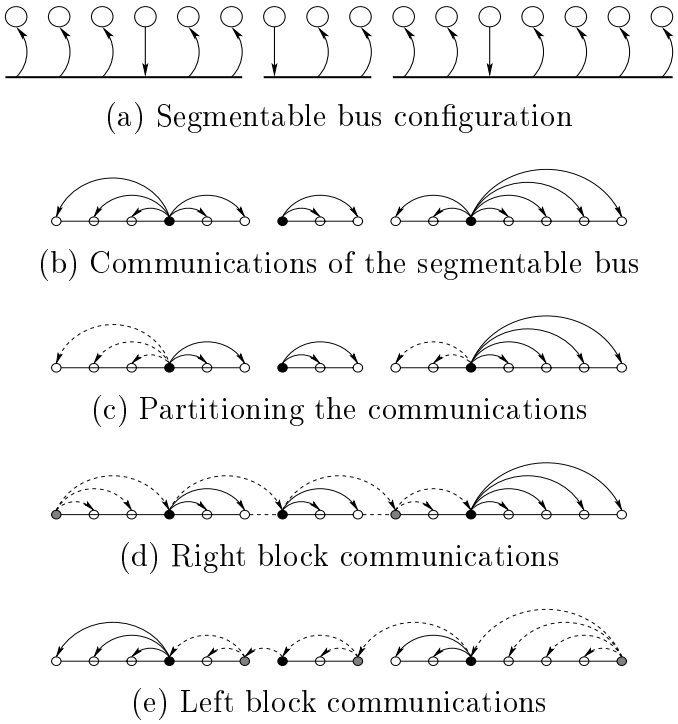
Recall that we modified the definition of the right-oriented segmentable bus to require writers to be readers as well. We now provide the intuition for this modification. As Theorem 6.2 establishes, the modified communication set requires only two symbols. As noted earlier, the spurious read from the previous segment can be discarded by the writer. On the other hand, with the original definition of a right-oriented segmentable bus (see Figure 6.1(a)), the algorithm would have to distinguish between switches whose leaves are all readers, all writers, and some readers/writers. The modification lumps the latter two cases into one.

Clearly the method of this section readily translates to one for a left-oriented segmentable bus.

6.2.2 Segmentable Bus with Exclusive Writes

In this section we use the oriented segmentable bus implementation of Section 6.2.1 to realize a segmentable bus in which each segment has at most one writer (not necessarily at one end of the segment).

Generally speaking, the idea is to partition the given exclusive write, segmentable bus communications into two blocks as follows. Recall that processors are indexed in increasing order from left to right. Since only writes have to be exclusive, each segment, S_i, has exactly one writer, w_i (say). Call processor $j \neq w_i$ of segment S_i a *left reader* (resp. *a right reader*) iff $j < w_i$ (resp. $j > w_i$). Partition the segmentable bus communications so that one block (the left block) contains only communications from the writers to their readers to the left and the other block (right block) contains only the communications from the writers to their readers to the right. Figure 6.5 shows an example of a segmentable bus configuration and Figure 6.5(b) show its communication patterns. Figure 6.5(c) shows the partitioning of the communications into two blocks (shown solid and dashed). Figure 6.5(d) shows the communications of the right block (shown solid) and some dummy communications (shown dashed). No data are sent on the dummy communications (they exist only to simplify the solution). Figure 6.5(e) shows the left block similarly. Note that the communications of the right (resp., left) block are the same as the communications of a right (resp., left) oriented segmentable bus. Here we implement the communications of the right



(a) Segmentable bus configuration

(b) Communications of the segmentable bus

(c) Partitioning the communications

(d) Right block communications

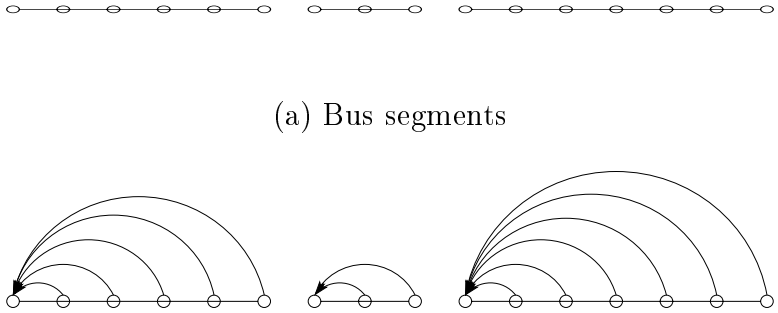
(e) Left block communications

Figure 6.5: Implementation of a segmentable bus with exclusive writes

block and the communications of the left block separately using the method presented in Section 6.2.1. Note that when implementing the communications of the right (resp., left) block, many paths are established (shown dashed in Figure 6.5(c) and (d)) but not used to transfer data.

With one full duplex link between each node and its parent, the right and the left blocks will have to be scheduled separately, as they could form a width-2 communication set. If each link is replaced by two full duplex links, then the communications from both blocks can be performed simultaneously in one step. (This amounts to setting $k = 2$ in the remark at the end of Chapter 3.)

Theorem 6.3 *A CST with two full duplex links per edge can perform all communications of a segmentable bus with exclusive writes in one step.* ■



(a) Bus segments

(b) Collection phase

(c) Broadcast phase

Figure 6.6: Implementation of a segmentable bus with concurrent writes

6.2.3 Segmentable Bus with Concurrent Writes

Here we present a method to implement a segmentable bus that admits concurrent writes to a segment. As explained in Section 2.2, concurrent writes are resolved using resolution rules. In this section we discuss the COMMON rule, in which all writers write the same value (other concurrent write rules can also be implemented in a similar way). In a concurrent writes implementation, each processor is a potential writer. In other words, paths should be established from each processor to all other processors. We accomplish this in two phases called the *collection* and *broadcast* phases. The collection phase collects the result of all concurrent writes (bus-value) to a segment into a fixed processor; we choose the leftmost processor of each segment as the *collector*. The second phase broadcasts the bus-value from the collector of each segment to the remaining processors of the segment. Consider the segmentable bus of Figure 6.6(a). Figure 6.6(b) shows the communications of the collection phase, and

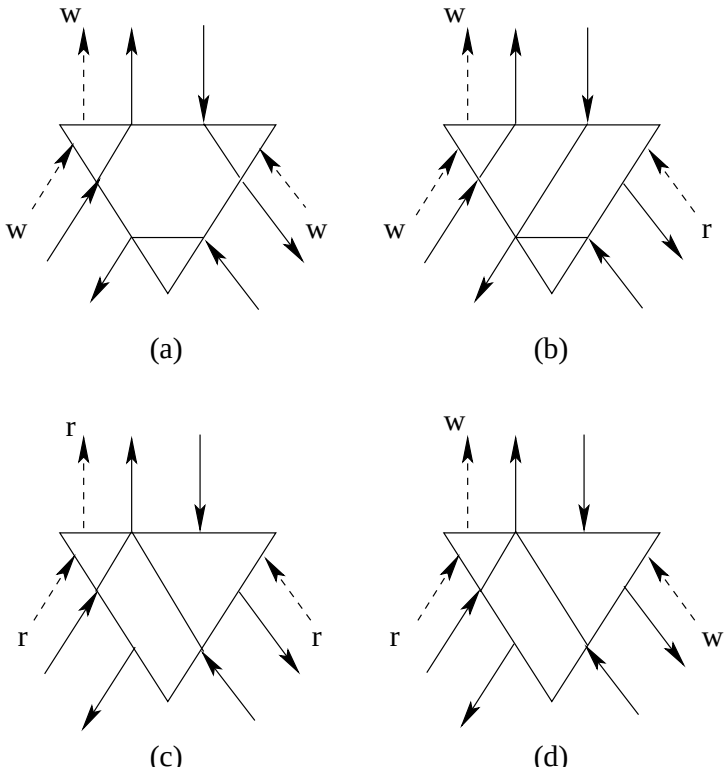


Figure 6.7: Reversing the directions of data flow

Figure 6.6(c) shows communications of the broadcast phase. The broadcast phase is simply the communications of a right-oriented segmentable bus, so the method of Section 6.2.1 suffices for its implementation.

The communications of the collection phase are the "dual" of those of the broadcast phase. In other words, if each switch is configured the same way as in the broadcast phase, except for reversing the directions of information flow, we end up with the configuration of the collection phase (see Figure 6.7). The only point that requires further elaboration is the case where two incoming lines are "connected" together. For the COMMON rule, these lines could simply be ORed (assuming 0 for no writes). For other rules, use other simple functions requiring constant hardware.
Remarks: Connections (data paths) of both phases could be cascaded to provide one seamless path from writers to readers.

Theorem 6.4 *A CST with two full duplex links per edge can perform all communications of a segmentable bus with concurrent writes in one step.* ■

Remarks: The difference between results of Theorems 6.3 and 6.4 is that additional resolution hardware is employed for concurrent writes.

6.3 Method for Small Processors

In this section we present an approach to implement segmentable buses with smaller processors of word-size $\Theta(w)$ bits, where $\log\log N \leq w \leq \log N$. This approach implements an N-processor segmentable bus to run in $\Theta(\frac{logN}{w})$ steps.

6.3.1 Another Segmentable Bus Implementation

This implementation of an N-processor segmentable bus uses a 1-step, 2^w-processor segmentable bus as a building block. We will refer to this building block as the *base segmentable bus*. (Section 6.2 gives one implementation of the base segmentable bus.)

Construction: We now give a recursive description of this new implementation of an N-processor segmentable bus. Without loss of generality, let $N = 2^{x(w-1)+1}$, where w reflects the processor word-size and relates to the number of processors in the base segmentable bus; $x \geq 1$ is an integer. Let $\mathcal{S}(x)$ denote the $\left(2^{x(w-1)+1}\right)$-processor segmentable bus implementation.

- If $x = 1$, then $N = 2^w$. Here $\mathcal{S}(1)$ is the base segmentable bus.
- If $x > 1$, then construct the segmentable bus as shown in Figure 6.8.

Divide the $2^{x(w-1)+1}$ processors into $\frac{N}{2^w} = 2^{(x-1)(w-1)}$ groups (or sets) $G(1), G(2), \cdots, G(2^{(x-1)(w-1)})$, each with 2^w contiguous processors. Connect processors in each group by the base segmentable bus $\mathcal{S}(1)$. Let p_i and q_i be the leftmost and the rightmost processors in group $G(i)$. Recursively connect the $2^{(x-1)(w-1)} \times 2 = 2^{(x-1)(w-1)+1}$ processors, p_i and q_i (where $1 \leq i \leq 2^{(x-1)(w-1)}$) using a $\left(2^{(x-1)(w-1)+1}\right)$-processor segmentable bus, $\mathcal{S}(x-1)$.

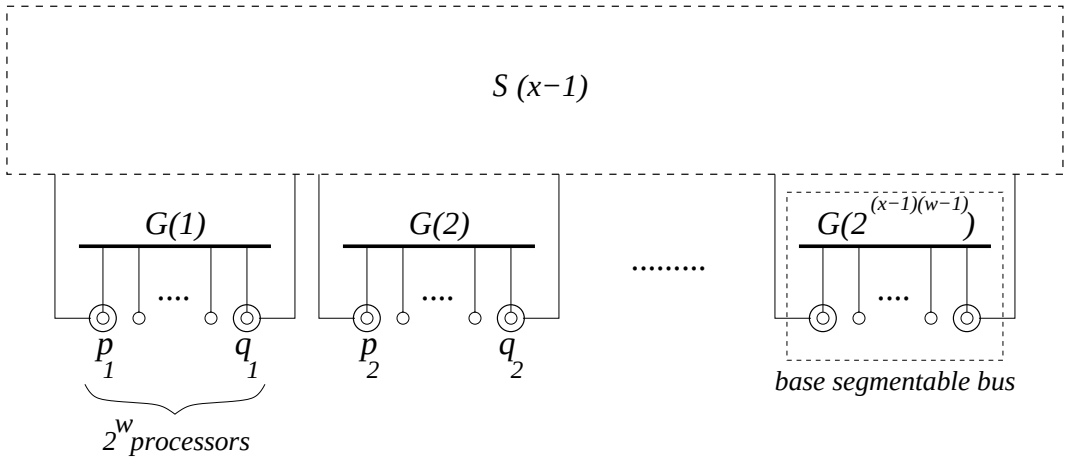


Figure 6.8: Structure of a segmentable bus implementation $\mathcal{S}(x)$

An Example: We now illustrate this construction for $w = 2$ and $x = 3$, so $N = 16$. Figure 6.9 shows the structure of $\mathcal{S}(3)$. We also illustrate the operation of $\mathcal{S}(3)$ using

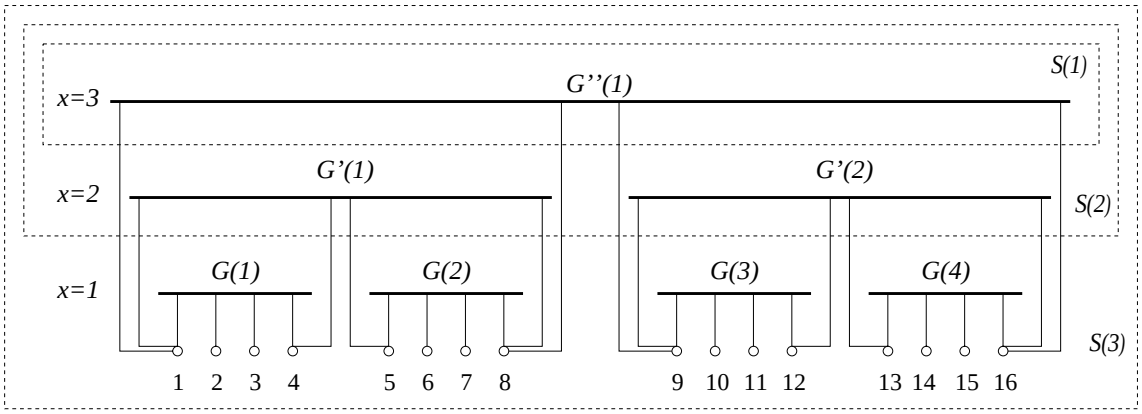


Figure 6.9: Structure of $\mathcal{S}(3)$

this example. Suppose the function of $\mathcal{S}(3)$ is as shown in Figure 6.10(a) where processors 4, 6, 9, and 15 open their segment switches. Suppose the segmentable bus uses a COLLISION writing rule (see also Section 2.2). Recall that under the COLLISION rule, a collision symbol, "#", is written to a segment, if more than one processor attempts a write to that segment. Let processors 3, 4, 5, 7, 8, 14, and 15

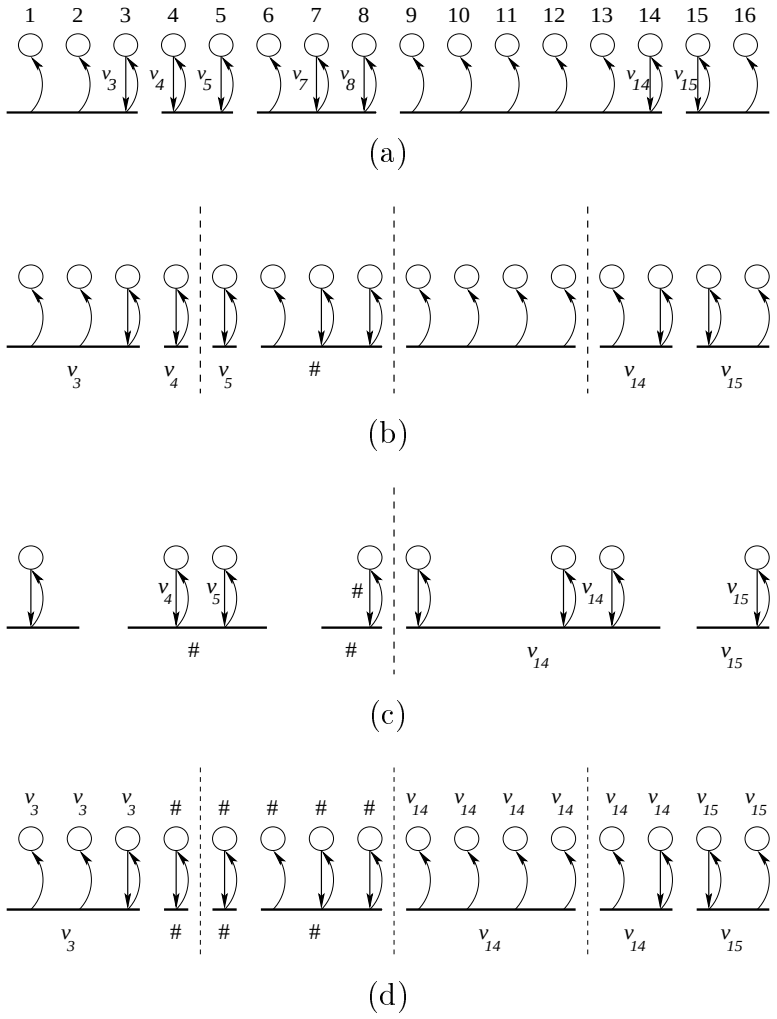


Figure 6.10: An illustration of the functioning of $\mathcal{S}(3)$

write values v_3, v_4, v_5, v_7, v_8, v_{14}, and v_{15}, respectively. In Figure 6.10, we follow the convention that only writers have arrows to the segmentable bus. The symbol on a bus segment is the bus-value and a symbol on a write arrow is the value written. The symbol above a processor is the value read.

Figure 6.10(b) shows the first step. Four groups $G(1) = \{1, 2, 3, 4\}$, $G(2) = \{5, 6, 7, 8\}$, $G(3) = \{9, 10, 11, 12\}$, and $G(4) = \{13, 14, 15, 16\}$ are formed. Each processor opens or closes its switch based on local data and writers write their data to the bus. The values on each segment (see Figure 6.10(b)) represent the value read

by all processors incident on that segment. Note that only the leftmost processor and the rightmost processor of each group (i.e., processors 1, 4, 5, 8, 9, 12, 13, and 16) are included in the next step. These processors determine the settings of their segment switches for the next step as follows. All leftmost processors (i.e., processors 1, 5, 9, and 13) retain their original segment switches' settings. Each rightmost processor of each group (i.e., processors 4, 8, 12, and 16) determines if it is on the same segment as the leftmost processor of that group. If it is, then the rightmost processor closes its switch in the next step. If not, then the rightmost processor opens its switch in the next step. Figure 6.10(c) shows the second step of the algorithm (processor groups are $G'(1) = \{1, 4, 5, 8\}$ and $G'(2) = \{9, 12, 13, 16\}$). Note that the rightmost processor of group $G(2)$ (processor 8 in the first step) opens its switch in the second step even though its switch was closed in the first step. This is because (in the first step) processor 8 is not on the same segment as processor 5. Again, all processors write their values, and the leftmost and the rightmost processors determine their new switch settings for the third step as explained before. This process is repeated till the number of leftmost and rightmost processors reduces to 2^w, at which time a single base segmentable bus suffices. Now, a reverse process is applied, and the collected data is broadcast to the appropriate processors. Figure 6.10(d) shows the final value read by each processor.

Operation of $\mathcal{S}(x)$: We now generalize the idea of the above example. Initially each processor has its local information; that is, each processor knows whether it is segmenting the bus and is aware of the value (if any) it is to write to the bus. As we saw in the example, the aim is to determine the bus-value for each processor on the segmentable bus. Let $P = \{1, 2, \cdots, N\}$ be the set of processors on $\mathcal{S}(x)$. For any $R \subseteq P$ and any $i \in P$, let $v(R, i)$ be the bus-value at processor i, assuming that only the writers in subset R write to the segmentable bus. For example, if $N = 10$, and suppose that processors $1, 5, 7$ and 10 write values v_1, v_5, v_7 and v_{10}, then for $R = \{z_1, z_2, z_3, z_4, z_5\}$, the quantity $v(R, z_2)$ would be the value read by processor z_2 assuming only processors 1 and 5 write values v_1 and v_5; the segment switches' settings are not altered by a choice of the set R. The procedure for $\mathcal{S}(x)$ is as follows.

Phase 1: Processors of $G(j)$ (where $1 \leq j \leq 2^{(x-1)(w-1)}$) determine the following:

- bus-value $v(G(j), z)$ for each $z \in G(j)$.
- processor q_j determines if it is in the same segment as processor p_j.

Phase 2: Let $\alpha_j = v(G(j), p_j)$ and $\beta_j = v(G(j), q_j)$ be the "local values" read by the end processors of the group. Processor p_j cuts the bus to its left if and only if it was supposed to do so initially (at the start of Phase 1). Processor p_j cuts the bus to its left if and only if in Phase 1 it determined that it was not in the same segment as p_j. Processors p_j, q_j write α_j, β_j, respectively.

With the above "local information," as write values and segment switches states of the processors p_j, q_j (for all $1 \leq j \leq 2^{(x-1)(w-1)}$) recursively determine bus-values and segment switches states on $\mathcal{S}(x-1)$. Let γ_j and ρ_j be the new bus values of p_j and q_j, respectively.

Phase 3: Processors p_j and q_j write γ_j and ρ_j on the local base segmentable bus of $G(j)$. If processor $z \in G(j)$ receives a value ξ, then ξ is its final bus-value. Otherwise, it retains its bus-value from Phase 1.

In Phase 1, getting the bus-value, $v(G(j), z)$, is simply a matter of segmenting the local $\mathcal{S}(1)$ as specified and writing to and reading from the segmentable bus. Processor q_j determines if it is on the same bus as p_j by waiting for a step to receive a signal issued by p_j. So Phase 1 runs in two steps. Phase 3 again is a matter of using the local base segmentable bus and runs in one step.

The correctness of the procedure stems from the following facts.

- Phase 1 correctly determines the "local" bus-values within groups.
- Adjacent groups $G(j)$ and $G(j+1)$ can interact only through processors q_j and p_{j+1}.
- Phase 2 recursively captures the final bus-value for processors p_j and q_j.
- Phase 3 conveys the final values locally.

Let $\mathcal{S}(x)$ require $T(x)$ steps. Clearly, $T(1) = 1$ and from the explanation above $T(x) = T(x-1)+3$. Solving this recurrence gives $T(x) = 3x-2 = \Theta(x) = \Theta\left(\frac{\log N}{w}\right)$.

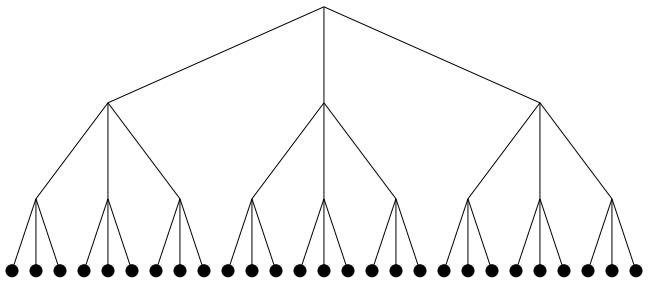


Figure 6.11: A balanced ternary ($k = 3$) tree of height 3

Given that a one-step 2^w-processor segmentable bus (with each processor of word-size w) can be constructed (Section 2.2) we have the following result.

Theorem 6.5 *For any w where $\log\log N \leq w \leq \log N$, the proposed implementation of an N-processor segmentable bus (using $\Theta(w)$-bit processors) runs in $\Theta\left(\frac{\log N}{w}\right)$ steps.* ■

Remark: The reason we require $w = \Omega(\log\log N)$ is that two processors of the segmentable bus are part of all x levels of the recursion. These processors are connected to x different base segmentable buses. Consequently, their word-sizes must be at least $\log x = \log\left(\frac{\log N}{w}\right)$. That is, if $w = \Omega(\log\log N)$, then the segmentable bus can operate as stated. If $w = o(\log\log N)$, the ideas presented by Vaidyanathan *et al.* [49] could be used.

Using k-ary Trees: For any $k \geq 2$, a balanced k-ary tree of height h has $N = k^h$ leaves, each at a distance of h from the root, and each internal node has k children (Figure 6.11 shows a balanced ternary tree ($k = 3$) of height 3).

A k-ary tree algorithm on $N = k^h$ inputs proceeds level-by-level from the leaves to the root of a k-ary tree. Each node u of the tree has a value $\alpha(u)$ associated with it. The value of a leaf is an input. The value $\alpha(u)$ of an internal node u with children $u_1, u_2, \cdots, u_k$ is a function $f(\alpha(u_1), \alpha(u_2), \cdots, \alpha(u_n))$. The value $\alpha(u_r)$ of the root r is the output of the algorithm. Reversing the direction of a k-ary tree algorithm generates outputs at the leaves. There is a clear parallel between

a k-ary tree algorithm and the segmentable bus implementation described above (see Figure 6.10 for which $k = 4$). Thus any platform suitable for a k-ary tree algorithm works for a segmentable bus as well. Dharmasena [15] proposed a multiple bus network (MBN) to run a k-ary tree algorithm on $N = k^h$ node in h steps. This MBN can also serve to implement a $\Theta\left(\frac{\log N}{w}\right)$-step segmentable bus connecting N processors.

6.4 Concluding Remarks

We have presented two approaches for segmentable bus implementation using binary trees. The first is suitable for large word-size processors and has variations that accommodate different writing abilities. The second approach achieves the implementation as a k-ary tree.

A Horizontal-Vertical Reconfigurable Mesh (HV-R-Mesh) [4] is an R-Mesh with a segmentable bus in each row and column. The bit model HV-R-Mesh [21] is a fine-grained version of the (word model) HV-R-Mesh with processors of constant size (like the PEs of the SRGA architecture). Theorem 3.19 extends to the following result.

Theorem 6.6 *If the SRGA architecture can support an N-leaf CST per row and column, then it can emulate any step of a bit-model $N \times N$ HV-R-Mesh in two steps.* ■

Chapter 7

Implementing the Linear Reconfigurable Mesh

As described in Chapter 1, most work on dynamically reconfigurable models such as the (R-Mesh) assumes "unit-cost" buses and entirely skirts the issue of bus delay. This makes the R-Mesh very difficult to implement. A more conservative "log-cost" measure [32] assigns a $\log N$ delay to a bus spanning N processors. While this measure is reasonable for a fixed bus, it does not capture the complexities arising from the ability of the LR-Mesh to configure its buses in an exponential number of ways. In this chapter we introduce a new measure of bus delay called "bends-cost" that considers the delay of a bus to be proportional to the number of times it bends between rows and columns of the LR-Mesh. We show that there exists an LR-Mesh implementation for which bends-cost is a faithful measure of the actual bus delay. This implementation uses a segmentable bus implementation. Consequently, our results are expressed in terms of Δ, the delay introduced by a segmentable bus spanning N processors. It should be noted that the specific implementation of the segmentable bus presented in Chapter 6 bounds Δ to be $O(\log N)$. The method proposed in this chapter is general enough to accommodate future improvements in the value of Δ, however.

We now describe our results in this chapter in a little more detail. Our results are primarily for LR-Meshes with "semimonotonic" buses. In any given step of such an LR-Mesh, all buses are laid out in some general orientation with respect to the underlying processor array (such as top to bottom, or left to right); Section 7.1 defines semimonotonic buses formally. Many fundamental algorithms (such as those for prefix

sums, multiple addition and sorting) run on LR-Meshes with semimonotonic buses [22, 32, 36]. We prove that each step of an $N \times N$ (unit-cost) LR-Mesh can be run in $O\left(\left(\frac{\log N}{\log D - \log \Delta}\right)^2\right)$ time on a $\Theta(N) \times \Theta(N)$ bends-cost LR-Mesh whose buses have a delay of at most D. For some special cases this time overhead can be reduced further to $O\left(\frac{\log N}{\log D - \log \Delta}\right)$. In particular, if $D = N^\epsilon$ for an arbitrarily small constant $\epsilon > 0$, then the running times of the the bends-cost LR-Mesh algorithms are to within a constant of their ideal (unit-cost) LR-Mesh counterparts. One implication of this result is that with N^ϵ delay, a $\Theta(N) \times \Theta(N)$ bends-cost LR-Mesh can perform prefix sums of N bits, add N b-bit numbers and sort N inputs in constant time. To our knowledge, this is the first general result to produce constant time algorithms on reconfigurable models without using the unit-cost assumption for bus delay.

We also present results for simulating the LR-Mesh (whose buses are not necessarily semimonotonic) and the more general R-Mesh on reconfigurable models with limited delay buses.

In the next section we introduce some definitions and properties of the LR-Mesh. In Section 7.2 we describe the bends-cost measure and an LR-Mesh implementation for which the bends-cost measure models bus delay accurately. Section 7.5 is devoted to the simulation of a unit-cost LR-Mesh with semimonotonic buses on a bends-cost LR-Mesh. Section 7.6 presents results for more general bus configurations. In Section 7.7 we summarize our results and make some concluding remarks.

7.1 Preliminaries

In this section we describe some properties of the LR-Mesh and define some terms. Recall that an $R \times C$ LR-Mesh consists of an R-row, C-column array of processors connected by an underlying mesh (see Figure 7.1). Each processor in an LR-Mesh has four ports (called North, South, East, and West ports in the obvious manner, and abbreviated N, S, E, and W).

Linear Bus Types: A linear bus can be *cyclic* (see dotted bus in Figure 7.1) or *acyclic*. The *row sequence* (resp., *column sequence*) of an acyclic linear bus is the

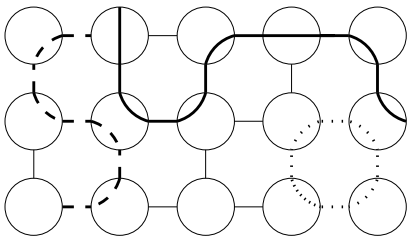


Figure 7.1: Examples of buses in a 3×5 LR-Mesh

sequence of row numbers (resp., column numbers) traversed when one traces the path of the bus from one of its end points to the other. For example, the row sequence of the bus shown solid in Figure 7.1 is $\langle 0, 1, 0, 1 \rangle$, as the (left end of the) bus starts at a port of a processor in row 0, moves to a processor at row 1, comes back to row 0 and finally ends in a port of a processor in row 1. (Note that reversing the sequence also produces a valid row sequence of the bus.) The column sequence of the above bus is $\langle 1, 2, 3, 4 \rangle$. The row and column sequences of the bus shown dashed in Figure 7.1 are $\langle 0, 1, 2 \rangle$ and $\langle 1, 0, 1, 0 \rangle$, respectively. A bus is *row monotonic* (resp., *column monotonic*) if its row sequence (resp., column sequence) is monotonic (either non-increasing or non-decreasing). The solid bus of Figure 7.1 is column monotonic but not row monotonic, whereas the dashed bus is row monotonic but not column monotonic. A bus that is row monotonic or column monotonic is said to be *semimonotonic*. An *incremental* bus is a row (resp., column) monotonic bus for which any two consecutive elements of its column (resp., row) sequence differ by 1. An incremental column monotonic bus moves up or down by at most one row at a time. The above ideas also apply to pieces of a linear bus. For example, in Figure 7.1 the piece of the solid bus between columns 1 and 2 is column monotonic but not row monotonic.

An LR-Mesh configuration is said to be *row monotonic* if every bus in the configuration is row monotonic. A *column monotonic* configuration is defined similarly.

Definition 7.1 A configuration that is row monotonic or column monotonic is said to be *semimonotonic*. An LR-Mesh configuration is *incremental*, if every bus in the configuration is incremental. ■

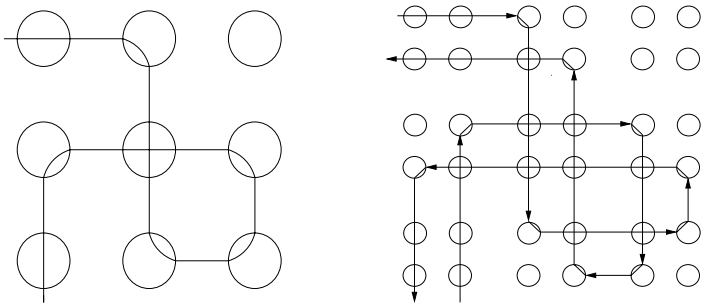


Figure 7.2: Replacing a linear, acyclic bus by two "directional buses"

7.1.1 Exploiting Features of the LR-Mesh

Here we first note some previous results on LR-Meshes. Then we use these to derive some properties of LR-Mesh algorithms that will simplify subsequent discussion.

Oriented bus: A linear acyclic bus is *oriented* iff each processor on the bus can determine which of its ports is closer to (say) the left end of the bus.

Lemma 7.1 *Every linear acyclic bus of an $X \times Y$ LR-Mesh can be oriented on an $2X \times 2Y$ LR-Mesh.*

Proof: Every linear acyclic bus has two end points. Call any one of these ends the left end and the other the right end. The LR-Mesh replaces each bus by two "oriented buses" as shown in Figure 7.2 [17] and assigns different orientations to each bus. ■

An important operation on an oriented linear bus is "neighbor localization." Given an oriented linear bus with each processor p_i on it flagged by a Boolean variable f_i, neighbor localization constructs a linked list of processors p_i with $f_i = 1$ in the order in which flagged processors are placed on the bus. If the linear bus is oriented, then neighbor localization can be solved in constant time on the bus [17].

Write rules: If a constant blowup in size is permissible, then all concurrent writes (except with PRIORITY) on an LR-Mesh can be replaced by exclusive writes.

Lemma 7.2 *Every step of an $R \times C$ LR-Mesh with only acyclic buses and which does not use the* PRIORITY *rule can be emulated on a $2R \times 2C$ CREW LR-Mesh in constant time.*

Proof: The LR-Mesh replaces each bus by two "directional buses" as shown in Figure 7.2. Each bus is now oriented. Then flag each writer on the bus and apply the neighbor localization algorithm to determine the "leftmost" writer on the bus. For the ARBITRARY and COMMON rules, this leftmost processor performs the exclusive write. For COLLISION rule, if there is only one writer (with no left and right neighbors), then it writes its value to the bus. If not, then the leftmost writer writes the collision symbol to the bus. For the COLLISION$^+$ rule, if there is only one writer, then it writes its value to the bus. If there is more than one writer, then each writer sends data to the next writer to its left (say). Each writer that detects a value different from its own flags itself with a 1. An OR operation is done on all writers (this again amounts to neighbor localization). If the result of the OR is 0, then all writers have the same value and the leftmost writer is chosen to write its value to the whole bus. If the OR is 1, then there are multiple writers with different data and the leftmost writer writes the collision symbol to the entire bus. ■

Lemma 7.3 *An LR-Mesh using exclusive writes or concurrent writes under the* COMMON, COLLISION, *or* COLLISION$^+$ *rules can assume, without loss of generality, that all buses are acyclic.*

Proof: Cut the bus at each writer. This transforms every bus (cyclic or not) into pieces each of which is linear. If the writes are exclusive, then the writer simply writes to the two segments of the bus (in two separate steps). Otherwise, each piece has exactly two writers, one at each end of that piece. (If a bus is cyclic, but with no writer on the bus at a certain step, then that bus can be ignored at that step.)

If the bus has writer(s), then for the COMMON rule, each writer cuts the bus and writes to both pieces on which it is incident. Clearly, all processors get the same value. For the COLLISION rule, each writer cuts the bus and writes to both pieces of bus it is incident. If there is only one writer, then all processors on the bus gets the value written. If there is more than one writer, then each writer receives a collision symbol (except the piece of the bus of the leftmost processor and the piece of the bus of the rightmost processor). In another step the collision symbol is written to both

of these pieces. For the COLLISION$^+$ rule, each writer cuts the bus and both ends of each piece exchange data. Then, each writer that detects different data than its own cuts the bus and writes a collision symbol. If there are no such writers, then all writers conclude that all the data written are the same and all original writers write this data to all other processors. ■

Semimonotonic Configurations: A semimonotonic bus could be row or column monotonic. So a configuration composed of a set of semimonotonic buses could have both row and column monotonic buses. A semimonotonic configuration is only permitted to have either row monotonic or column monotonic buses. In the following lemma, we prove that a set of semimonotonic buses can be treated as a semimonotonic configuration.

Remarks: The CREW LR-Mesh buses have the same properties (such as semimonotonicity etc.) as the CRCW counterparts.

Before we proceed, we establish a preliminary result that may be of independent interest.

Define the *gossiping problem on set* S as follows. Let each port i of each processor hold a value $v_i \in S \cup \mathtt{n}null$; the null indicates that a port may not hold a value. The problem is for each port incident on a bus b to determine the set $\{v_i : i \text{ is incident on } b\}$.

Lemma 7.4 *A* COMMON *or* COLLISION *CRCW LR-Mesh can solve the gossiping problem on set* S *in* $|S|$ *steps.*

Proof: Without loss of generality, let $S = \{1, 2, \cdots, \alpha\}$. Iterate α times as follows. In iteration j (where $1 \leq j \leq \alpha$), each port i holding value $v_i = j$ writes a signal to its bus and all ports incident on the bus read. A port receiving the signal in iteration j concludes that some port on its bus holds value j. For the COMMON rule, each port writes j in iteration j. For COLLISION, a signal of any value will be read by all ports incident on the bus. ■

Lemma 7.5 *Let $\mathcal{L}$ be a* COMMON *or* COLLISION *CRCW LR-Mesh. If every bus of a configuration of $\mathcal{L}$ is given to be semimonotonic, then $\mathcal{L}$ can partition the configuration into two semimonotonic configurations in $O(1)$ time.*

Proof: Let $\mathcal{C}$ be the given configuration. The aim is to create two configurations $\mathcal{C}_r$ and $\mathcal{C}_c$ so that every bus of $\mathcal{C}_r$ (resp., $\mathcal{C}_c$) is row (resp., column) monotonic, and every bus of $\mathcal{C}$ in either $\mathcal{C}_r$ or $\mathcal{C}_c$. We explain the algorithm for one bus b. All buses follow on the same line.

Each processor p in which a bus b bends cuts the bus within that processor. Let x and y be the two ports of a processor through which bus b traversed before it was cut. Assign values v_x to x and v_y to y as follows.

$$v_x = \begin{cases} 1, & \text{if } y \in North \\ 2, & \text{if } y \in South \\ 3, & \text{if } y \in East \\ 4, & \text{if } y \in West \end{cases} \qquad v_y = \begin{cases} 1, & \text{if } x \in North \\ 2, & \text{if } x \in South \\ 3, & \text{if } x \in East \\ 4, & \text{if } x \in West \end{cases}$$

Also assign value 5 to each processor holding an end of bus b; since b is semimonotonic, it must be acyclic. All remaining ports hold value `null`. Now solve the gossiping problem on set $\{1, 2, 3, 4, 5\}$ for each segment of bus b. This require three iterations; i.e., constant time. At this point, ports of each processor in which a bus bends know the value held by ports at neighboring bends.

A processor in which bus b bends could determine the status (row monotonic or column monotonic) of bus b as follows. Let x and w be two ports at the ends of a segment of bus b. Let Q be the set of value(s) obtained by port x after solving the gossiping problem on set $\{1, 2, 3, 4, 5\}$. Clearly, $1 \leq |Q| \leq 2$.

Case 1 $|Q| = 1$. Here $Q = \{v_x\}$. Therefore the other end of the bus segment starting at x must also have value v_x. If $v_x = 1$ or $v_x = 2$, then we have the situation in Figure 7.3 Port x determines that bus b is column monotonic. If $v_w = 3$ or $v_w = 4$, then the situation in Figure 7.4 ensures that bus b is row monotonic.

Figure 7.3: Detection of a column monotonic bus

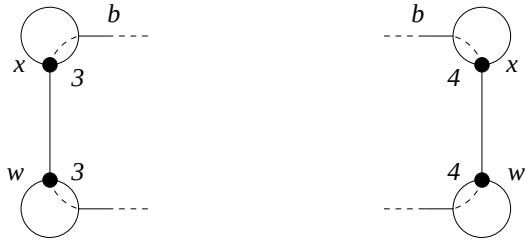


Figure 7.4: Detection of a row monotonic bus

Case 2 $|Q| = 2$. Here $Q = \{v_x, v_w\}$. If $v_w \neq 5$, then we have the situation in Figure 7.5, and port x cannot tell whether bus b is column monotonic or row monotonic.

If $v_w = 5$ (see Figure 7.6), then again x cannot determine definitely whether bus b is column monotonic or row monotonic.

At this point, each port has either determined its bus to be row monotonic (R), column monotonic (C) or undecided (`null`). Processors of $\mathcal{L}$ reconnect their ports to reconstruct bus b. Solve the gossiping problem again on set $\{R, C\}$ for the entire bus b. Since every bus is either row or column monotonic, the result of the gossiping is a set with at most one element. If $S = \{R\}$, then the bus is row monotonic. If $S = \{C\}$, then the bus is column monotonic. If $S = \emptyset$, then the the bus qualifies to be both row monotonic and column monotonic. Assign it to be either row or column monotonic. ■

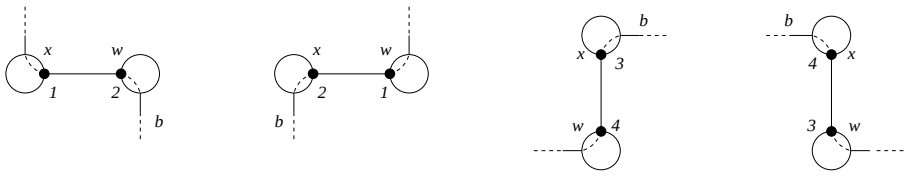


Figure 7.5: Illustration of the case $v_w \neq 5$

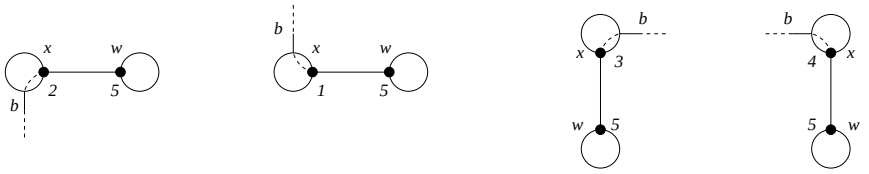


Figure 7.6: Illustration of the case $v_w = 5$

Scalability: In general an LR-Mesh has an optimal scaling simulation [3]. That is, for any $R' < R$ and $C' < C$, an $R \times C$ LR-Mesh can be simulated in $O\left(\frac{RC}{R'C'}\right)$ steps on an $R' \times C'$ LR-Mesh. This allows many LR-Mesh algorithms that use $\Theta(P)$ processors to state results for P processors. However, the scaling simulation algorithm used for this result, destroys the structural characteristics of the simulated LR-Mesh buses (semimonotonicity etc.). Therefore we cannot use the LR-Mesh scalability freely in our results, which rely on buses having these structural properties. Figure 7.7 illustrates the situation and shows why constants c_1 and c_2 cannot be removed in general.

If the given LR-Mesh algorithm uses monotonic buses (both row and column monotonic) then a different scaling simulation due to Murshed [33] can be used. This simulation preserves the buses' monotonicity. As shown in Figure 7.8, constants can now be removed.

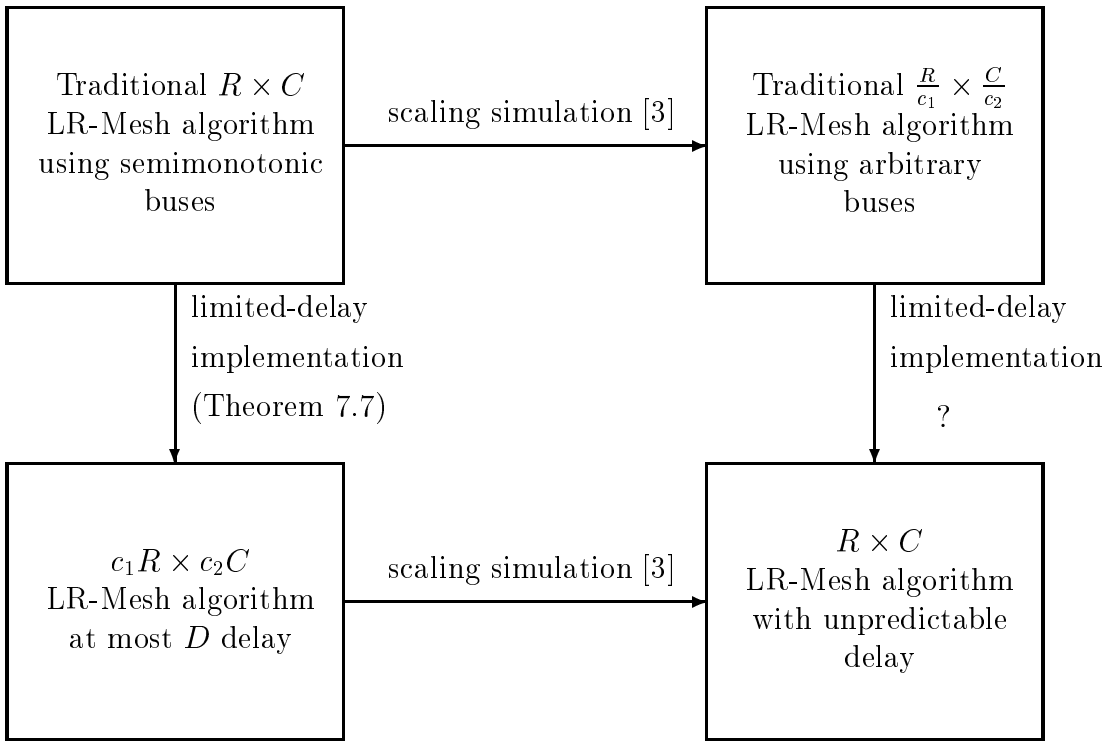


Figure 7.7: General scaling simulation do not work for semimonotonic buses

7.2 The Bends-Cost Measure

As mentioned in Section 2.5, a reconfigurable bus is a combinational circuit that establishes a data path from each potential writer to all processors connected to the bus. Because there are relatively few taps between two successive gates (switches used to configure the bus), capacitive loading [44] is not a predominant factor. The primary concern is the gate delay of the longest path of this circuit. Thus, the bus delay could be considered to be the gate delay in the longest path traversed by data in a bus. In Section 2.5 we described several measures for the bus delay. In this section we introduce a new measure for linear buses called bends-cost.

Bends-Cost: Under bends-cost, the delay of a bus is assumed to be roughly proportional to the number of "bends" in the bus. (Each transition of the bus from a row of the LR-Mesh to a column, or vice versa, is called a bend.) Specifically, if a

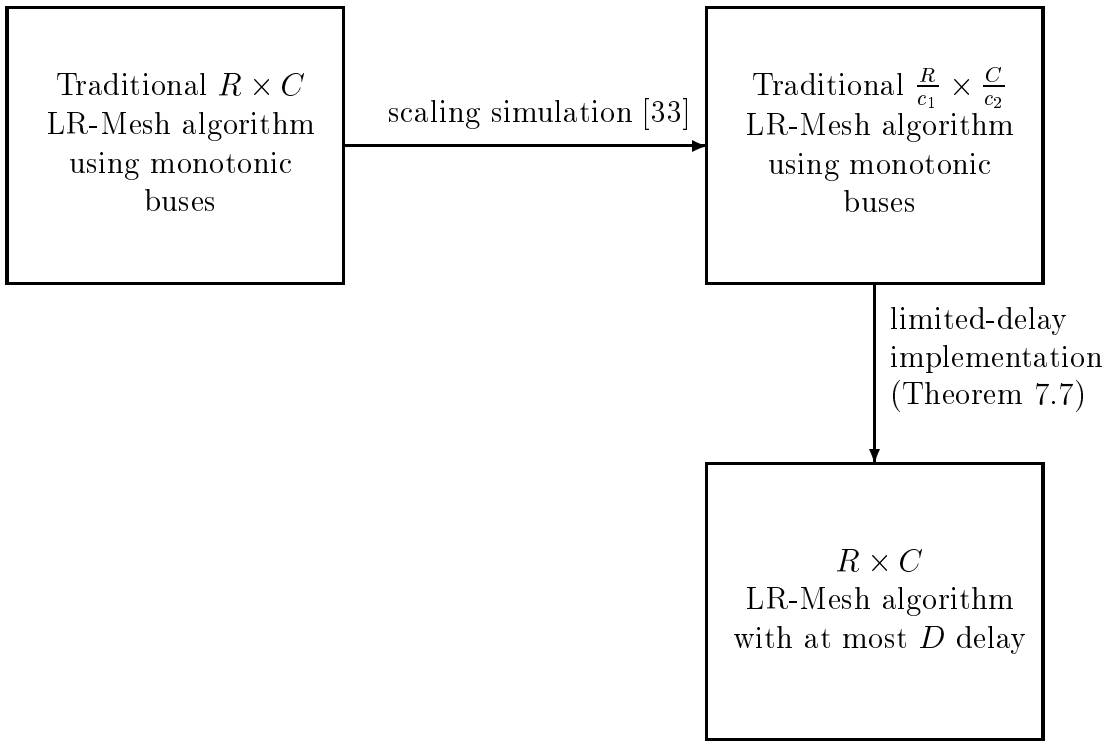


Figure 7.8: Using a restricted scaling simulation for semimonotonic buses

linear bus snakes through r rows and c columns of the LR-Mesh, then its bends-cost delay is $r + c$. Consider the $N \times N$ LR-Mesh (for $N = 7$) of Figure 7.9. Buses labeled A and B have the same end points and both span $O(N)$ processors. However, bus A has one bend and $O(1)$ bends-cost delay, while bus B has $\Theta(N)$ bends and therefore $\Theta(N)$ delay. Bus C also has $\Theta(N)$ delay, even though it alternates between the same two rows of the LR-Mesh.

Lemma 7.6 *For any $x \leq N$, each bus of any column monotonic configuration of an $N \times x$ LR-Mesh has at most $2x - 2$ bends.*

Proof: A bus originating at the leftmost column can have at most $2x - 2$ bends before it reaches the rightmost column. Since it is column monotonic, it cannot turn back to a previously traversed column. ■

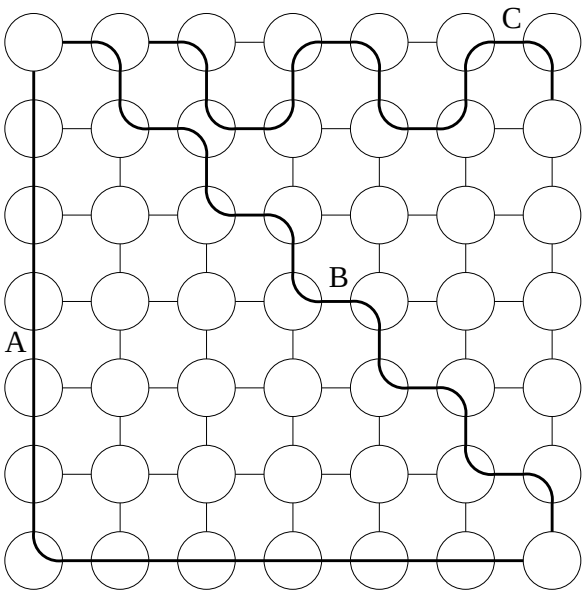


Figure 7.9: Buses with different numbers of bends for an $N \times N$ LR-Mesh ($N = 7$)

7.3 A Bends-Cost LR-Mesh Implementation

In this section, we outline an LR-Mesh implementation for which the bends-cost measure is an accurate indicator of the actual bus delay. A segmentable bus is an important building block of this implementation. Recall that a segmentable bus (see Figure 2.4) consists of processors connected to a bus with each processor p capable of controlling a switch that can segment the bus between p and the previous processor. The segmentable bus is similar in function to a 1-dimensional R-Mesh (see Section 2.2). Let $S(N)$ denote an N-processor segmentable bus whose (gate) delay is at most Δ. By Theorem 6.4 (page 115) and the fact that each step on a CST with N leaves has $O(\log N)$ gate delay, $\Delta = O(\log N)$. We note, however, that the results of this chapter are not premised on any particular segmentable bus implementation and are general enough to accommodate future improvements in Δ.

Construct an $N \times N$ bends-cost LR-Mesh as follows. Arrange processors as an $N \times N$ array and connect each row and each column of processors by a segmentable bus $S(N)$ (see Figure 7.10(a)). At this point, this structure is an implementation of

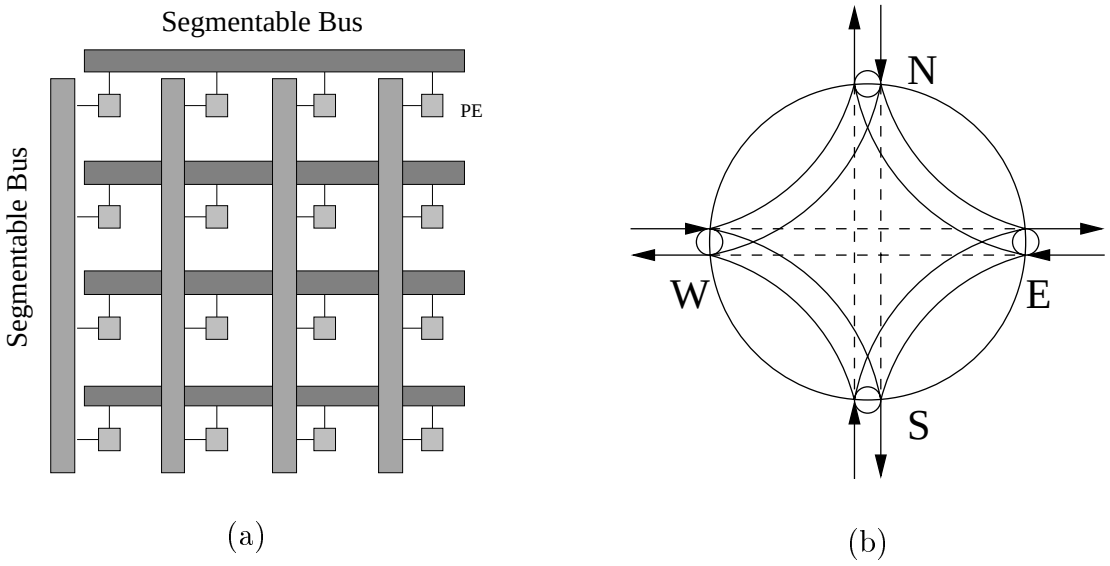


Figure 7.10: Structure of a bends-cost LR-Mesh implementation

a special case of the LR-Mesh called the HVR-Mesh [3] (or Basic R-Mesh [6]) that restricts all its buses to be without bends. Within each processor, additional switches allow a bus segment to bend from a row to a column or vice versa (see Figure 7.10(b)). The connections between N, S or E, W ports is through the segmentable bus; Figure 7.10(b) shows these connections dashed. The switching fabric in each processor has the form shown in Figure 7.11 requiring only four 2-input multiplexers one for

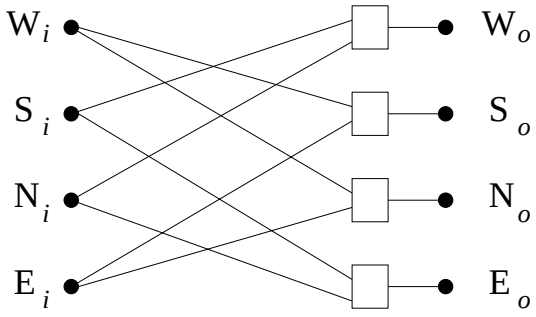


Figure 7.11: Switching fabric of a bends-cost LR-Mesh processor

each port. Thus, this additional switching fabric along with the CST implementation of a segmentable bus could realize the entire bus as a combinational circuit.

Thus a bus with β bends consists of $\beta+1$ row and column bus segments connected in tandem, each with at most Δ delay. Consequently, the actual delay of the bus is $\Delta(\beta+1)$ which is proportional to the quantity $\beta+1$ predicted by the bends-cost measure.

Thus, we can now state results for the bends-cost LR-Mesh using buses with β bends or an LR-Mesh result using buses of $D=\beta\Delta$ delay.

7.4 Designing Implementable LR-Mesh Algorithms

In the last section we showed that the bends-cost measure of bus delay can be an accurate model of the actual delay. Therefore one way to design an implementable LR-Mesh algorithm is to ensure that the algorithm uses buses with limited numbers of bends. One way to approach this task is to redesign the LR-Mesh algorithms. The other approach is to design an automatic method to convert large classes of the LR-Mesh algorithms (configurations) to run with bounded delay. We adopt the latter approach as it offers the possibility of harnessing the large body of results for the LR-Mesh.

In the next section we begin our discussion with semimonotonic configurations. Many fundamental LR-Mesh algorithms (including counting, prefix sums, multiple addition, and sorting) use semimonotonic configurations. We show that every semimonotonic configuration can be emulated quickly and efficiently on buses of bounded delay (number of bends). Subsequently in Section 7.6, we consider more general configurations.

In these sections we will consider LR-Meshes that use either the unit-cost or the bends-cost measure of bus delay. In addition to the number of steps used, we will characterize a bends-cost LR-Mesh algorithm by the maximum delay D (or the maximum number of bends) that any bus in any configuration of the algorithm may have.

7.5 Simulating Semimonotonic Configurations

Without loss of generality, let the configuration be column monotonic (see Lemma 7.5). Let $\mathcal{U}$ be a column monotonic configuration of an $N \times N$ unit-cost LR-Mesh. We use the symbol $\mathcal{U}$ to denote the above configuration and the unit-cost LR-Mesh as well. Let $\mathcal{B}$ be a $c'N \times c''N$ bends-cost LR-Mesh, where c' and c'' are constants whose values will become apparent later. One could view each processor of $\mathcal{U}$ as corresponding to a unique $c' \times c''$ "cluster" of processors of $\mathcal{B}$. Clearly, each sub-LR-Mesh of $\mathcal{U}$ also corresponds to a sub-LR-Mesh of $\mathcal{B}$. Let $D \geq \Delta$ denote the maximum delay that a bus of $\mathcal{B}$ can incur. The main result of this section is the following theorem.

Theorem 7.7 *Let Δ be the delay of an N-processor segmentable bus. For any $D \geq \Delta$, any semimonotonic configuration of an $N \times N$ unit-cost LR-Mesh can be simulated in $O\left(\left(\frac{\log N}{\log D - \log \Delta}\right)^2\right)$ time on a $\Theta(N) \times \Theta(N)$ bends-cost LR-Mesh using buses with at most D delay.* ■

The following corollary reflects an interesting special case of this result.

Corollary 7.8 *For any $\epsilon > 0$, any semimonotonic configuration of an $N \times N$ unit-cost LR-Mesh can be simulated in $O(1)$ time on a $\Theta(N) \times \Theta(N)$ bends-cost LR-Mesh using buses with at most N^ϵ delay.* ■

Remark: This is the first general method to achieve constant time on a reconfigurable model without resorting to the unit-cost measure of bus delay.

Most of the remainder of this section is devoted to establishing Theorem 7.7. We organize this section into three subsections. In Section 7.5.1, we reduce the simulation to a "channel assignment" problem in an LR-Mesh. Section 7.5.3 first solves a restricted version of this channel assignment problem and develops results that find use later. Section 7.5.4 uses the results of Section 7.5.3 to solve the (general) channel assignment problem and completes the simulation.

7.5.1 Simulation Algorithm

The purpose of the simulation is as follows. Suppose that writes during a step with configuration $\mathcal{U}$ result in value v_p on some port p of the slice. Then on the corre-

sponding port p' of $\mathcal{B}$, the same value v_p must appear at the end of the simulation. Moreover, $\mathcal{B}$ can only employ buses with $O(D)$ delay.

Let $x = \frac{cD}{\Delta}$, where $c < 1$ is a constant, Δ is the delay of an N-element segmentable bus, and $D \geq \Delta$ is the maximum allowed delay of buses in $\mathcal{B}$. Without loss of generality, let $N = x^\alpha$, for integer $\alpha \geq 1$. At this point we note that the delay of buses of $\mathcal{B}$ will be proportional to the quantity x. Since we can select the constant c in the definition of x without constraint, we may assume that $\mathcal{B}$ permits $O(D)$ delay, rather than "at most D delay," as required.

For $1 \leq k \leq \alpha = \log_x N$, we use the following recursive algorithm to simulate an $N \times x^k$ sub-LR-Mesh $\mathcal{U}_k$ of $\mathcal{U}$ on the corresponding sub-LR-Mesh $\mathcal{B}_k$ of $\mathcal{B}$. Note that since $\mathcal{U}$ has a column monotonic configuration, so does $\mathcal{U}_k$.

1. If $k = 1$, then we have an $N \times x$ LR-Mesh $\mathcal{U}_1$. By Lemma 7.6, every bus of $\mathcal{U}_1$ has at most $O\left(\frac{D}{\Delta}\right)$ bends and, therefore, $O(D)$ delay. Consequently, $\mathcal{B}_1$ can simulate $\mathcal{U}_1$ by using the exact same bus configuration (without incurring excessive bus delay).

2. If $k > 1$, then divide $\mathcal{U}_k$ into x *slices*, each consisting of x^{k-1} contiguous columns; that is, each slice is an $N \times x^{k-1}$ sub-LR-Mesh of $\mathcal{U}_k$ (see Figure 7.12). Moreover, each slice has a column monotonic configuration. Similarly, divide $\mathcal{B}_k$ into corresponding slices.

   We will refer to the W ports of processors on the leftmost column of a slice collectively as the *left border* of the slice. Similarly define the *right*, *top*, and *bottom* borders of the slice (see Figure 7.13(a)). Adjacent slices touch only at their left and right borders.

3. Recursively simulate (in parallel) the slices of $\mathcal{U}_k$ on the corresponding slices of $\mathcal{B}_k$. Now, for each bus b of each slice S of $\mathcal{U}_k$ the following statements hold.

   - All processors of b hold the value, if any, written to b from within the slice.
   - All processor of b hold the end points of the bus.

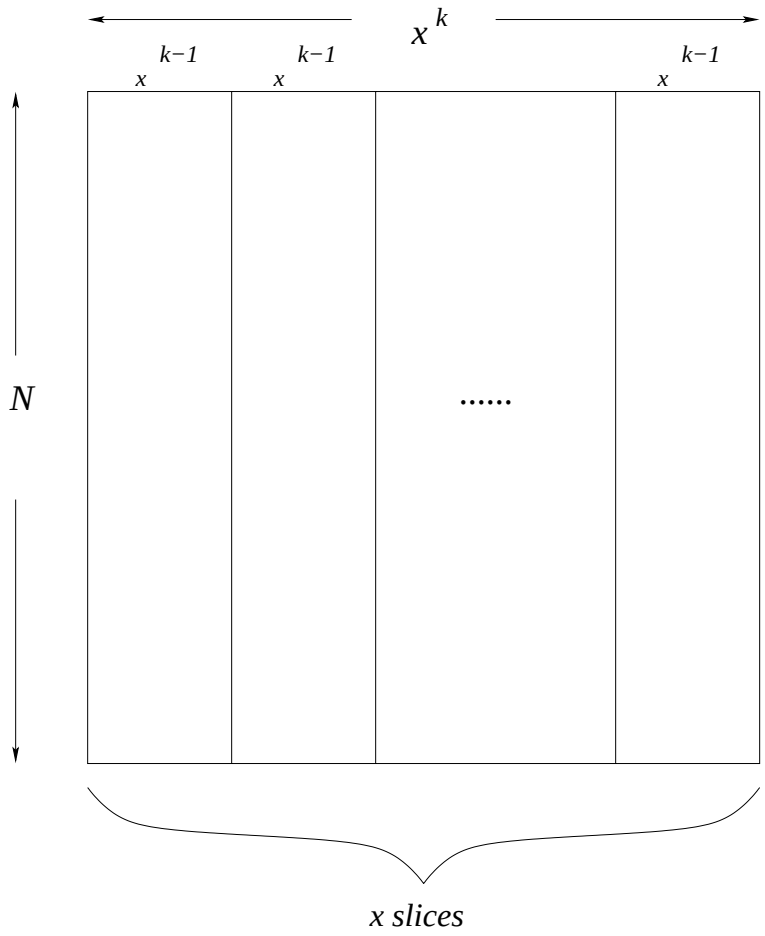


Figure 7.12: Dividing a slice into x slices

The remaining phases serve to propagate bus values among slices. Once a slice has received values (if any) that come from outside it, it is easy to reverse the steps of the recursion to propagate these new values within the slice. So we focus only on propagating values among slices.

4. Consider the following classification of buses of slice S of $\mathcal{U}_k$ or the corresponding slice S' of $\mathcal{B}_k$.

   - Category 0: Neither end point of bus b touches the left or right border of S (see buses marked C–F in Figure 7.13).

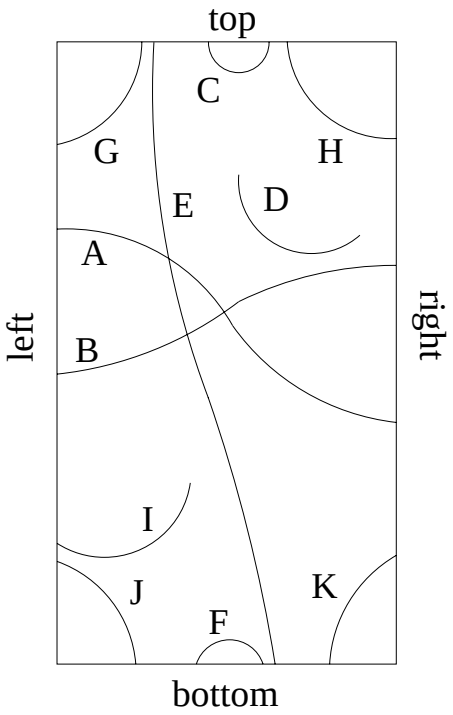


Figure 7.13: Bus types

- Category 1: One end point of b touches the left or right border of S (see buses G–K of Figure 7.13).
- Category 2: The end points of b are on the left and right borders of S (see buses of Type A and B in Figure 7.13).

Note that for a column monotonic bus, both end points cannot be on the left border (or both on the right border) of $\mathcal{B}_k$. In this phase, $\mathcal{B}_k$ identifies the category of each bus b as follows. Let the end points of b be r and s. Clearly, both cannot be on the left border or on the right border. If r or s is on the left border, then $\mathcal{B}_k$ writes the index of the port on the bus. Next, if r or s is on the right border, then $\mathcal{B}_k$ writes the port index on the bus.

It is easy to verify that bus b is in Category i ($0 \leq i \leq 2$) iff it receives i values in the above steps. In addition, a Category 2 bus can identify its Type (A or B) as well by ascertaining whether its left end is higher than the right (for Type A) or not (for Type B).

5. In this phase, $\mathcal{B}_k$ performs actions for a bus b depending on its category.

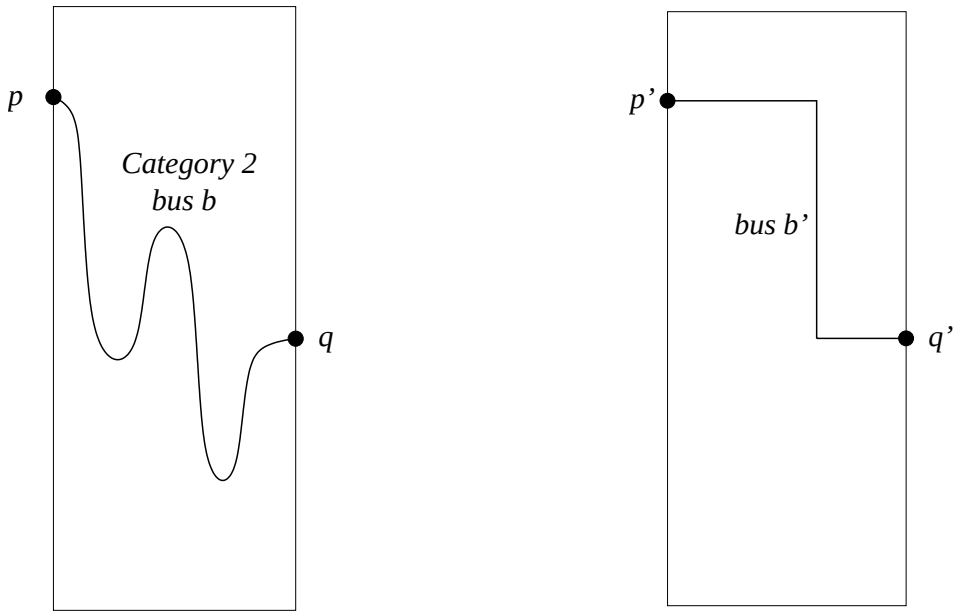
   - Category 0: Here b has no effect on other slices, nor is it affected by other slices. Therefore b does not participate further in the simulation.
   - Category 1: Let bus b touch another bus b_0 of an adjacent slice that traverses a port on the left or right border of this adjacent slice. Let this port of the adjacent slice be in processor p_0. If the recursive step computes the value of bus b to be v (that is, there is a write from within slice S), then the end of bus b touching bus b_0 sends v to processor p_0 and bus b does not participate further in the simulation. If slice S does not generate a value for bus b, then it waits to hear from processor p_0 about its final bus value.
   - Category 2: Let p and q be the ports at the left and right end points of bus b. Construct a column monotonic bus b' in the corresponding slice S' of $\mathcal{B}_k$ so that (i) the end points (clusters) p' and q' of b' correspond to the end points p and q of b, and (ii) bus b' has a constant number of bends (see Figure 7.14).

   The action for Category 2 buses is undertaken after Category 1 buses have been handled. Thus, the entire slice $\mathcal{B}_k$ is available for Category 2.

6. Note that each bus in each of the x slices of $\mathcal{U}_k$ is either removed or replaced by a column monotonic bus with a constant number of bends. Thus, the configuration of the bends-cost LR-Mesh $\mathcal{B}_k$ is column monotonic, and each bus has $O(x)$ bends, or $O(D)$ delay. Therefore, $\mathcal{B}_k$ can use its buses without incurring excessive delay and convey values between slices.

Let $T(k)$ denote the time to perform the above simulation on an $N \times x^k$ LR-Mesh. Let t_{k-1} be the time to handle Category 2 buses for an $N \times x^{k-1}$ slice. Then, $T(1)$ is constant and for all $k > 1$, $T(k) = T(k-1) + t_{k-1} + \text{constant}$. Solving this recurrence, we have,

$$T(k) = O\left(\sum_{q=1}^{k-1} t_q\right). \tag{7.1}$$

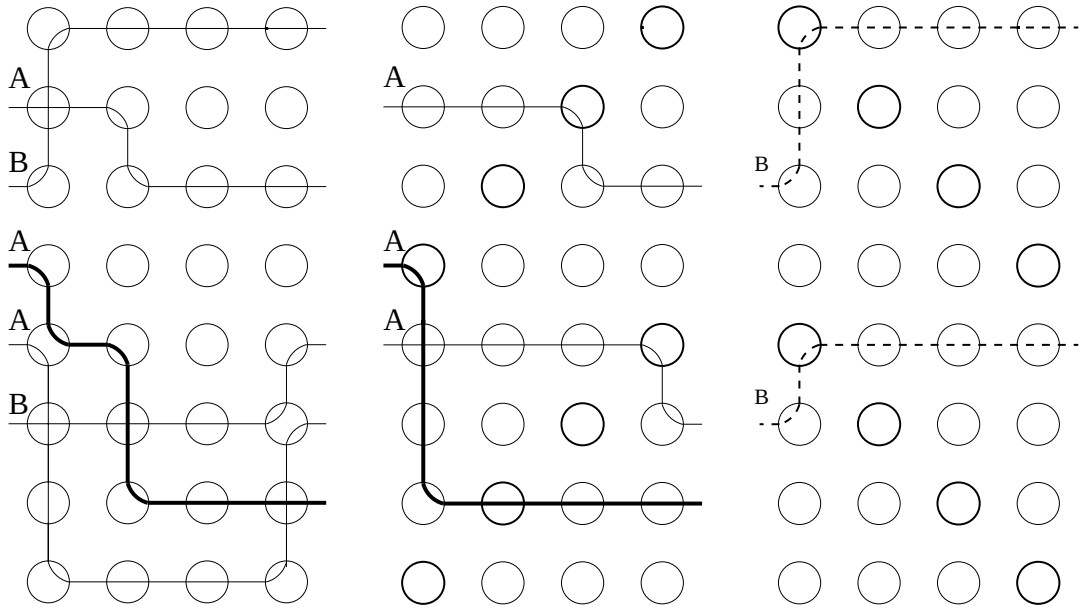


(a) (b)

Figure 7.14: Handling Category 2 buses

Handling Category 2 Buses: The only step of this algorithm requiring further elaboration is the handling of Category 2 buses. Figure 7.15(a) shows representative buses of Types A and B of Category 2. Both these types of buses have one end point on the left border and one end on the right border. The only difference is that for a Type A bus the left end point is higher than the right, and for a Type B bus, the left end point is lower.

Our solution for handling Category 2 buses of Type A will use a $c_1N \times c_2x^{k-1}$ sub-LR-Mesh of $\mathcal{B}_k$ to handle a slice of $\mathcal{U}_k$. Clearly, Type B buses can also be handled similarly; that is, all Type B buses can also be laid out on a $c_1N \times c_2x^{k-1}$ sub-LR-Mesh of $\mathcal{B}_k$. We now describe how both types can be handled simultaneously on $\mathcal{B}_k$. Call the sub-LR-Mesh handling Type A (resp., Type B) buses as Tier A (resp., Tier B). The two tiers are interleaved into a $2c_1N \times (2c_2x^{k-1} + 2)$ sub-LR-Mesh so that their buses can be laid out without interfering with each other. Figure 7.15(a) shows a set of Type A and B buses. Figures 7.15(b) and (c) show these routed in separate tiers. Figure 7.16 shows the tiers combined.



(a) (b) (c)

Figure 7.15: Routing Type A and B buses in different tiers

The basic idea is as follows. Divide the $2c_1N \times (2c_2x^{k-1}+2)$ sub LR-Mesh $\mathcal{L}$ (say) into three vertical strips. The first and third consist of the first and last columns of the sub-LR-Mesh (shown shaded in Figure 7.16). The middle strip is a $2c_1N \times (2c_2x^{k-1})$ LR-Mesh $\mathcal{L}'$ (say). The layout of Types A and B buses in $\mathcal{L}$ use the following rules. For Type A buses, all vertical (resp., horizontal) segments occupy only even columns (resp., rows) of $\mathcal{L}'$ (number rows and columns 0,1,$\cdots$). For Type B buses, on the other hand, all vertical (resp., horizontal) segments, occupy only odd columns (resp., rows). Thus the only way a Type A and a Type B buses traverse the same processor is in different directions (one horizontal and the other vertical). Consequently, a Type A bus will never get in the way of a Type B bus, and vice versa.

We now explain the function of the first and the last strips (columns in Figure 7.16) of $\mathcal{L}$. In the method explained above, Type A (resp., Type B) buses exit the left and right borders of $\mathcal{L}'$ at even (resp., odd rows). However, a Type A bus on one slice may be a Type B bus of the next. That is, Type A and Type B buses are not known *a priori*. The additional realignment columns (shown dashed in Figure 7.16) serve to

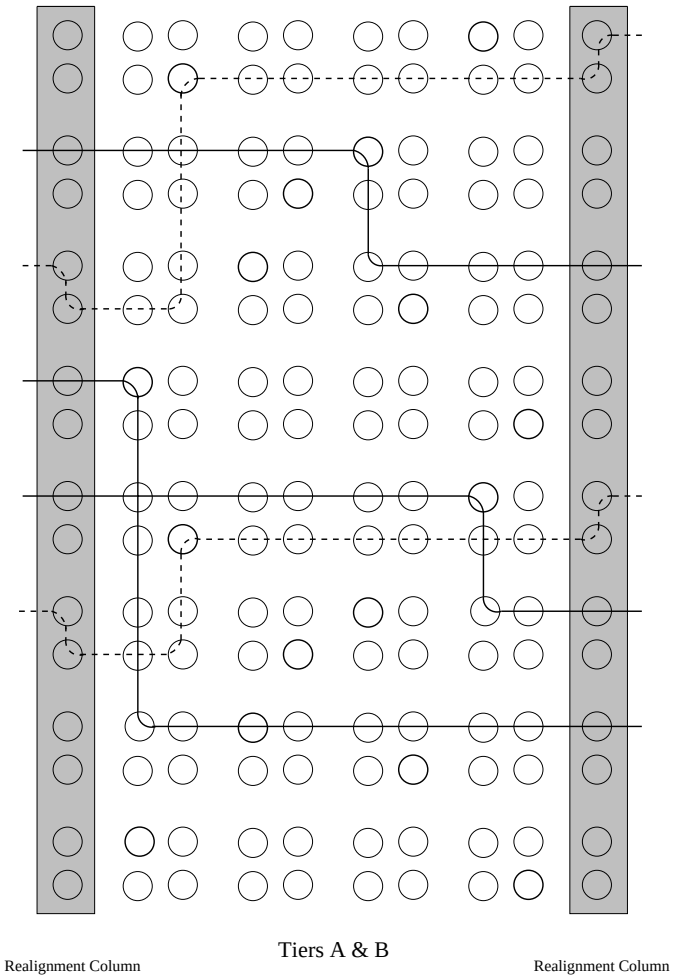


Figure 7.16: Combining two tiers

position buses on the same rows of adjacent slices (regardless of the bus type). Note that since a processor on the left or right border of $\mathcal{U}_k$ can have a Type A or a Type B bus (but not both), and that this type is known *a priori*, the realignment columns can be configured appropriately.

Thus with $c' = 2c_1$ and $c'' < 3c_2$, the size of the simulating bends-cost LR-Mesh $\mathcal{B}$ is $c'N \times c''N$.

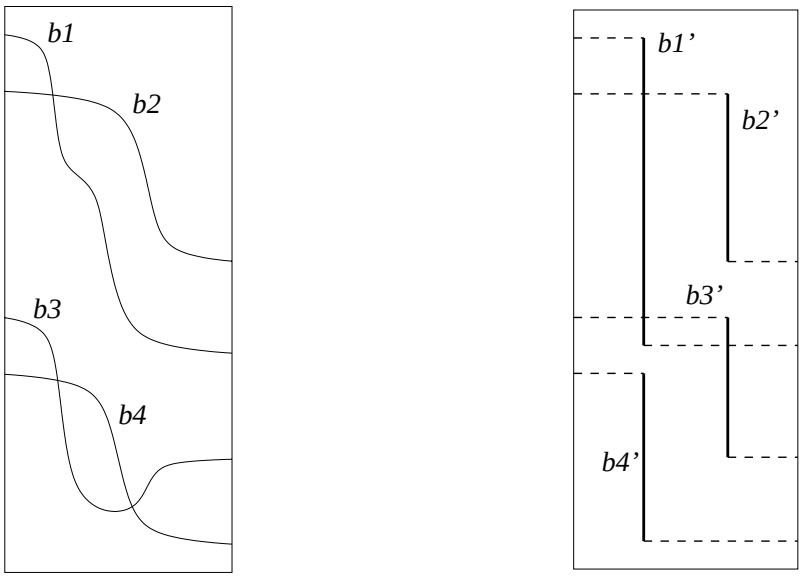
This idea of using two tiers to accommodate two classes of buses can be extended to multiple tiers; m tiers of an $R \times C$ LR-Mesh can be accommodated on an $mR \times (mC + 2)$ LR-Mesh.

Since subsequent discussion is for Type A buses (Type B being handled analogously), we will refer to rows, columns, and processors of $\mathcal{B}_k$ rather than pairs of rows, columns, and clusters, to mean rows, columns, and processors of $\mathcal{B}_k$ that handle Type A buses.

7.5.2 The Channel Assignment Problem

All that remains now is the handling of Type A buses on a $c_1N \times c_2x^{k-1}$ LR-Mesh. Specifically, let the end points of bus b be at row p of the left border and at row q of the right border of a slice of $\mathcal{U}_k$, where $p \leq q$. The aim is to construct a column monotonic bus b' in $\mathcal{B}_k$ starting from the left border at row p' and ending at the right border at row q'. The constructed bus b' will have three segments (two bends or a delay of 3Δ): the first segment is horizontal and runs in row p' from the left border to some column m; the third segment runs in row q' from column m to the right border of the slice. The middle segment is a vertical bus on column m between rows p' and q'. For example, for the Type A bus in bold in Figure 7.15(a), the algorithm constructs the three segment bus shown in bold in Figure 7.15(b). Since no other bus of the given slice of $\mathcal{U}_k$ can have p and q as end points, it is straightforward to construct the first and third segments of bus b'. The challenge lies in selecting an appropriate column m for each bus so that no two buses overlap, yet all buses are accommodated within x^{k-1} columns.

The task now reduces to the following *channel assignment problem*. Denote by $\{b_0, b_1, \cdots, b_{y-1}\}$, a set of Type A buses of an $N \times X$ LR-Mesh (slice of $\mathcal{U}_k$), where $X = x^{k-1}$. Each of these buses has a delay of at most D. For each bus b_i, let s_i and e_i denote the row numbers where the bus touches the left and right borders of the slice; we will call these rows the *starting* and *ending* rows, respectively, of bus b_i. Since b_i is a Type A bus, $s_i \leq e_i$; recall that rows are numbered in increasing order from the top to the bottom of the R-Mesh. The task is to use a $c_1N \times c_2X$ bends-cost LR-Mesh to assign a column number m_i, where $0 \leq m_i < X$, to each bus b_i such that if $m_i = m_j$ (for $i \neq j$), then either $s_i \geq e_j$ or $e_i \leq s_j$. For example, if the input buses are shown in Figure 7.17(a), then the buses could be assigned columns as shown in Figure 7.17(b).



(a) (b)

Figure 7.17: Assignments of columns to buses

In the next section we solve a restricted case of the channel assignment problem and develop some tools used subsequently to solve the general (unrestricted) channel assignment problem. In these sections we will often treat the simulating bends-cost LR-Mesh, $\mathcal{V}$ (say), as an $N \times X$ LR-Mesh so that its rows and columns are in one-to-one correspondence with the simulated $N \times X$ unit-cost LR-Mesh, $\mathcal{S}$ (say). The details necessary to tailor the discussion to the actual size of the simulating LR-Mesh are tedious and will provide no additional insight. To distinguish the input buses $b_0, b_1, \cdots b_{y-1}$ of S from buses of $\mathcal{V}$ used to solve the channel assignment problem, we will refer to the y input buses as "BUSES." That is, "buses" are physical buses configured by $\mathcal{V}$ during the simulation of $\mathcal{S}$, whereas "BUSES" are simply inputs to this simulation.

7.5.3 Restricted Channel Assignment

In this section we solve a simple case of the channel assignment problem where $e_i - s_i \leq X$, for each BUS b_i (that is, each BUS has at most X rows between its starting and

ending rows). The solution is quite straightforward. Simply assign $m_i = s_i (\mathrm{mod}\ X)$ to bus b_i. BUS b_i occupies column m_i from the S port of the processor in row s_i to the N port of the processor in row e_i. The next BUS b_j that can use column m_i has $s_j = s_i + X \geq e_i$. Thus the restricted channel assignment problem can be solved in constant time.

Lemma 7.9 *An instance of the channel assignment problem where $e_i - s_i \leq X$ for each $0 \leq i < y$, can be solved in $\Theta(1)$ time.* ■

Remarks: For only Type A buses an $N \times X$ simulating LR-Mesh suffices for the above result (i.e., $c_1 = c_2 = 1$). If both Types A and B are possible in the configuration, then $c_1 = c_2 = 2$.

Recall the definition of an incremental configuration in Section 2.4. By Lemma 7.5, there is no loss of generality in assuming that a given incremental configuration is column monotonic. In such a configuration a bus cannot cover more than X rows while traversing through X columns. Clearly, the simulation of such a configuration will require only the solution to the restricted channel assignment problem (Lemma 7.9). From this observation and Equation (7.1), we have the following result (here, too, $c_1 = c_2 = 2$).

Theorem 7.10 *Let Δ denote the delay of an N-element segmentable bus. For any $D \geq \Delta$, any incremental configuration of an $N \times N$ unit-cost LR-Mesh can be simulated in $O\left(\frac{\log N}{\log D - \log \Delta}\right)$ time on a $\Theta(N) \times \Theta(N)$ bends-cost LR-Mesh whose buses have at most D delay.*

Proof: By Lemma 7.9, $t_q = O(1)$ for $1 \leq q < k$. Therefore, by Equation 7.1, $T(k) = O(k)$. Since $x^\alpha = N$, our simulation simulates an $N \times x^\alpha$ LR-Mesh. So the total time is $T(\alpha) = O(\alpha) = O\left(\frac{\log N}{\log x}\right) = O\left(\frac{\log N}{\log \frac{D}{\Delta}}\right) = O\left(\frac{\log N}{\log D - \log \Delta}\right)$. ■

Remark: Note that even if a configuration is not incremental, it may be possible to simulate it using the restricted channel assignment problem.

In the next few subsections, we show that counting N bits and sorting N numbers can be performed on a $\Theta(N) \times \Theta(N)$ bends-cost LR-Mesh whose buses have at most D delay in $O\left(\frac{\log N}{\log D - \log \Delta}\right)$ time. For adding N b-bit numbers, the time is the same, but

a $\Theta(N) \times \Theta(Nb)$ bends-cost LR-Mesh is used. These results will be useful in the more general technique for simulating any semimonotonic configurations (not necessarily incremental).

7.5.3.1 Applications

In this section we apply Theorem 7.10 to some fundamental algorithms (counting N bits, adding N b-bit numbers, and sorting N numbers).

Counting N bits: An $(N+1) \times N$ unit-cost R-Mesh can count the number of 1's among N input bits $b_0, b_1, \cdots, b_{N-1}$ in constant time. Index the rows (resp. columns) of the R-Mesh $0, 1, \cdots, N$ (resp. $0, 1, \cdots, N-1$). Initially, processor $(0, j)$ in row 0 and column j holds input bit b_j. The algorithm involves an initial step to broadcast b_j along column j. This step uses buses with no bends and does not pose any problem on a bends-cost R-Mesh implementation. Subsequent steps involve buses with $\Theta(N)$ bends and we focus our attention only on those.

The algorithm constructs incremental monotonic buses starting at the processors of the left border of the R-Mesh and moving down one row each time a 1 is encountered (see Figure 7.18).

The bus starting at processor $(0, 0)$ reaches processor $(z, N-1)$ iff the input bits include z 1's. If processor $(0, 0)$ sends a signal from its W port, it will reach processor $(z, N-1)$ where z is the number of 1's. Obviously all buses are of type A. All other bus types could be ignored as they do not carry the signal. Clearly, a bus could bend $\Theta(N)$ times (as there could be $\Theta(N)$ 1's in the input). A direct implementation of this algorithm on the bends-cost LR-Mesh could have buses with a delay of $\Theta(N\Delta)$, which is more than that of a naive LR-Mesh implementation using linear-cost buses.

With Theorem 7.10, we have the following result.

Theorem 7.11 *Let Δ denote the delay of an N-element segmentable bus. For any $D \geq \Delta$, a $\Theta(N) \times \Theta(N)$ bends-cost LR-Mesh can count N bits in $O\left(\frac{\log N}{\log D - \log \Delta}\right)$ time using buses of at most D delay.* ■

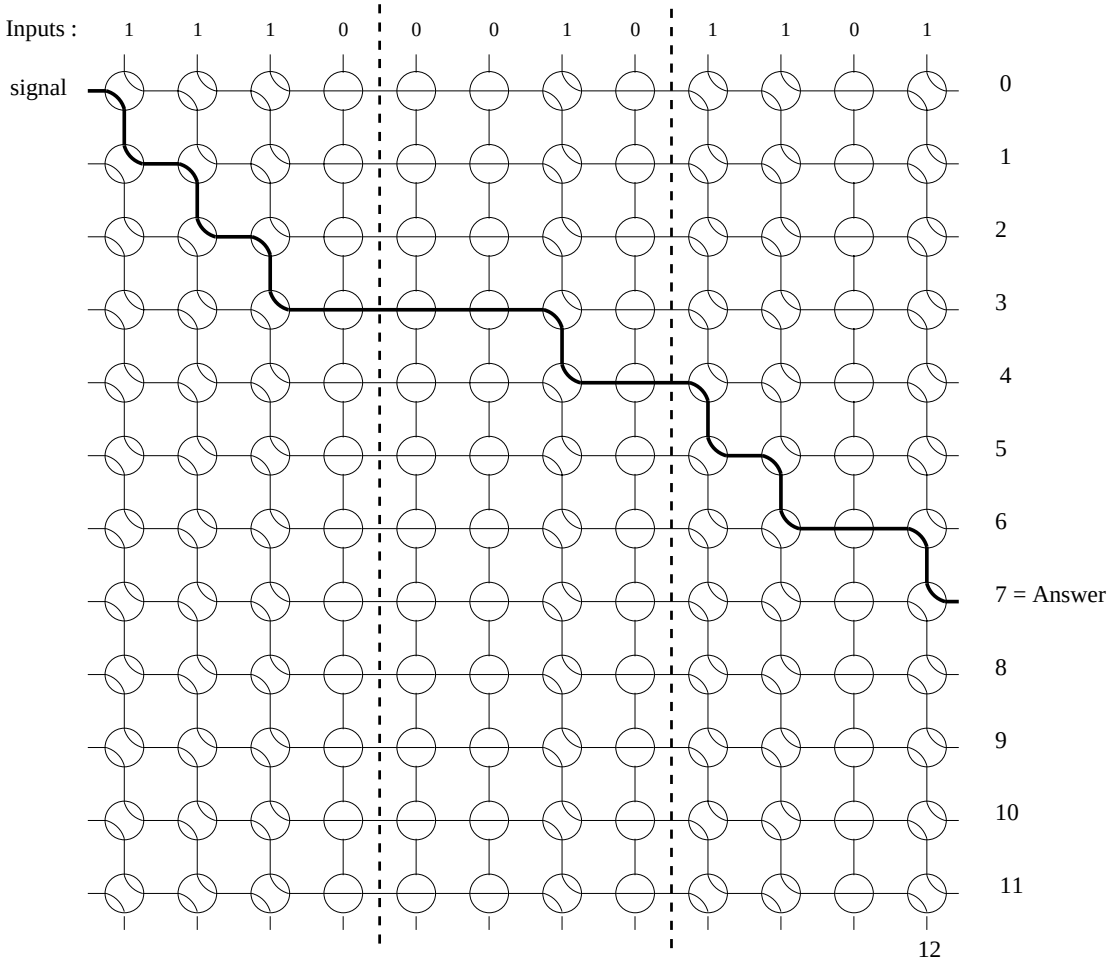


Figure 7.18: Counting bits on the LR-Mesh

Multiple Addition: The multiple addition problem involves adding N b-bit integers (where $b = O(\log N)$). Jang and Prasanna [22] established that a $2N \times 2Nb$ LR-Mesh can solve this problem in constant time. For brevity, we do not go into the details of this algorithm. The following observations distill structural aspects of their algorithm relevant to a bends-cost LR-Mesh implementation.

The algorithm has steps that involve broadcasting within a row or a column and using a segmentable bus within a row or column. These steps use buses with no bends. The only step that does not fall in this category involves constructing

column-monotonic incremental buses and transmitting signals from the West and North edges.

From Theorem 7.10, we have the following result.

Theorem 7.12 *For any $D \geq \Delta$ and $b = O(\log N)$, a $\Theta(N) \times \Theta(Nb)$ bends-cost LR-Mesh can add N b-bit integers in $O\left(\frac{\log N}{\log D - \log \Delta}\right)$ time using buses of at most D delay.* ■

Sorting: The problem of sorting an array of elements $A = (a_0, a_1, \cdots, a_{N-1})$ is to arrange the elements of A in increasing (or decreasing) order. The only assumption about the elements of A is that they are pairwise comparable. The constant time sorting algorithm of Jang and Prasanna [22] runs on an $N \times N$ unit-cost LR-Mesh. We now show that each step of this algorithm can be performed by an incremental configuration.

The algorithm is based on the Leighton's seven-step column sort [25] that requires an $N \times N^{\frac{3}{4}}$ LR-Mesh to sort $N^{\frac{3}{4}}$ elements in four of the seven steps and an N-element permutation routing in the remaining three steps. The following result is standard.

Fact 1: An $N \times N$ LR-Mesh can perform a permutation routing on N elements in a row in $O(1)$ time using buses with at most two bends. ■

Therefore we only need consider the algorithm to sort $N^{\frac{3}{4}}$ numbers on an $N \times N^{\frac{3}{4}}$ LR-Mesh. Jang and Prasanna [22] applied the algorithm of Theorem 7.12 a constant number of times to design an $N^{\frac{1}{4}} \times N^{\frac{3}{4}}$ LR-Mesh algorithm to add $N^{\frac{3}{4}}$ bits. This counting algorithm is used to design the $N^{\frac{3}{4}}$-element sorter as follows. Let a'_i $(0 \leq i < N^{\frac{3}{4}})$ be the elements to be sorted.

Divide the $N \times N^{\frac{3}{4}}$ LR-Mesh into $N^{\frac{3}{4}}$ blocks each of size $N^{\frac{1}{4}} \times N^{\frac{3}{4}}$. The i^{th} block determines (by all possible comparisons and counting (Theorem 7.11)) the number of inputs h_i smaller or equal to a'_i; the comparisons use buses with no bends. The LR-Mesh routes (Fact 1) a'_i to position h_i. Thus, we have the following fact.

Fact 2: An $N^{\frac{1}{4}} \times N^{\frac{3}{4}}$ bends-cost LR-Mesh can sort $N^{\frac{3}{4}}$ elements in $O\left(\frac{\log N}{\log D - \log \Delta}\right)$ steps using buses of at most D delay. ■

Theorem 7.13 *Let Δ denote the delay of an N-element segmentable bus. For any $D \geq \Delta$, a $\Theta(N) \times \Theta(N)$ bends-cost LR-Mesh can sort N elements in $O\left(\frac{\log N}{\log D - \log \Delta}\right)$ time using buses of at most D delay.* ■

We now have the tools to tackle the general channel assignment problem.

7.5.4 General Channel Assignment

In the last section we solved the restricted channel assignment problem on a $\Theta(N) \times \Theta(X)$ bends-cost LR-Mesh with the restriction that the end points of each BUS be no more than X rows apart. Here we remove this restriction. For completeness, we define the channel assignment problem once again.

Let $\mathcal{S}$ be an $N \times X$ LR-Mesh with Type A BUSES $b_0, b_1, \cdots, b_y$. For $1 \leq l < y$, let BUS b_l touch the left border of $\mathcal{S}$ at row s_l (starting row) and the right border at row e_l (ending row). We will call the processors at the left border of row s_l and the right border of row e_l as the left and right ends, respectively, of BUS b_l (see Figure 7.19 for an example). Without loss of generality, assume that for each bus b_l, $e_l - s_l > X$; if not, b_l can be processed separately as in Section 7.5.3. Let $\mathcal{V}$ be a $c_1 N \times c_2 X$ LR-Mesh. The solution to the channel assignment problem is for $\mathcal{V}$ to assign to each BUS b_l a column index m_l (where $0 \leq m_l < X$) satisfying the following condition: for all $0 \leq l < l' < y$, if $m_l = m_{l'}$, then $e_l < s_{l'}$; i.e., buses with the same column index do not overlap.

Assume that no left end of a bus is in the same row as the right end of another bus. This assumption is without loss of generality. For some $l < l'$, if $e_l = s_{l'}$, then stretch $\mathcal{S}$ into a $2N \times X$ LR-Mesh so that all end points are on even rows. If $e_l = s_{l'}$, then move e_l to the previous odd row. The effect of stretching $\mathcal{S}$ can be incorporated into the constant c_1 in the size of $\mathcal{V}$.

Even though $\mathcal{S}$ and $\mathcal{V}$ are different in size, there is clearly a correspondence between their rows and columns. For bus b_l, we will reuse symbols s_l and e_l to also denote corresponding rows of $\mathcal{V}$.

The algorithm for the general channel assignment problem has three main stages.

Stage 1–Leader Determination: Identify the BUSES with the X smallest left ends. Call these BUSES the *leaders*. Without loss of generality, let the leaders be $b_0, b_1, \cdots, b_{X-1}$; that is, for all $0 \leq j < X$ and $X \leq k < y$, row numbers $s_j < s_k$. Assign index $m_j = j$ to leader b_j, for each $0 \leq j < X$. If the number y of BUSES is at most X, then the problem is solved at this point. Therefore, assume that $y > X$.

Stage 2–List Creation: For each $0 \leq j < X$, construct a list L_j of BUSES such that L_j contains BUS b_j and for any two BUSES $b_l, b_{l'}$, if b_l precedes $b_{l'}$ in the list, then $e_l < s_{l'}$; that is, BUS $b_{l'}$ does not start before b_l ends. Consequently, all BUSES in L_j can be assigned the same column index as BUS b_j, namely j. Figure 7.21(b) illustrates the list for the example in Figure 7.19.

Stage 3–Broadcasting in List: For each list L_j, create a bus (in the simulating slice $\mathcal{V}$) that traverses the left end of row s_l for each bus b_l in L_j (the traversal is in the order of list L_j). Then broadcast the column index j to these left end processors.

Each of these stages runs in $O\left(\frac{\log N}{\log D - \log \Delta}\right)$ time on a $c_1 N \times c_2 X$ bends-cost LR-Mesh $\mathcal{V}$ using buses with at most D delay.

Recall that $\mathcal{S}$ is the simulated $N \times X$ unit-cost LR-Mesh and $\mathcal{V}$ is the simulating $c_1 N \times c_2 X$ bends-cost LR-Mesh. Recall also that the buses of $\mathcal{S}$ each have at most D delay, so $\mathcal{V}$ can simulate them directly. As before, we will treat the simulating LR-Mesh $\mathcal{V}$ as an $N \times X$ LR-Mesh for ease of explanation. We now detail the stages of the algorithm.

Figures 7.19–7.23 show a running example.

For this example, $X = 3$ and the slice contains nine BUSES, $b_0, b_1, \cdots, b_8$ (see Figure 7.19).

7.5.4.1 Stage 1—Leader Determination

For each row i of $\mathcal{V}$, set a flag $start(i)$ to 1 iff there is a BUS b_l with $s_l = i$. For each row i with $start(i) = 1$, configure each processor of that row with the partition

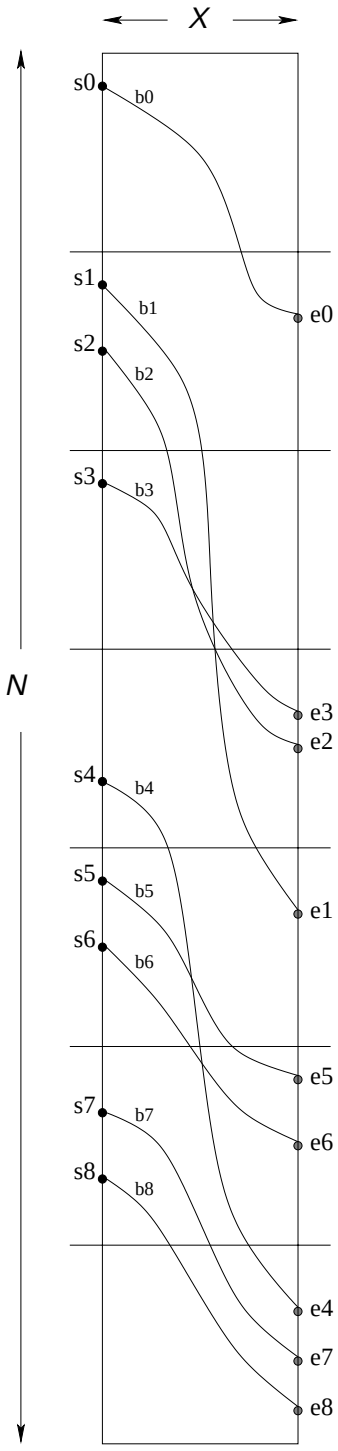


Figure 7.19: An example of the channel assignment problem

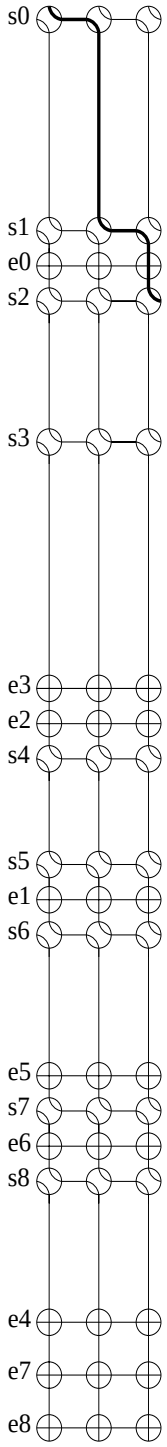


Figure 7.20: Configuration for Stage 1

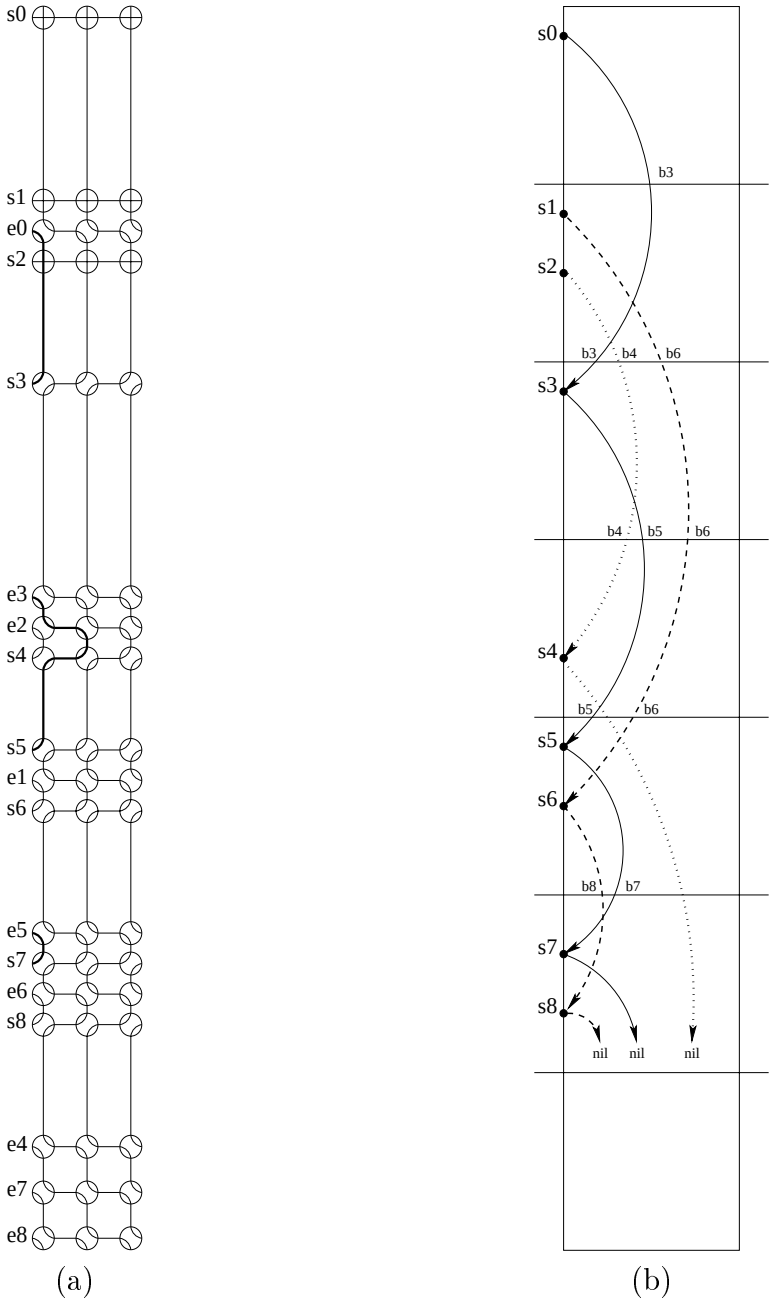


Figure 7.21: Configuration and result of Stage 2; part (b) shows the list connecting starting rows of buses. We use these starting rows as identifiers for the buses. The pointers themselves are labeled with buses only for clarity.

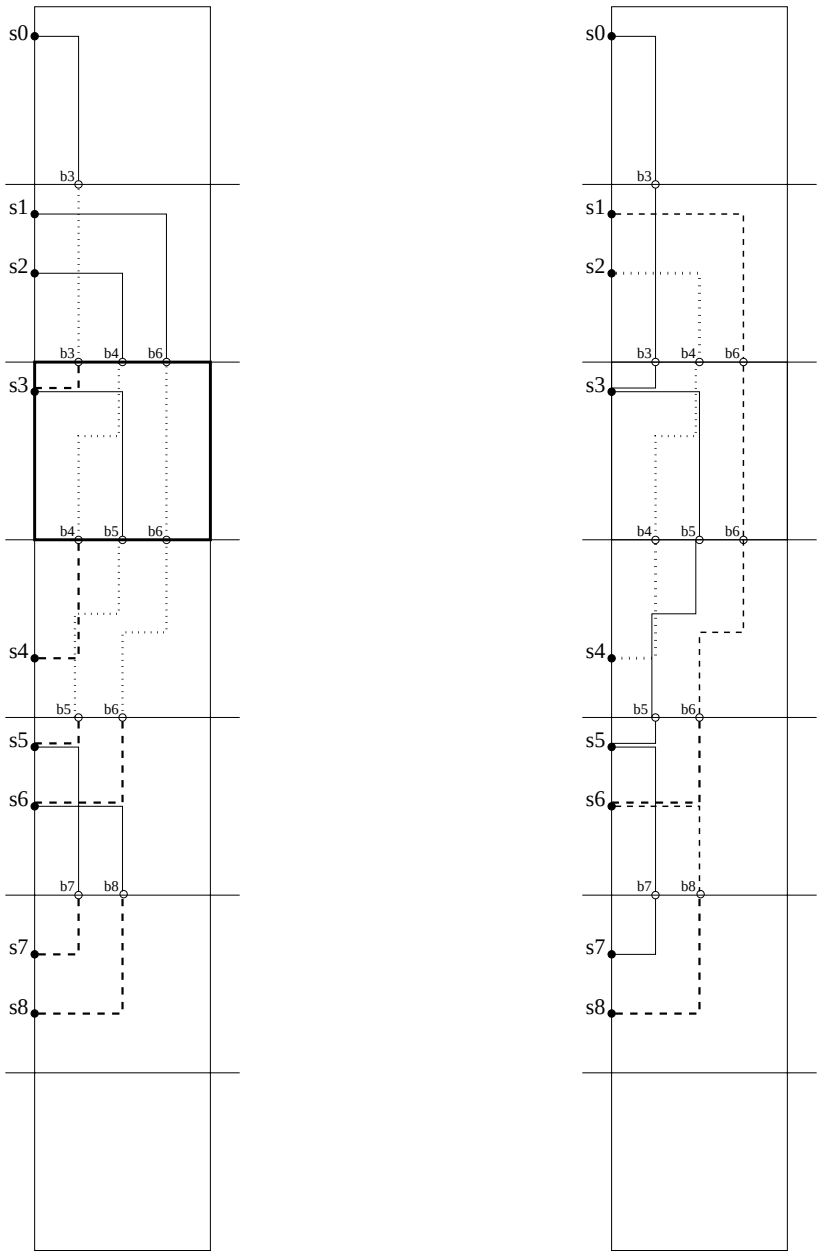


Figure 7.22: Illustration of Stage 3; part (a) shows buses separated by class, part (b) shows buses separated by list.

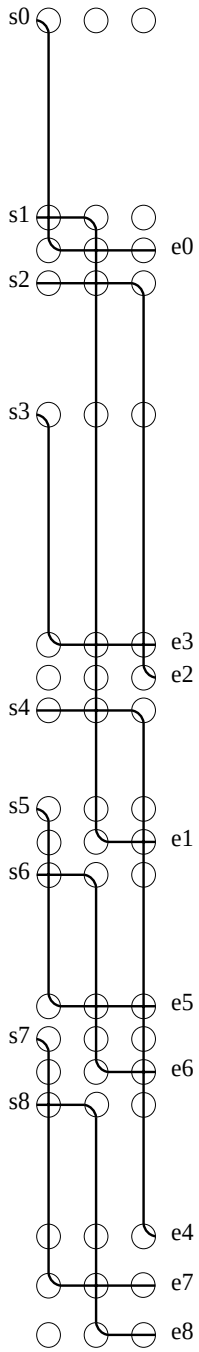


Figure 7.23: The result of channel assignment

$\{\overline{N,E}\ ;\ \overline{S,W}\}$; i.e., connect the N and E ports together and the S and W ports together in the processor. Configure each processor in the remaining rows with the partition $\{\overline{N,S}\ ;\ \overline{E,W}\}$ (see Figure 7.20). Observe that all buses in the configuration described above are row monotonic (column monotonic in the transposed $X \times N$ slice) and incremental. Suppose that the processor at row 0 and column 0 (top left processor) sends a signal through its N port. It is easy to see that for each row i with $start(i) = 1$, the processor at row i and column j receives the signal at its N port iff the BUS starting at row i has the j^{th} smallest starting row index. Thus, this broadcast will not only identify leader BUSES $b_0, b_1, \cdots, b_{X-1}$, but also associate the column index j with each BUS b_j (for $0 \leq j < X$). By Theorem 7.10, LR-Mesh $\mathcal{V}$ can broadcast the signal described above in $O\Big(\frac{\log N}{\log D - \log \Delta}\Big)$ time.

For the running example, BUSES b_0, b_1, b_2 (starting at rows s_0, s_1, s_2) are selected as leaders (Figure 7.20).

7.5.4.2 Stage 2—List Creation

First configure the simulating LR-Mesh $\mathcal{V}$ exactly as the simulated LR-Mesh $\mathcal{S}$; thus each BUS of $\mathcal{S}$ is now a bus of $\mathcal{V}$. Let the processors at the left and right ends of each BUS exchange information about each other. (Since the BUSES of $\mathcal{S}$ have at most D delay, $\mathcal{V}$ can perform this data exchange without excessive delay.) At this point, we may assume that each row of $\mathcal{V}$ is aware of all information about the BUS (if any) that starts or ends at that row.

The algorithm for Stage 2 has three broad steps. The purpose of Step 1 is as follows. Suppose BUS b_l was to precede $b_{l'}$ in list L_i. Then this stage establishes a bus from the left end of row e_l to the left end of row $s_{l'}$. Figure 7.21(a) shows (in bold) the buses established for the list $L_0 = \langle s_0, s_3, s_5, s_7 \rangle$.

Step 1: Configure each row i as described below.

- If $i = s_l$ for some leader BUS b_l (a leader starts at row i), then configure each processor of row i of $\mathcal{V}$ as $\{\overline{N,S}\ ;\ \overline{E,W}\}$.
- If $i = s_l$ for some non-leader BUS b_l (a non-leader starts at row i), then configure each processor of row i of $\mathcal{V}$ as $\{\overline{N,W}\ ;\ \overline{S,E}\}$.

- If $i = e_l$ for some BUS b_l (a bus ends at row i), then configure each processor of row i of $\mathcal{V}$ as $\{\overline{N, E}\ ;\ \overline{S, W}\}$.
- If there is no BUS b_l such that $i = s_l$ or $i = e_l$ (no bus starts or ends at row i), then configure each processor of row i of $\mathcal{V}$ as $\{\overline{N, S}\ ;\ \overline{E, W}\}$.

Step 2: The leftmost processor in each non-leader starting row $s_{l'}$ writes $s_{l'}$ (the identifier for bus $b_{l'}$) to its W port and the leftmost processor in ending row e_l reads from its W port.

Step 3: If e_l reads $s_{l'}$ in Step 2, then BUS b_l points to BUS $b_{l'}$ in its list. If e_l does not receive anything on its W port, then BUS b_l is the last element of its list. For our example, the end result of this stage is shown in Figure 7.21(b).

To see why this algorithm works, consider the sequence σ' of rows s_l and e_l for all $0 \leq l < y$ in ascending order. For our example $\sigma' = \langle s_0, s_1, e_0, s_2, s_3, e_3, e_2, s_4, s_5, e_1, s_6, e_5, s_7, e_6, s_8, e_4, e_7, e_8\rangle$ (see Figure 7.21(a)). From this sequence remove the first X starting rows (of leaders) and the last X ending rows. Let the resulting sequence be σ. For the example, $\sigma = \langle e_0, s_3, e_3, e_2, s_4, s_5, e_1, s_6, e_5, s_7, e_6, s_8\rangle$.

In general, let $\sigma = \langle \pi_1, \pi_2, \cdots, \pi_{2z}\rangle$ where π_i is some $e_{l'}$ or $s_{l''}$. The sequence σ has to have an even number of elements. This is because σ' has matching s_l, e_l pairs and σ is derived from σ' by removing X starting rows and X ending rows.

Lemma 7.14 *For any $1 \leq k \leq 2z$, let $\sigma_k = \langle \pi_1, \pi_2, \cdots, \pi_k\rangle$. Let σ_k have n_e ending rows and n_s starting rows. Then, $n_e \geq n_s$.*

Proof: Let

$$\sigma'_k = \langle \underbrace{s_0, s_1, \cdots, s_{X-1}}_{X \text{ starts}},\ \underbrace{\pi_1, \pi_2, \cdots, \pi_k}_{n_s \text{ starts, } n_e \text{ ends}}\ \rangle$$

Clearly, σ'_k has $X + n_s$ starting rows and n_e ending rows. If we examine the given slice S just after row π_k, then $X + n_s$ Type A BUSES would have started, of which n_e would have ended. Therefore $X + n_s - n_e$ BUSES cross row π_{k+1}. Since $\mathcal{S}$ has X columns, $X + n_s - n_e \leq X$; i.e. $n_s \leq n_e$. ■

Lemma 7.14 together with the fact that sequence σ has the same number of starting and ending rows, allows σ to be viewed as a well-nested parentheses sequence

(see Section 3.4) by simply replacing each ending row with an opening parentheses and a starting row by a closing parentheses. Thus each ending row e_l has a matching starting row $s_{l'}$ in sequence σ. Since $s_{l'} > e_l$, $l' \neq l$.

Lemma 7.15 *For each matching pair $(e_l, s_{l'})$ of sequence σ, Step 1 of Stage 2 establishes a bus between the* W *port of the left end processor of rows e_l and $s_{l'}$.*

Proof outline: Let p_l and $p_{l'}$ be the left end processors of rows e_l and $s_{l'}$. The bus from the W port of $p_{l'}$ moves right by one column for each starting row (including itself) it traverses and left by one column for each ending row. Thus this bus can reach a W port on the left border only at processor p_l in the matching row e_l. ■

Thus in Step 2 of Stage 2, the left end of e_l reads a row number $s_{l'}$ iff $(e_l, s_{l'})$ is a matching pair. That is, in Step 3, bus b_l points to bus $b_{l'}$ iff $(e_l, s_{l'})$ is a matching pair.

Lemma 7.16 *The three step procedure of Stage 2 is correct.*

Proof: By the same argument, each BUS $b_{l'}$ is pointed to from (at most) one BUS b_l. Thus the pointers of all BUSES constitute a set of lists. Since $e_l < s_{l'}$, it is clear that if b_l points to $b_{l'}$, then both BUSES can occupy the same column (as required).

We now show that there is one list per leader. Since b_l points to $b_{l'}$ in a list iff $(e_l, s_{l'})$ is a matching pair, and since $s_0, s_2, \cdots, s_{X-1}$ are absent from σ, no BUS can point to a leader. That is, each leader heads a list. We now show that no non-leader heads a list; i.e., each non-leader is an element of a list headed by a leader. For each non-leader BUS $b_{l'}$, its starting row $s_{l'}$ is in the sequence σ. Therefore, there is an e_l in σ such that $(e_l, s_{l'})$ is a matching pair and so b_l points to $b_{l'}$. ■

As is also evident from Figure 7.21(a), the buses created in Stage 2 are row monotonic and incremental. By Theorem 7.10, Stage 2 runs in $O\left(\frac{\log N}{\log D - \log \Delta}\right)$ time.

7.5.4.3 Stage 3—Broadcasting in List

This stage first constructs a bus corresponding to each list L_i, where $0 \leq i < X$; specifically, if list $L_i = \langle b_i, b_{i(1)}, b_{i(2)}, \cdots, b_{i(u)} \rangle$, then this stage constructs a bus that traverses the left end of rows $s_i, s_{i(1)}, s_{i(2)}, \cdots, s_{i(u)}$ in that order. In other words,

each pointer in a list corresponds to a segment of the bus representing that list. This stage uses Theorem 7.10 on slice $\mathcal{S}$ to simulate the above buses in $O\left(\frac{\log N}{\log D - \log \Delta}\right)$ time and broadcasts the column index of the leader of the list to all other BUSES within the list. The only point requiring further elaboration is the construction of buses corresponding to lists L_i.

We start by dividing the simulating LR-Mesh $\mathcal{V}$ into $\frac{N}{X}$ "windows," each an $X \times X$ sub-LR-Mesh consisting of X contiguous rows of $\mathcal{V}$. Call the topmost and bottommost rows of a window as its *borders*. Recall that for each bus b_l, $\quad e_l - s_l > X$. Therefore the lists are such that a BUS within a window points to a BUS outside the window. Let BUS b_l points to BUS $b_{l'}$ in some list. All we need to do is construct a bus from left end processor p_l of row s_l to the left end processor $p_{l'}$ of row $s_{l'}$ for all such BUS pairs $(b_l, b_{l'})$. The algorithm has two phases. In Phase 1, windows collect and record information about buses (pointers of lists) crossing them. In Phase 2, the windows use this information to independently route buses passing through them.

Phase 1: Phase 1 has three broad steps.

- First configure $\mathcal{V}$ exactly as $\mathcal{S}$ to establish the input Type A BUSES.
- Broadcast on each bus b_l of $\mathcal{V}$ the identifier s_l and its pointer $s_{l'}$ (assuming b_l points to $b_{l'}$). Each processor in a window border through which b_l passes collects this information and records it. Bus b_l may cross a window border several times. This could cause pointer $s_{l'}$ to be recorded multiple times in a border, whereas the algorithm requires the border to record each pointer only once. This can be done by sorting the (at most) X pointers crossing a window border and then selecting only the first occurrence of each pointer value. By Theorem 7.13, this part runs in $O\left(\frac{\log X}{\log D - \log \Delta}\right) = O\left(\frac{\log N}{\log D - \log \Delta}\right)$ time.
  Also BUS b_l may go below row e_l and then come up again to end at the right border of row e_l. Since we will establish a direct path between rows s_l and $s_{l'}$, each border at row $i > s_{l'}$ (or row $i < s_l$) ignores the pointer (does not record it).
- Construct (exactly as in Step 1 of Stage 2) a bus from the left end processor of row e_l to the left end processor of row $s_{l'}$. Broadcast on each bus the value of

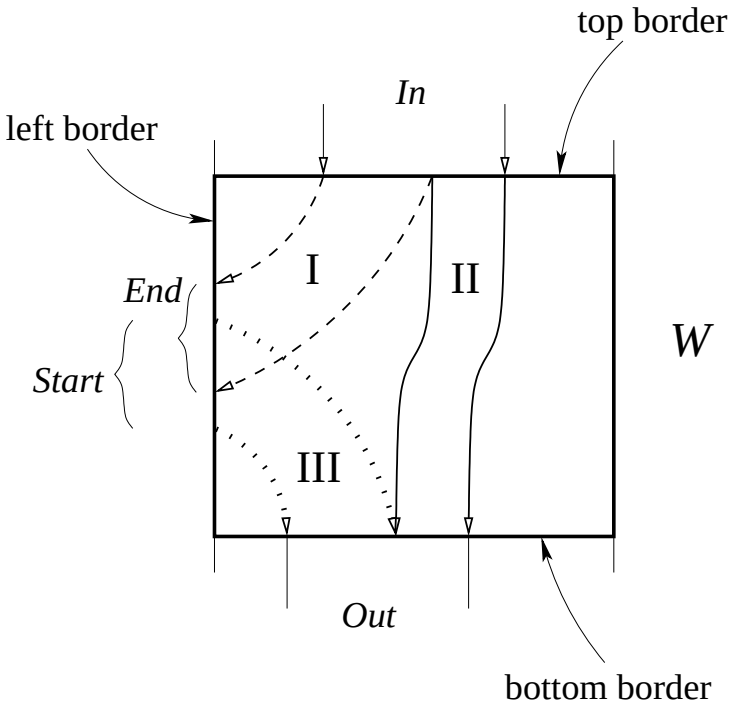


Figure 7.24: Illustration of Stage 2

$s_{l'}$. As before, record the value of $s_{l'}$ once at each border crossing for a direct path.

At the end of Phase 1, each border has recorded information about each pointer that must cross it in a direct path between BUS b_l and its successor $b_{l'}$ in the list. Each window also has information about all starting rows within that window.

Phase 2: Here we use the information recorded in Phase 1 to construct buses according to the list obtained in Stage 2.

Consider any window $\mathcal{W}$ that has a set *In* of incoming pointers (buses in $\mathcal{V}$) recorded at its top border. Let *Out* be a set of outgoing pointers recorded at its bottom border. Let *End* be the set of ending rows within the window $\mathcal{W}$. Let *Start* be the set of starting rows within the window $\mathcal{W}$ (see Figure 7.24).

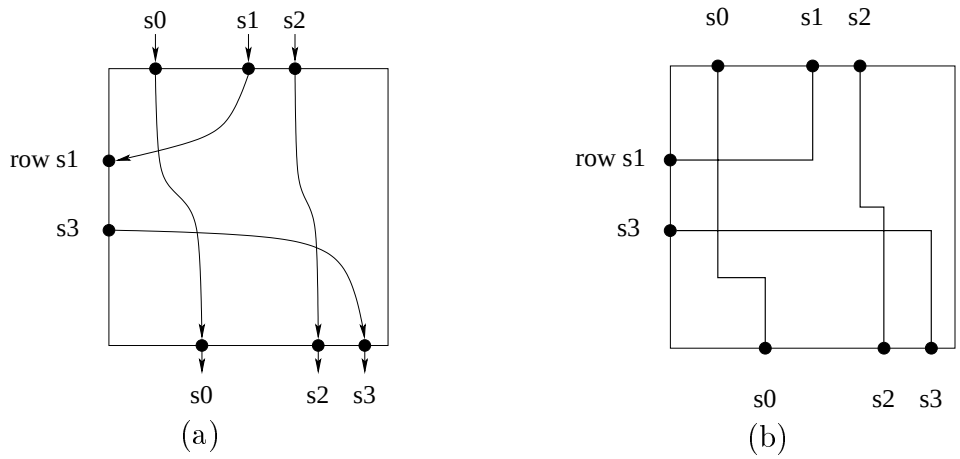


Figure 7.25: Examples of pointers in a window

To illustrate the action of phase 2 consider a window with $In = \{s_0, s_1, s_2\}$, $Out = \{s_0, s_2, s_3\}$, $End = \{s_1\}$, and $Start = \{s_3\}$ (see Figure 7.25(a)). That is, the pointers to buses b_0, b_1, b_2 enter the window from the top border. Of these, the pointer s_1 is to a row within the window, so this pointer exits at row s_1. Another row in the window starts a pointer s_3 (possibly continuing where s_1 ended) and this pointer exits the window along with s_0 and s_2 through the bottom border.

The task of phase 2 is to create "corresponding buses" in accordance with these pointers. For our example, the window may create row monotonic buses that may be as shown in Figure 7.25(b). This task is accomplished by the information gathered in Phase 1. Divide the pointers into three classes:

- Class I consists of pointers in set $End = In - Out$. Their corresponding buses run between the top and left borders of the window.
- Class II consists of pointers in set $Out - Start$. Their corresponding buses run between the top and bottom borders of the window.
- Class III consists of pointers in set *Start*. Their corresponding buses run between the left and bottom borders of the window.

It should be clear that each pointer entering or exiting the window falls in exactly one class. Each window independently constructs the corresponding (row monotonic) buses for its pointers.

Each class of buses is routed on a different tier of processors so that all corresponding buses can be accommodated on three tiers (see discussion on page 144). For the running example, consider the window shown in bold in Figure 7.22(a). Set $In = \{s_3, s_4, s_6\}$, set $Out = \{s_4, s_5, s_6\}$, set $End = \{s_3\}$ and set $Start = \{s_5\}$. The figure shows the corresponding buses, dashed, dotted and solid for Classes I, II, III.

Since each window constructs row monotonic buses corresponding to its pointers, the buses representing the lists in Stage 2 are row monotonic as well (see Figure 7.22(a)).

By Theorem 7.10, $\mathcal{V}$ completes Stage 3 in $O\left(\frac{\log N}{\log D - \log \Delta}\right)$ time.

Lemma 7.17 *Let Δ be the delay of an N-processor segmentable bus. For any $D \geq \Delta$, the channel assignment problem on an $N \times X$ slice can be solved in $O\left(\frac{\log N}{\log D - \log \Delta}\right)$ time on a $\Theta(N) \times \Theta(X)$ bends-cost LR-Mesh using buses with at most D delay.* ■

With Equation 7.1 and the above lemma, we have Theorem 7.7 and Corollary 7.8 stated at the start of this section.

In the next two subsections we show how the general channel assignment problem can be applied to efficiently computing the prefix sums of N bits.

7.5.4.4 Prefix Sums of Bits

Here we apply Theorem 7.7 to compute the prefix sums of N input bits. That is, for bits $a_0, a_1, \cdots, a_{N-1}$, we compute $b_0, b_1, \cdots, b_{N-1}$ where $b_i = \sum_{j=0}^{i} a_j$ for each $0 \leq i < N$. The algorithm starts with an inefficient approach and progressively refines it.

Inefficient Prefix Sums: The R-Mesh counting algorithm also gives the prefix sums of the input bits. If the j^{th} prefix sum ($0 \leq j < N$) is b_j, then the signal reaches the E port of processor (b_j, j). Since our transformation maintains only the end points of buses, the bends-cost R-Mesh algorithm will not directly yield the prefix sums. By

reversing the recursion, the bends-cost R-Mesh can easily compute the prefix sums, however. Each processor within a slice holds the identity of the left and right ends of its bus. If the bus through processor (b_j, j) has processor β as its left end, then the j^{th} prefix sum is b_j iff processor β receives the signal. Reversing the steps of the counting algorithm (that goes from thin slices to wider slices) returns information from wider slices back to thin slices and ultimately to individual columns.

Lemma 7.18 *For any $D \geq \Delta$, a $\Theta(N) \times \Theta(N)$ bends-cost LR-Mesh can find the prefix sums of N bits in $O\left(\frac{\log N}{\log D - \log \Delta}\right)$ time using buses of at most D delay.* ■

Modulo Prefix Sums: For any $m \geq 1$ the jth modulo m prefix sum of input bits $a_0, a_1, \cdots, a_{N-1}$ is $(a_0 + a_1 + \cdots + a_j) (\text{mod } m)$. The modulo m prefix sums can be computed on an $(m+1) \times 2N$ unit-cost R-Mesh in constant time [36]. This algorithm uses buses of Types A and B. While a Type A bus is incremental, Type B buses could extend between the first and last rows; that is, their left and right ends in an $m \times X$ slice could be more than X rows apart. From Theorem 7.7, we have the following result.

Lemma 7.19 *Let Δ denote the delay of an N-element segmentable bus. For any $1 \leq m \leq N$ and $D \geq \Delta$, a $\Theta(m) \times \Theta(N)$ bends-cost LR-Mesh can compute the modulo m prefix sums of N bits in $O\left(\left(\frac{\log N}{\log D - \log \Delta}\right)^2\right)$ time using buses of at most D delay.*

Olariu *et al.* [36] proved that using modulo m prefix summing, an $m \times N$ (unit-cost) LR-Mesh can compute the prefix sums of N bits in $O\left(\frac{\log N}{\log m}\right)$ time. With the result of Lemmas 7.18 and 7.19, we have the following result.

Theorem 7.20 *Let Δ be the delay of an N-processor segmentable bus. For any $1 \leq m \leq N$ and $D \geq \Delta$, a $\Theta(m) \times \Theta(N)$ bends-cost LR-Mesh can compute the prefix sums of N bits in $O\left(\left(\frac{\log N}{\log m}\right)\left(\left(\frac{\log N}{\log D - \log \Delta}\right)^2\right)\right)$ steps using buses with delay of at most D.*

Remark: Once again if $D = N^{\epsilon_1}$ for constant $\epsilon_1 > 0$, then the time is $O\left(\frac{\log N}{\log m}\right)$. In addition, if $m = N^{\epsilon_2}$ for constant $\epsilon_2 > 0$, then the time is constant.

It is possible to modify the simulation of Lemma 7.19 to reduce the overhead to $O\left(\frac{\log N}{\log D - \log \Delta}\right)$ [16]. Hence, the time overhead of Theorem 7.20 can be reduced to $O\left(\left(\frac{\log N}{\log m}\right)\left(\frac{\log N}{\log D - \log \Delta}\right)\right)$. However, it should be stressed that this reduction in time comes from exploiting properties of the modulo prefix sums algorithm and does not translate to any improvement in the result of Theorem 7.7.

7.5.5 Special Cases

In this section we show that the time overhead of Theorem 7.7 can be reduced for some special cases of a semimonotonic configuration. Let the simulated slice be an $N \times X$ unit-cost LR-Mesh. Recall that $x = \frac{D}{\Delta}$ which is the maximum number of bends that a bus of the simulating bends-cost LR-Mesh can have. We consider two spesial cases.

Oscillating Configurations: These configuration are semimonotonic configurations in which each Category 2 (Type A or Type B) bus has left and right ends s and e from two fixed x-elements subsets S_1, S_2 of $\{0, 1, \cdots, N-1\}$ (see Figure 7.26). Thus every bus must either stay in the same subset S_1 or S_2, or oscillate between them. Although Figure 7.26 shows only buses starting from the leftmost corner of the LR-Mesh, the definition of an oscillating configuration admits the "mirror range" band of buses starting at the bottom left of the figure. In general, an oscillating configuration restricts end points of buses within slices to oscillate between two fixed ranges of x rows. These ranges need not be the topmost and bottommost x rows as shown in Figure 7.26. Since the number of buses is at most $x \leq X$, the number of columns within the slice, then the static channel assignment used in Section 7.5.3 can be used to assign each bus to a channel. This can be done in constant time and the entire simulation algorithm runs in $O\left(\frac{\log N}{\log D - \log \Delta}\right)$ time.

Theorem 7.21 *Let Δ denote the delay of an N-element segmentable bus. For any $D \geq \Delta$, any oscillating configuration of an $N \times N$ unit-cost LR-Mesh can be simulated in $O\left(\frac{\log N}{\log D - \log \Delta}\right)$ time on a $\Theta(N) \times \Theta(N)$ bends-cost LR-Mesh whose buses have at most D delay.* ■

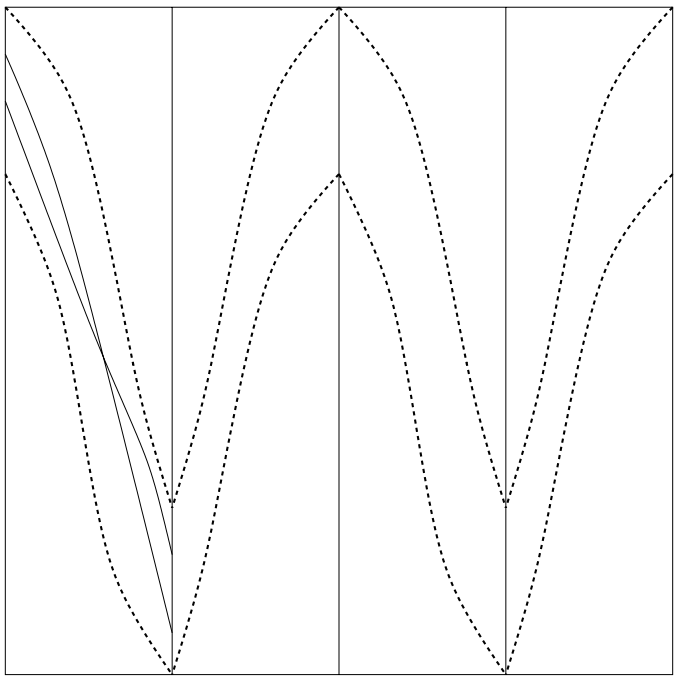


Figure 7.26: Example of an oscillating configuration

Parallel Configurations: parallel configuration of size y is a monotonic configuration in which each Category 2 (Type A or Type B) bus has left and right ends s and e that satisfy $|s-e| = y$, where $y \in \{0, 1, \cdots, N-1\}$ (see Figure 7.27).

Consider an $N \times N$ unit-cost LR-Mesh with a parallel configuration. Without loss of generality let $\frac{y}{x}$ and $\frac{N}{x}$ be integers. Divide this into $N \times x$ slices as before. Divide each slice into $x \times x$ windows. Number the windows of a slice from $0, 1, \cdots, \frac{N}{x} - 1$. Let the modulo index of a window of (actual) index i be $i(\mathrm{mod}(\frac{y}{x}))$. For $0 \leq j < y$, let S_j be the set of all buses that cross a left border of a window with modulo index j. Figure 7.27 shows the buses with modulo index 0.

Notice that all buses of any fixed set S_j can be assigned a channel (within slices) statically as in Section 7.5.3. Thus S_j buses can be simulated with bounded delay D in $O\left(\frac{\log N}{\log D - \log \Delta}\right)$ time. To simulate all $\frac{y}{x}$ sets S_j, we need at most $\frac{y}{x}$ iterations. If $\frac{y}{x} > \left(\frac{\log N}{\log D - \log \Delta}\right)$, we can use the algorithm of Theorem 7.7.

Theorem 7.22 *Let Δ denote the delay of an N-element segmentable bus. For any $D \geq \Delta$, any parallel configuration of size y of an $N \times N$ unit-cost LR-Mesh can be*

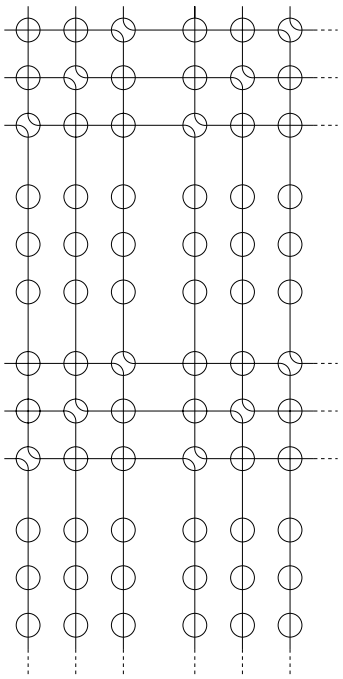


Figure 7.27: Example of a parallel configuration

simulated in $O\Big(\Big(\min\Big\{\Big(\frac{\log N}{\log D - \log \Delta}\Big), \frac{y}{x}\Big\}\Big)\Big(\frac{\log N}{\log D - \log \Delta}\Big)\Big)$ *time on a* $\Theta(N) \times \Theta(N)$ *bends-cost LR-Mesh whose buses have at most* D *delay.* ■

7.6 Simulating General Configurations

Here we present results for simulating the LR-Mesh (with not necessarily semimonotonic buses) and the general R-Mesh on bends-cost LR-Meshes. Recall the definitions of row and monotonic subsequence of a linear bus (see page 124).

Definition 7.2 Let b be an acyclic piece of a linear bus. The piece b is a row (or column) *U-turn* of the bus if and only if b has a row (or column) subsequence of the form $\langle i, j, i \rangle$. ■

In Figure 7.1 the portions of the dashed bus between rows 0 and 1, and between rows 1 and 2 are two column U-turns. Clearly a bus has no row (or column) U-turn iff it is row (or column) monotonic.

One way to quantify the amount by which a bus is "not semimonotonic" is by the number of U-turns (which cause it to lose its monotonicity). Let B be the set of

buses in an LR-Mesh configuration. For any bus $b \in B$, let $\vartheta_r(b)$ (resp., $\vartheta_c(b)$) be the number of row (resp., column) U-turns in b. Let $\nu_r(B) = \max\{\vartheta_r(b) \ : \ b \in B\}$ and let $\nu_c(B) = \max\{\vartheta_c(b) \ : \ b \in B\}$. The number of U-turns in an LR-Mesh configuration with bus set B is $\nu(B) = \min\left\{\nu_r(B), \nu_c(B)\right\}$.

Theorem 7.23 *Let Δ be the delay of an N-element segmentable bus. For integers D, ϑ such that $D \geq \Delta\vartheta$, a configuration of an $N \times N$ unit-cost LR-Mesh with ϑ U-turns can be simulated in $O\left(\frac{\log^2 N}{(\log D - \log \Delta)(\log D - \log \vartheta - \log \Delta)}\right)$ time on a $\Theta(N) \times \Theta(N)$ bends-cost LR-Mesh using buses with at most D delay.*

Proof outline: This result follows on the same lines as the result of Theorem 7.7. The main difference is that $x = \frac{cD}{\Delta\vartheta}$ to guarantee that each slice uses buses with delay at most D. The number of levels of recursion is therefore $O\left(\frac{\log N}{\log D - \log \vartheta - \log \Delta}\right)$. Each level involves the solution to the channel assignment problem. This solution runs in unaltered in $O\left(\frac{\log N}{\log D - \log \Delta}\right)$ time as it is based on end points of buses rather than their shapes.

The only other issue is that a slice could now have Type L and M buses (see Figure 7.28) in addition to Type A and B buses. The double bus scheme of Section 7.1.1 can be used to identify Type L and M buses and distinguish their end points. After that they can be handled exactly as Type A and B buses. ■

Remark: If $\frac{D}{\vartheta} = N^{\epsilon}$ for an arbitrarily small constant $\epsilon > 0$, then the simulation overhead is a constant.

Matsumae and Tokura [31] proved that an $N \times N$ HVR-Mesh can simulate any step of an R-Mesh in $O(\log^2 N)$ time. Since an HVR-Mesh uses only horizontal and vertical buses with no bends, we have the following result.

Theorem 7.24 *Let Δ be the delay of an N-element segmentable bus. Any configuration of an $N \times N$ unit-cost R-Mesh can be simulated in $O(\log^2 N)$ time on an $N \times N$ bends-cost HVR-Mesh using buses with at most Δ delay.* ■

7.7 Concluding Remarks

We introduced the bends-cost measure of bus delay in linear reconfigurable meshes and showed this measure to be a faithful reflection of bus delay on an implementable

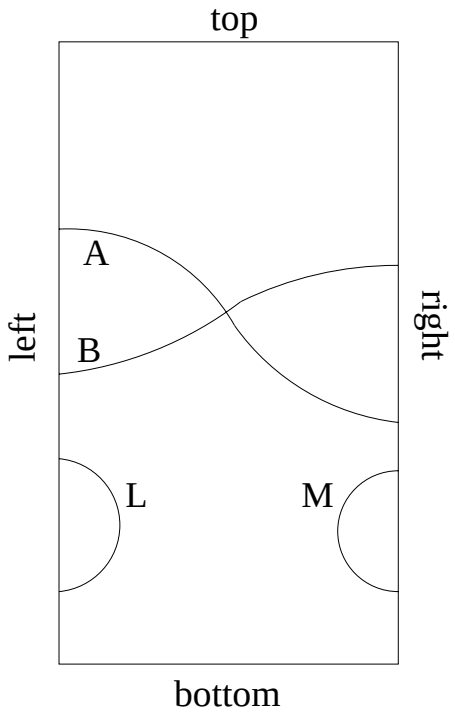


Figure 7.28: Bus types with U-turns

platform. We also presented simulations for several classes of LR-Mesh configurations on the bends-cost model that uses limited delay buses. We showed that an important class of LR-Mesh algorithms can be implemented using limited delay buses. In particular, we showed that it is possible to design constant time algorithms on reconfigurable models without resorting to the unit-cost assumption.

Chapter 8

Computational Power of the Bends-Cost LR-Mesh

Two models of computation $\mathcal{M}_1$ and $\mathcal{M}_2$ are said to have the same power if an arbitrary step of one can be simulated on the other in $O(1)$ steps, allowing polynomial blowup in size for the simulating model. For R-Mesh type models, a model's size is the number of processors in it. In this chapter we prove that if the allowed delay for buses is polynomial in the number of processors, then the unit-cost LR-Mesh and the bends-cost LR-Mesh are equal in power. Specifically, we show that any step of an $N \times N$ unit-cost LR-Mesh can be simulated in constant time on an $N^{\Theta(1)} \times N^{\Theta(1)}$ bends-cost LR-Mesh whose buses have at most N^{ϵ} delay, for any constant $\epsilon > 0$.

Key to this result is a simulation of a step of an $N \times N$ unit-cost LR-Mesh on a $\Theta\left(\frac{DN^2}{\Delta}\right) \times \Theta\left(\frac{DN^2}{\Delta}\right)$ bends-cost LR-Mesh in $O\left(\frac{\log N}{\log D - \log \Delta}\right)$ time, using buses of at most D delay; Δ is the delay of an N-processor segmentable bus. Our approach is based on a well-known R-Mesh list ranking technique called distance embedding [19].

Generally speaking, the idea is as follows. Suppose that the buses of the simulating bends-cost LR-Mesh can have at most β bends (to limit the delay). If a bus has $B > \beta$ bends, then cut the bus after every β bends into $\lceil \frac{B}{\beta} \rceil$ segments. Then, proceeding along the lines of Chapter 7, replace each bus segment (that has at most β bends) by another segment connecting the same end points but with a constant number c of bends. The new bus has at most $c\lceil \frac{B}{\beta} \rceil < B$ bends. Proceed recursively until the entire bus has at most β bends, at which point it can be simulated directly. By reversing the recursion, bus value can be propagated back to each port.

We use an R-Mesh technique for list ranking [47] to cut the bus correctly after a sequence of β bends.

8.1 The Simulation Algorithm

By Lemma 7.3 (page 127), there is no loss of generality in assuming that the simulated LR-Mesh has no cyclic buses. By Lemma 7.1 (page 126), each bus is oriented. So, we may describe a bus as going from port u to port v.

Let $\rho = \frac{D}{2c\Delta}$. The simulation of an LR-Mesh $\mathcal{S}$ on a $4\rho N^2 \times 4\rho N^2$ bends-cost LR-Mesh $\mathcal{V}$ has the following steps.

1. Divide $\mathcal{V}$ into a $4N^2 \times 4N^2$ grid of submeshes, each of size $\rho \times \rho$. Denote the submesh in row i and column j (where $0 \leq i, j \leq 4N^2$) of this grid as $\mathcal{V}_{i,j}$.

2. Number the $4N^2$ ports of the unit-cost LR-Mesh $0, 1, \cdots, 4N^2 - 1$. Each port i (where $0 \leq i \leq 4N^2$) is represented by diagonal submesh $\mathcal{V}_{i,i}$.

3. Let the diagonal processors of $\mathcal{V}_{k,k}$ (where $0 \leq k \leq 4N^2$) be p_u^k, where $0 \leq u < \rho$. If an oriented bus goes from port i to port j in $\mathcal{S}$, then connect processor p_u^i to $p^j_{(u+1)(\text{mod } \rho)}$, for each $0 \leq u < \rho$. If $i < j$, then use submeshes $\mathcal{V}_{i,i}$, $\mathcal{V}_{i+1,i}$, $\cdots$, $\mathcal{V}_{j,i}$, $\mathcal{V}_{j,i+1}$, $\cdots$, $\mathcal{V}_{j,j}$. to establish the connection. Otherwise, use submeshes $\mathcal{V}_{i,i}$, $\mathcal{V}_{i,i+1}$, $\mathcal{V}_{i,j}$, $\cdots$, $\mathcal{V}_{i+1,j}$, $\mathcal{V}_{i+2,j}$, $\cdots$, $\mathcal{V}_{j,j}$. Figure 8.1 shows an example for connecting port u to port v (where $u < v$) and port v to port w (where $w < v$).

   At this point some processor p_u^i is connected through a bus to another processor $p_v^{i'}$ iff there are $(v - u)(\text{mod } \rho)$ ports on the bus between ports i and i' of $\mathcal{S}$. This is the standard technique for contacting a list on the R-Mesh.

4. Cut the bus traversing processor p_0^i, for each $0 \leq i < 4N^2$.

5. Each processor p_0^i writes its index in the direction of the bus oriented towards the next port of i in $\mathcal{S}$. This write by p_0^i traverses a bus of $\mathcal{V}$ with 2ρ bends (or D delay). This index reaches port $p_0^{i'}$ iff on the bus of $\mathcal{S}$, ports i and i' are separated by exactly ρ ports. Similarly, $p_0^{i'}$ sends its index i' to p_0^i.

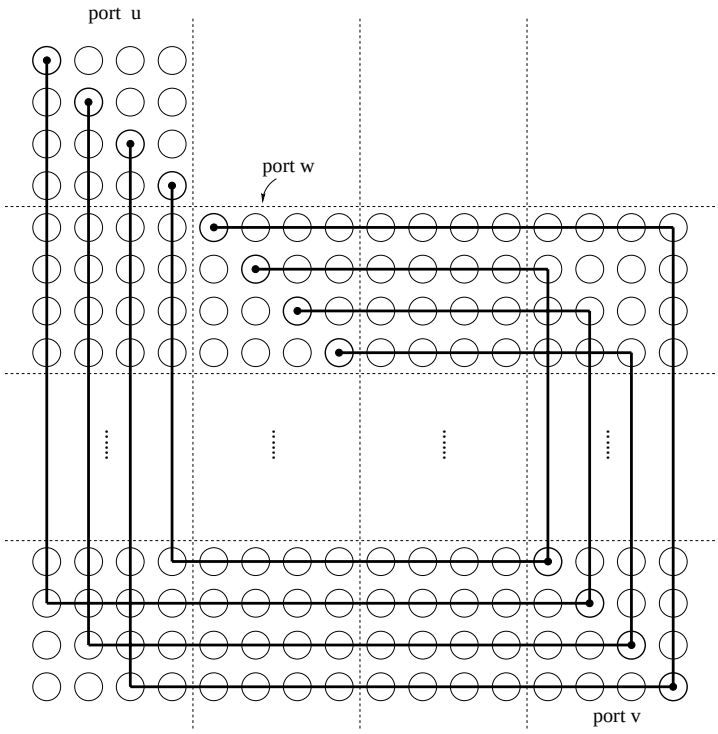


Figure 8.1: Some CST switch configurations

6. Now $\mathcal{V}$ assumes port i to be connected directly to port i'. Accordingly, connect $\mathcal{V}_{i,i}$ to $\mathcal{V}_{i',i'}$ as in Step 3.

7. Repeatedly reduce the bus by a factor of ρ in each iteration till the bus is of size ρ or less.

Clearly, $\log_\rho 4N^2 = O\Big(\frac{\log N}{\log D - \log \Delta}\Big)$ iterations suffice.

Theorem 8.1 *Let Δ be the delay of an N-element segmentable bus. Any configuration of an $N \times N$ unit-cost R-Mesh can be simulated in $O\Big(\frac{\log N}{\log D - \log \Delta}\Big)$ time on an $O\Big(\frac{DN^2}{\Delta}\Big) \times O\Big(\frac{DN^2}{\Delta}\Big)$ bends-cost R-Mesh using buses with at most D delay.* ■

Corollary 8.2 *For any $\epsilon > 0$, any step of an $N \times N$ unit-cost LR-Mesh can be simulated in $O(1)$ time on an $O\Big(\frac{DN^2}{\Delta}\Big) \times O\Big(\frac{DN^2}{\Delta}\Big)$ bends-cost LR-Mesh using buses with at most N^ϵ delay.* ■

Theorem 8.3 *The unit-cost and bends-cost LR-Meshes are equal in power if a polynomial delay is permitted.* ■

8.2 Concluding Remarks

In this chapter we proved that LR-Meshes with sublinear (but polynomial) bus delay are as powerful as the unrestricted LR-Mesh with linear bus delay. This result places the role of bus delay in the context of the power hierarchy of reconfigurable models [5, 46, 48].

Chapter 9

The Enhanced-SRGA

In this chapter we introduce a reconfigurable architecture, the Enhanced Self Reconfigurable Gate Array (E-SRGA) Architecture, that is based on the SRGA architecture proposed by Sidhu *et al.* [40] (see also Section 1.2.1). The SRGA is an FPGA-type architecture with the additional ability to generate configuration information from within the chip (self reconfiguration), for instance, to connect two PEs. This eliminates the need to load the chip with configuration information through the limited number of input pins of the chip. Like the SRGA, the E-SRGA consists of an array of PEs. Each row and column connect by CSTs. It also has the self reconfiguration feature of the SRGA. However the the E-SRGA possesses additional reconfiguration features known to be useful in the R-Mesh.

One addition in the E-SRGA architecture is the ability to operate the CSTs as segmentable buses; we use the implementation of Chapter 6. The E-SRGA also assigns each switch of a CST to a processing element (PE) in the row or column and enables the PE to control its switches directly. Specifically, each PE "owns" the row switch that succeeds it in the in-order traversal of the CST (see Figure 9.1). Consequently, each PE owns (at most) two switches, one each from its row and column CSTs. The architecture has been implemented in VHDL and we have conducted cost-benefit tradeoff for various dynamic reconfiguration features in the setting of an FPGA-like device. This study has shown our approach to be feasible. With algorithm design in mind, we have developed a programming model of the E-SRGA. This model abstracts away architectural details. This part of the research is work in progress.

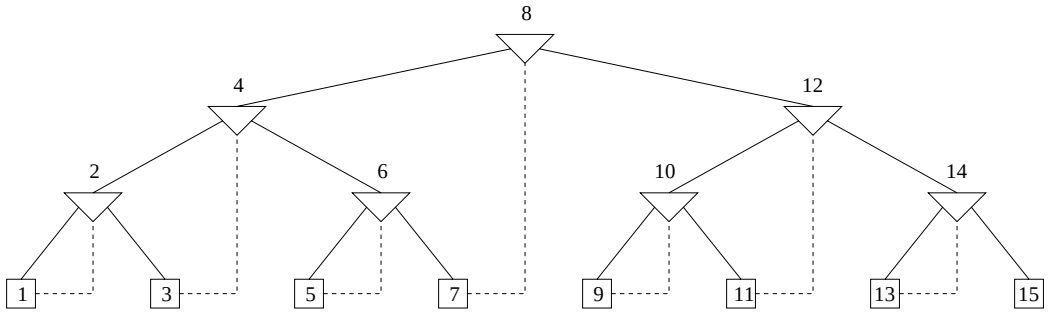


Figure 9.1: Associating CST switches with PEs. The CST nodes are numbered in inorder. Each switch has a dashed line to the PE associated with it

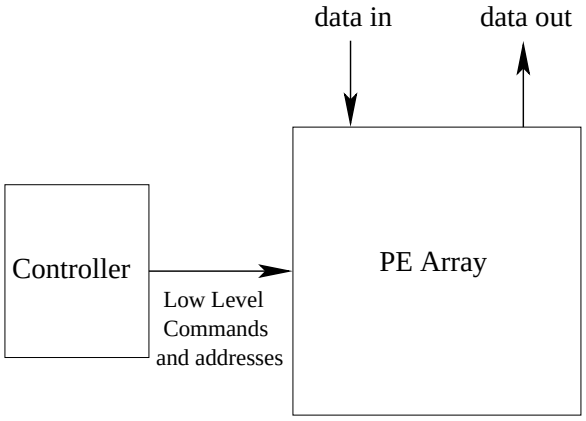


Figure 9.2: Overview of the E-SRGA architecture

The next section gives an overview for the E-SRGA architecture and Section 9.2 provides details. Section 9.3 describes our VHDL implementation of the E-SRGA and presents simulation results. Sections 9.4 and 9.5 describe the programming model of the E-SRGA.

9.1 Architecture Overview

The Enhanced Self-Reconfigurable Gate Array architecture (E-SRGA) consists of an array of processing elements (PEs) and an external controller that is responsible of issuing low-level commands to the PE array (see Figure 9.2). Each row and column of

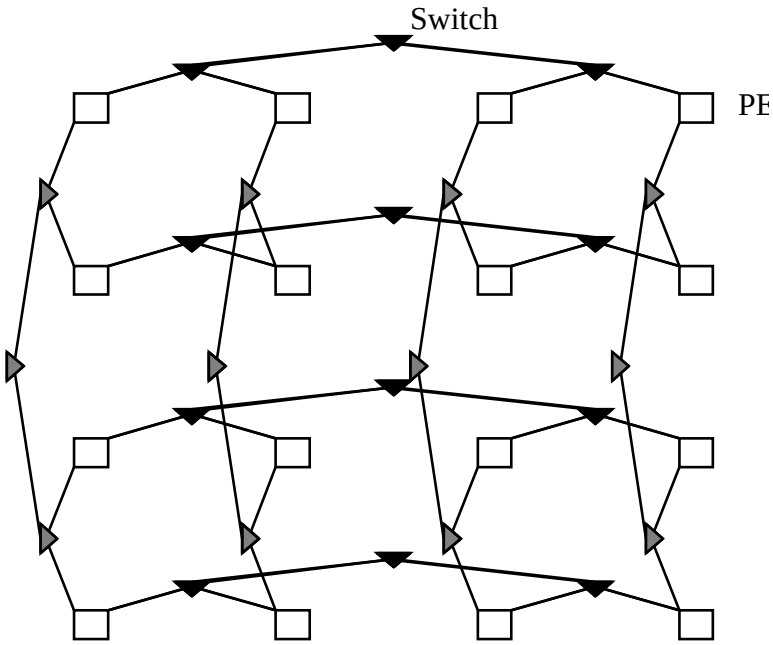


Figure 9.3: 4×4 PE array

the array is connected by a CST (see Section 2.1). That is, the basic interconnection structure is a binary tree whose leaves are PEs and whose internal nodes are switches. Figure 9.3 shows a 4×4 PE array. This array is suitable for VLSI implementation with the PEs laid out in a 2-dimensional mesh. The architecture also scales well as the collective areas of the CSTs grows logarithmically with the array size.

As described in Chapter 2, each switch of the CST has a full-duplex link to its parent (if any) and two children. Each switch is owned by a PE that can configure it to connect to its parent and children in various ways. Figure 2.2 (page 20) shows representative configurations that are assumed in this work. Some of these configurations are simple extensions of those used in the SRGA architecture to include broadcasting.

Each PE consists of a logic cell and a memory block. The logic cell contains a 20-bit look-up table (LUT) and 2 flip-flops, collectively capable of implementing many 2-input, 2-output Boolean functions. The memory block in a PE can hold data as well

as configuration contexts. Each configuration context contains bits that configure the logic cell and the two switches owned by the PE. This context can be changed by the PE in one clock cycle (as in the SRGA). That is, the functionality of the PE and the CST configuration can be changed in one clock cycle.

The controller is responsible for issuing commands to the PE array that specify the operations to be performed in the next clock cycle. To solve a problem on the E-SRGA a high level algorithm is first designed, which is then translated to a low level sequence of instructions understandable by the PE array. The controller is responsible of issuing these low level commands to the PE array. We describe the high level commands, and low level commands, and the correspondence between them in Section 9.5.4. In a typical implementation of an algorithm on the E-SRGA, the controller receives a high level command program, then PE array is loaded with an initial context set and data, if any. Next the controller issues low level commands causing the PE array to take the appropriate action. The important part to note is that the only run-time interaction between the PE array and the controller is through short commands requiring a few input pins (if the array is in a separate chip).

9.2 Architectural Details

In this section we describe the detailed architecture of each component of the PE array.

9.2.1 Interconnection Network

The interconnection network of the E-SRGA is the CST. That is, a binary tree whose leaves are PEs, and whose internal nodes are switches and edges are full duplex links (in which information can flow in both directions simultaneously). This interconnection network connects PEs in a row (or column). The actual connections between a PE could be either specified in a configuration context or it could result from a configuration operation (described in Chapter 5.) In Chapters 3–6, we showed this interconnection fabric to be capable of implementing a variety of communication sets, including those of a segmentable bus in at most 2 steps (clock cycles).

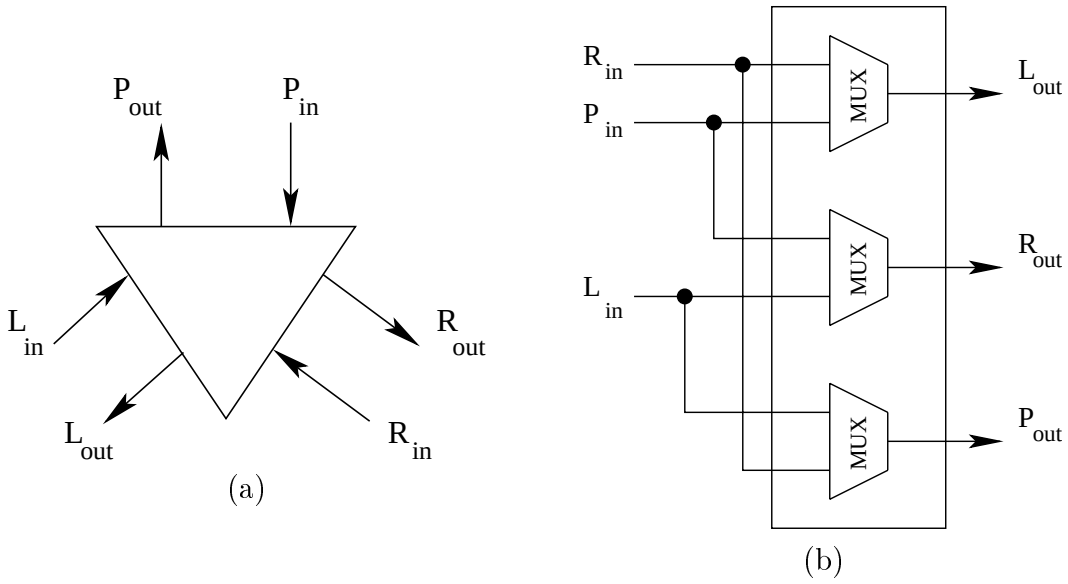


Figure 9.4: Structure of a CST switch

9.2.2 Switches

Each (three-sided) switch is an internal (non-leaf) node of the CST. It is connected to its parent (if any) and two children through a full duplex link (see Figure 9.4(a)). Each switch is owned by a PE that can configure it to connect to its parent and children in various ways (see Figure 2.2, page 20.) Observe that a switch cannot connect an incoming link to an outgoing link in the same "side" of the switch. This ensures that for a tree with N leaves (PEs), every communication will traverse no more than $2 \log N$ switches, where N is the number of PEs in a row. Each switch has 3 inputs and 3 outputs (an input/output pair per side). Each output can be connected to any of the 2 inputs on a different side of the switch via multiplexers (MUXes). To configure the switch, 3 bits are needed for the three MUXes (see Figure 9.4(b)).

The E-SRGA has the ability to internally generate configuration information for basic routing operations such as connecting two PEs. To generate this configuration information, however, the granularity of the E-SRGA PEs is somewhat larger than logic blocks of typical FPGAs. The configuration is performed so that the entire tree is configured in a single clock cycle. Chapter 5 discusses issues of configuring the

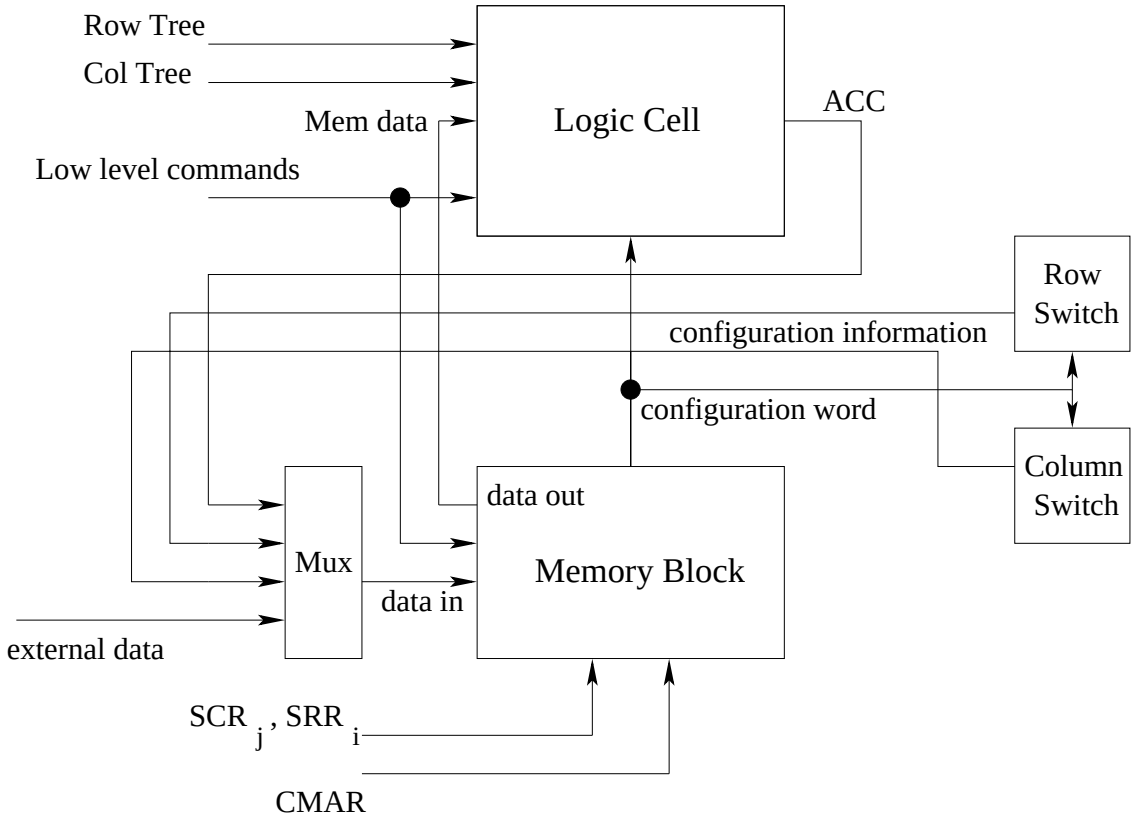


Figure 9.5: Structure of a PE

CST and the communication classes that can be accommodated on it. Each switch (or the associated PE) has a logic module that enables the configuration of the CST in accordance with certain communication classes. Each E-SRGA switch contains logic modules to handle edge-exclusive communication sets (see Section 5.2) and the communications of a segmentable bus (see Section 6). A switch also contain logic to enable the associated PE to directly configure the switch (without resorting to the techniques of Chapters 5 and 6).

9.2.3 Processing Elements

A block diagram of a PE is shown in Figure 9.5. Its main components are a logic cell and a memory block. (The two switches owned by the PE are also shown in the figure.) The memory block contains space for storing data and configuration

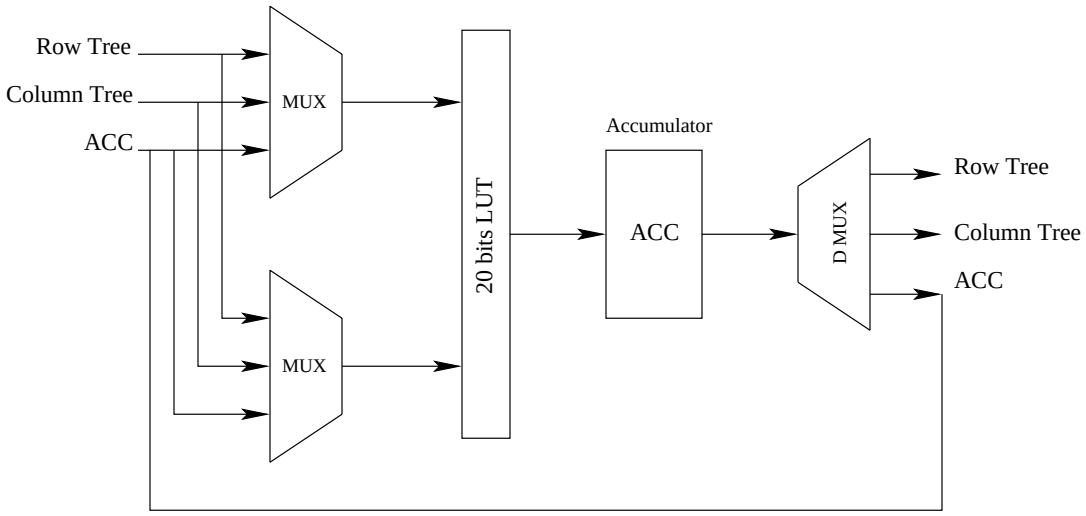


Figure 9.6: Logic cell structure

contexts. The logic cell can compute Boolean functions based on inputs from the row tree/column tree/memory bits. Depending on the low level command issued by the controller, the PE performs an operation on the local data or changes its configuration using a certain configuration word stored in the memory block. A PE receives control, data and address inputs. The control inputs are the low level commands from the controller. They determine which operation will be performed by the PE in the next clock cycle. The data inputs include data from row and column trees and external data (typically used only to load the initial data) that will be processed in the current cycle. The PE in row i and column j of the array also receives address inputs (SCR_j, SRR_i and CMAR) whose function is to enable/disable the PE or to address the memory. More details appear in Section 9.2.6. The memory block stores configuration words as well as acts as scratch pad memory. It is essentially like a regular RAM with the ability to access individual bits of each memory word.

9.2.4 Logic Cells

The structure of a logic cell is shown in Figure 9.6. One of the most important components of the logic cell is a 20-bits look-up table (LUT) that can implement a

Boolean function with 2 inputs, each 2 bits long. The LUT is actually two LUTs, one 16×1 bits and the other 4×1 bits. Because many useful functions have least significant outputs, depending only on least significant inputs, this arrangement works well. An accumulator (ACC) holds the logic cell output. The two data inputs to the logic cell can be chosen from the row tree, the column tree or from the previous output of the logic cell (ACC). The output of the logic cell could be directed to the row tree, the column tree or the accumulator.

9.2.5 Memory Block

The memory block has 8 words, each of width 46 bits as shown in Figure 9.7 (in general it could have n w-bit words). Each word contains a configuration for the PE and its two switches. The memory block could also be used as scratch pad memory. Thus an access to a single bit in a word is also allowed. The memory is addressed by a 9-bit address register, CMAR, the first three bits of which select a word of the memory, and the remaining 6 bits select a bit within the word. The data for the selected bit can come from the row switch, column switch, logic cell (ACC) or from outside the chip (external data).

Figure 9.8 shows the detailed format of the configuration word. The selection lines of the input and output MUXes and MUXes of the logic cell are set using the part of the bit labeled $M_0, M_1, \cdots, M_7$ in Figure 9.8. Bits 20-22 and 23-25 define the states of the row and column switches of the PE. Bits 26-41 and 42-45 specify the contents of 16×1 and 4×1 LUTs.

9.2.6 Registers

The E-SRGA contains several global and local registers. They are used to hold the low level instructions (or the decoded instruction) issued by the controller. The local registers are one per PE, whereas global register are shared by all PEs (see Figure 9.9).

1. Operation register (O-Register) and Assistant register (A-Register): These registers are 4 and 3 bits long (respectively). Together they act as the op-code for a low level command. The difference between them is that the O-Register is

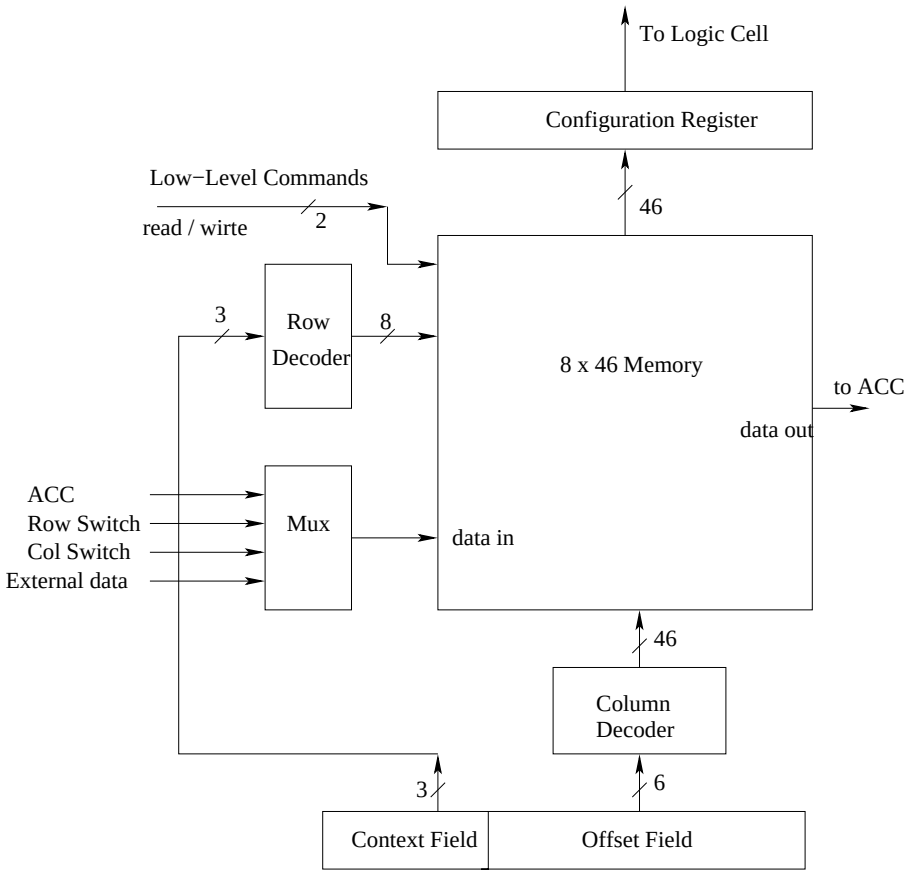


Figure 9.7: Memory Architecture

decoded outside the processor array chip, while the A-Register is decoded inside. Thus $2^4 + 3 = 19$ bits of op-code enter the processor array chip. Table 9.1 show the various low level commands and their corresponding op-code.

2. Qualify register (Q-Register), and Don't care Register (ϕ-Register): Assume the PE array to be of size $X \times Y$. The O-Register and the ϕ-Register (collectively called the Select or S-Registers) are in two sets, one for rows and the other for columns. The row S-Registers are each $\log X$ bits long. Together they define a subset of the X rows to select. One way to use these is as follows. The $2 \log X$ bits can be used to specify X^2 of the 2^X subsets of rows. These subsets can be programmed into the decoder for the row S-Registers. In the same way the

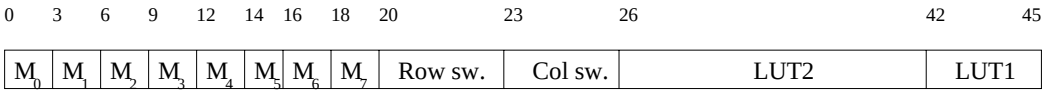


Figure 9.8: Details of a configuration word

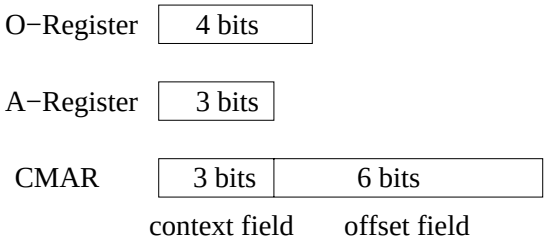


Figure 9.9: Global registers

column S-Registers is $2 \log Y$ bits long. After decoding the contents of these registers, SRR_i, SCR_j are used to store the decoded values (see Figure 9.10). A PE is enabled, iff its row and column are enabled. The N flags to select rows and columns are called the select row register (SRR) and select column register (SCR) (see Figure 9.10).

3. The Context and Memory Address Register (CMAR) has been considered in Section 9.2.3.

9.3 Implementation

The E-SRGA has been implemented in VHDL and synthesized using a 0.5 micron library of standard cells from AMI. The Leonardo Spectrum synthesis tool was used for the synthesis and optimization of the architecture. A C program was written to automate the implementation of E-SRGAs of different sizes. Within the memory restrictions on our server, we implemented and synthesized arrays of sizes 2×2, 4×4, and 8×8. Based on our measurements, an array size of $1 \times N$ appears to be a good predictor of an $N \times N$ array in terms of speed. In one dimension, we could implement arrays as large as 1×64. In our implementation we did not add the ability of a bus

Table 9.1: Low level commands of the E-SRGA; X denotes a dontcare, a * denote that these commands use addresses also provided by the control unit; †, these are explained in Section 9.5

O-Register	A-Register	Command Class	Function
0000	XXX	Continue	No change from current settings
0001	XX0 *	Memory access	write back configuration of row switch
	XX1 *		write back configuration of column switch
0010	XX0 *		write contents of ACC to memory
	XX1 *		read memory to ACC
0011	X00	Communication sets	configure as edge exclusive sets
	X01		configure as segmentable bus
0100	000	Direct switch control	set switch to left to right
	001		set switch to parent to left
	010		set switch to right to parent
	011		set switch to right to left
	100		set switch to left to parent
	101		set switch to parent to right
0101	X00	Set local FF	set to Zero
0101	X01		set to One
0101	X10		set to Zero flag
0101	X11		complement local FF
0110	XXX *	Switch context	Switch to another specified context
1000	X00 *	Set enable flag	Set enable flag type1 †
	X01 *		Set enable flag type2 †
	X10		Set enable flag on local FF
0111	X00	Set PE class	Set PEs as sources
	X01		Set PEs as segmenters
	X10		Set PEs as readers
1111	XXX *	Initial load	Load initial data

to bend between columns and rows and vice versa (as in the bends-cost LR-Mesh of Section 7.2), so that we could measure the clocking rate for a single row or column.

We implemented different versions of the architecture, each with different sets of features and compared the results of the simulation in terms of speed. We varied the size of the memory block of a PE to see its effect on the area. Our key findings based on the simulation results are as follows.

1. The clock rate is logarithmic in the array size (see Table 9.2 and Figure 9.11). In the figure, the horizontal axis is logarithmic in the array size. Clearly, this is due to the logarithmic diameter of the tree. The curve labeled "all features" represents the architecture with all the configuration features (segmentable buses,

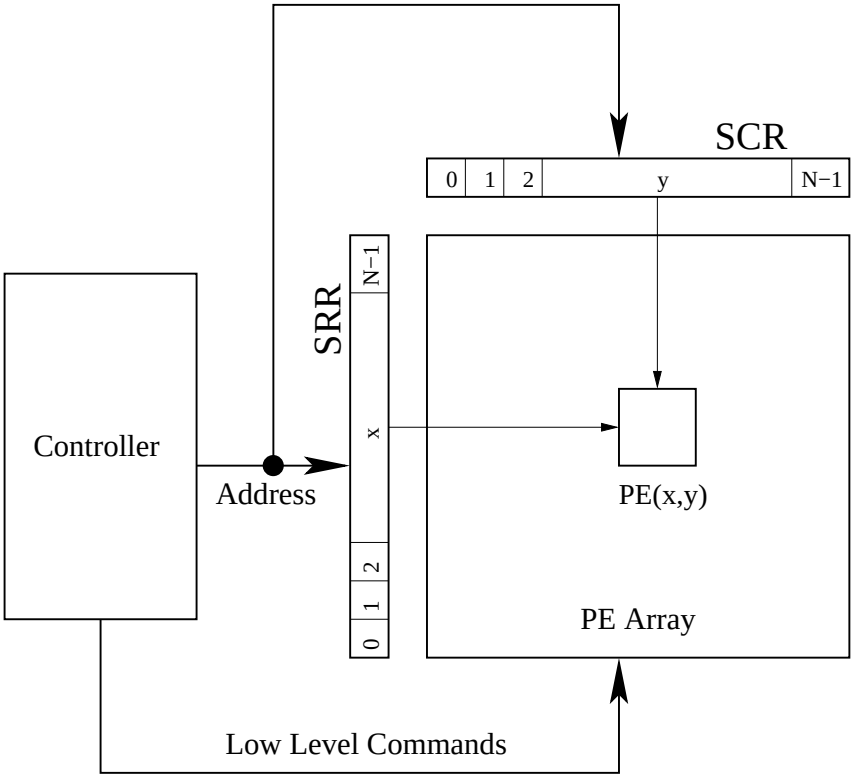


Figure 9.10: Interaction between controller and PE array

edge-exclusive sets, and direct control of the switches). The curve labeled "removing segmentable bus" represents the architecture with only the ability to implement edge-exclusive sets and direct switch control of PEs. The curve labeled "removing connect pairs" represents the architecture with only the ability to implement segmentable buses and direct switch control. The curve labeled "Architecture without configuration circuit" represents the architecture without support for any of these features. The logic required to implement the feature that enables PEs to set its switches directly has almost negligible cost.

2. The configuration hardware (needed to implement all configuration features) at a switch reduces the system clock considerably. For example, for 1×64 E-SRGA, the full-blown configuration hardware reduces the clock by almost 41%. The ability of implementing a segmentable bus only reduces the clock by

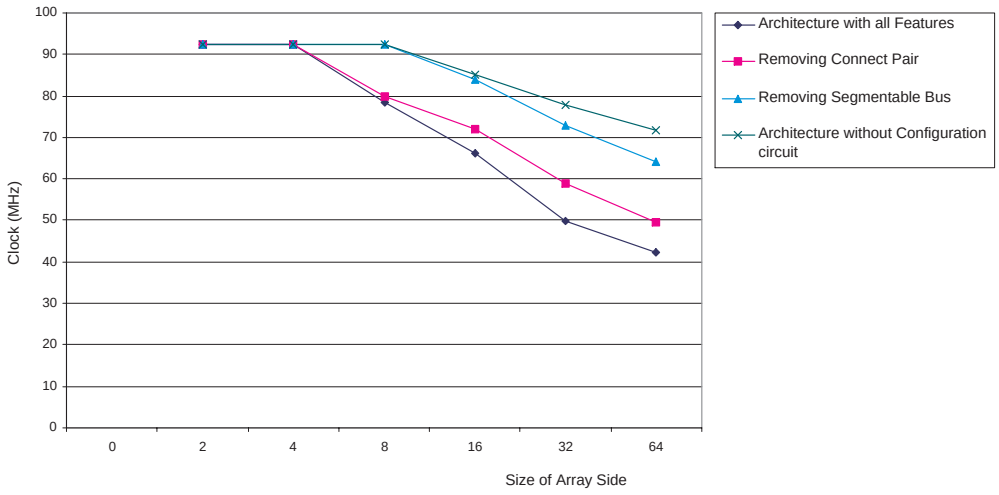


Figure 9.11: Effect of array size and different features on clock

about 30% (see Figure 9.11). The ability of implementing edge-exclusive sets only reduces the clock by about 10% (see Figure 9.11). Direct switch control by the PEs reduces the clock by about 1%.

3. The configuration hardware increases the switch area by a factor of about 3. This seems very costly in terms of the architecture area, however the area of the interconnection fabric including the switches (that have the configuration hardware) is still a very small factor (6%) of that of the the entire architecture.

4. The context memory size is the dominant factor for the area of the PE (see Figure 9.12 and Table 9.3). The figure shows that the area of the PE increases almost linearly with the number of words in the memory block. This means that if the number of words in the memory block is doubled, the area of the PE will almost double.

5. The interconnect area is about 6% only of the whole architecture (we removed all switches to establish this quantity.) In an FPGA-like device, the intercon-

Table 9.2: Effect of array size and afferent features on clock; the area is in number of gates, and the clock is in MHz

without Config. Hardware		with Connect Pair		with Seg. Bus		with Both	
Area	Clock	Area	Clock	Area	Clock	Area	Clock
9829	92.5	10159	92.5	10287	92.5	10387	92.5
19658	92.5	20318	92.5	20574	92.5	20701	92.5
39316	92.5	40637	92.5	41647	79.9	41548	78.5
78631	85	81274	83.9	82259	72.1	82805	66.3
157262	77.8	162548	72.8	164590	58.9	166192	49.9
314525	71.7	325095	64.2	329181	49.7	332383	42.3

nection fabric (the routing channels) occupies almost 80-90 % of the chip area. This contrast points to the E-SRGA having a better functional density than traditional FPGA. Actually this may be due to the way the E-SRGA architecture and FPGAs solve problems. FPGA-like devices actually build a circuit to solve the problem (hardware solution). On the other hand, the E-SRGA is programmed to solve the problem; a sequence of instructions is issued by an outside controller.

9.4 Modeling

Solving a problem on the E-SRGA would typically start with a high level algorithm design to the controller. Then the algorithm is translated to a sequence of low level instructions understandable by the E-SRGA architecture. Finally, the controller issues the low level commands sequence to the PE array. In this section, we abstract the architectural details of the E-SRGA and develop a programming model based on the architecture. This model could facilitate the design of algorithms without the need to know all architecture details. We specify the model of the E-SRGA in terms of some model parameters, connectivity, PE structure and capabilities and the interconnection fabric as described below. Other modeling approaches have been proposed before [7, 8, 9, 10, 11].

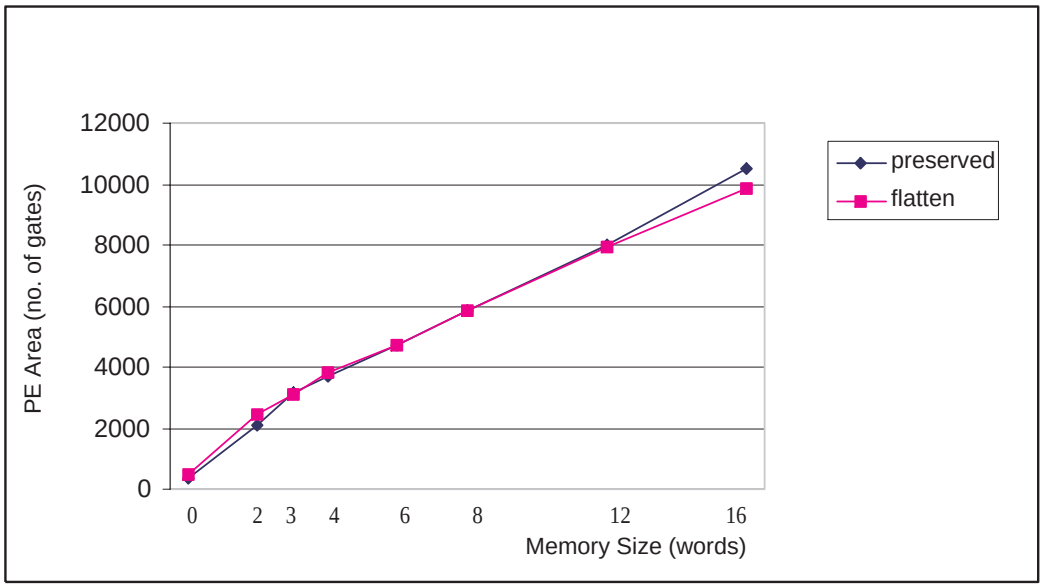


Figure 9.12: Effect of memory size on PE area for different optimization options

Model Parameters: The computational model has the following parameters:

- X, Y: Dimensions of the PE array (X is the number of rows, Y is the number of columns).
- P: Processing element word-size.
- PE_{area}: Area of the PE.
- $IF_{area}(N)$: Area of the interconnection fabric needed to connect N PEs in a row or column.
- C : Number of initially stored communication patterns (contexts).

Connectivity: Each PE is connected to the row/column interconnect through one full duplex link (or two half duplex links) which can be used to connect pairs of PEs (one-to-one communication) or broadcast data from one PE to several PEs in a row/column. PEs communicate with each other through P-bit wide communication links. The interconnect could be any network topology that is capable of implement-

Table 9.3: Effect of memory size on PE area

Number of Words	All Preserved		All Flattened		Memory Flattened	
	Area	Clock	Area	Clock	Area	Clock
0	365	181.4	500	370		
2	2076	89.4	2457	88.5	2524	92.5
3	3163	84.4	3089	81.6	3008	92.5
4	3715	83.6	3794	83.8	3458	92.5
6	4743	77.4	4691	81	4394	92.5
8	5827	73.7	5850	71	5273	92.5
12	7973	66.1	7942	67.7	7388	89.4
16	10498	77.3	9842	52	9328	89

ing a segmentable bus. However in this work we adopt the CST implementation of segmentable buses presented in Chapter 6.

PE Structure and Capabilities: The general structure of a PE is as follows.

- Each PE is connected to row/column CST.
- Each PE has a constant size memory to hold the stored communication patterns (contexts). These contexts could be changed during execution and restored again.
- Each PE has one P-bit accumulator (ACC) to hold the current result of computations or an initialization value. The PE can write the ACC contents into the memory.
- In one unit of time, a PE can perform any binary operation on two P-bit operands to produce a P-bit result. The operands could come from one of the following : ACC, Row CST, or Column CST.
- Each PE has an Enable flip flop. If set, the PE will participate in the current step.
- Each PE has a number of flags (such as a Zero flag) that reflect the status of the ACC.

Types of Communication: There are two types of communications.

1. PEs in a Row/Column can be connected in pairs (unicast or one-to-one communication).

2. One PE can broadcast data to several other PEs in the same row/column.

Both one-to-one communications and broadcasting take 1 unit of time. One unit of time is proportional to $\log N$, where N is the number of leaves of the tree.

The CST connects pair(s) of PEs in a row (or a column) such that all connected pairs in row/column satisfy topological limitations. Such limitations on communication on the CST is presented in Chapter 3.

9.5 Programming Model

The model is synchronous at the step level. At any time all PEs performing the same type of step (explained later) or idle. A step can be of three different types.

9.5.1 Com Step

A Com(municate/pute) step is a basic unit of computation or communication. This step always takes 1 unit of time. A PE receives two operands (from ACC, row CST, or column CST), performs a binary operation and stores the result in the ACC.

9.5.2 Sel Step

The Sel(ect) step selects (enables) a set of PEs for participation in the current step. Each row (or column) of PEs has a specific address of length $\log X$ (or $\log Y$) bits. The programmer can define the PEs to be enabled by providing the row and column addresses. Also, the programmer can enable a subset of rows or columns as explained in Section 9.2.6. There are three types of Sel steps. All of them run in 1 unit of time.

Type 1: Select PEs based on Row/Column The programmer defines a subset of rows and a subset of columns to be active. PEs at rows and columns are enabled.

Type 2: Select from already enabled PEs based on Row/Column This differs from a Type 1 Sel step only in that the enabled PEs are drawn from those that were enabled in the previous step. This allows a stepwise refinement of a subset of PEs enabled.

Type 3: Select PEs based on local flip flop The programmer selects the PEs to be enabled based on the contents of the local flip flops. The local flip flops can be set, reset, complemented or set to local data (based on the Zero latch for example) in a previous step.

9.5.3 Con step

The objective of the Con(figure) step is to make changes in the current CST settings so that different connections between PEs are established. This step takes at most 4 units of time. There are three types of Con steps.

Type 1: Connect Pairs (one-to-one) At each selected row/column, this step connects the same corresponding pairs of PEs. The programmer selects a source and a destination to be connected. This step allows an incremental change in the communication pattern. At the end of this step, the enabled pair of PEs at a Row/Column will be connected by a path from the source PE to the destination PE. By applying this step k times, k source-destination pairs could be connected in each tree.

Type 2: Connect PEs as Segmentable Bus The objective is to connect each row/column as a segmentable bus. PEs that are writers, segmenters, and readers have to be defined then the configuration is done as described in Chapter 6.

Type 3: Configure Switch Directly Each enabled PE sets its switches independently. Any of the above three types of steps can store the changes in the switch settings back into the configuration memory or it makes the changes only in the interconnect.

Table 9.4: Translation between high and low level commands

High Level Command	Equivalent Low Level Command(s)
Compute/Communicate	-Continue or -Switch Context -Continue
Select PEs based on Row/Column	-Set Enable flags type1
Select Already enabled PEs	-Set Enable flags type2
Select PEs based on local data	-Set local FF -Set Enable flags
Connect pair (c is the original context)	-Select PEs as sources -Select PEs as readers -connect edge-exclusive sets -Write row/column configuration bits (3 clock cycle) -Switch context (to context c)
Connect PEs as segmentable bus (c is the original context)	-Select PEs as sources -Select PEs as segmenters -Select PEs as readers -Connect as segmentable bus -Switch context (to context c)
Configure switches directly	-Direct switch control (at most 3 clock cycles) -Write ACC to Memory (3 clock cycles) -Switch context (to context c)

9.5.4 Relation between High and Low Level Commands

The purpose of the model presented in section 9.4 is to have a high level of abstraction for designing algorithms on the E-SRGA. However, for the algorithm to be actually executed on the E-SRGA, the high level commands have to be translated into low level commands that can be understood by the E-SRGA architecture. Table 9.4 shows the translation between the commands.

Table 9.5 establishes the time needed to run the high level commands on the E-SRGA.

9.6 Concluding Remarks

In this chapter we presented the E-SRGA architecture that has the ability to solve problems algorithmically. The E-SRGA has self reconfiguration ability where the configuration information for connecting pairs of PEs and connecting rows/columns

Table 9.5: Estimated time for high level commands

High Level Command	Estimated Time
compute/communicate (any type of operation, sending, or receiving data on already established path)	1 - 2 clock cycles
Select PEs based on Row/Column	1 clock cycle
Select Already enabled PEs based on Row/Column	1 clock cycle
Select PEs based on local data	2 clock cycles
Connect pair	7 clock cycles
Connect PEs as segmentable bus	8 clock cycles
Configure switches directly	7 clock cycles

as segmentable buses can be generated from within the chip. A cost-benefit trade-offs for the different dynamic reconfiguration features were obtained. Also, a high level abstraction (for designing algorithms) for the E-SRGA architecture has been developed that abstracts away some of the architectural details.

Chapter 10

Conclusions

Although a very powerful computing paradigm, dynamic reconfiguration has proved to be difficult to realize. This dissertation deals with different aspects of implementing dynamic reconfiguration. Chapters 3–6 dealt with an important communication structure called the CST. These chapters laid the foundation for developing primitive communication mechanisms used in subsequent chapters. The segmentable bus (Chapter 6) was used as a building block in an implementation of an LR-Mesh (Chapter 7). The idea of edge-exclusive communications was used in the E-SRGA architecture of Chapter 9. This work collectively addresses many facets of implementing dynamic reconfiguration, ranging from hardware details and low level architectures to modeling issues and high level algorithm design.

In Chapter 3 we analyzed the communication capability of the circuit switched tree (CST). We identified a property of a communication set, called width partitionability, that allows the communications to be scheduled efficiently on the CST. Then we showed three classes of communication sets to possess this property. As a special case of one of these results, we showed that the set of communications that can be performed in one step on a segmentable bus [48] can be scheduled in two steps on the CST.

In Chapter 4, we showed that any communication set that is not width partitionable has to satisfy a minimum set of requirements. We presented two "simplest sets" satisfying these minimum requirements and proved that these are the only two possible. We then showed that a communication set of width w could require as many as $\frac{5}{4}w$ steps to schedule on the CST. We also proved that, in general, non-

oriented, well-nested and non-oriented, monotonic communication sets are not width partitionable.

Chapter 5 presented a method to configure the full duplex CST to establish the communication paths of a one step communication set in one step. We applied our method to edge-exclusive communication sets. We showed that any one step communication set can be decomposed into at most three edge-exclusive sets and hence can be performed in at most three steps. Together with results of Chapters 3 and 4, this establishes a comprehensive method to perform communications on the CST.

In Chapter 6 we presented two approaches for implementing segmentable buses. The first is suitable for processors with large word-size using the CST. The second approach uses a binary tree algorithm and is better suited for small word-size processors.

Chapter 7 introduced the bends-cost measure of bus delay in linear reconfigurable meshes and showed this measure to be a faithful reflection of the actual bus delay in an implementation of the LR-Mesh called the bends-cost LR-Mesh. We also proved that an important class of LR-Mesh algorithms can be implemented using limited delay buses. In particular, we showed that it is possible to design constant time algorithms on reconfigurable models without resorting to the unit-cost assumption.

In Chapter 8 we proved that if polynomial delays are admissible, then the unit-cost LR-Mesh and the bends-cost LR-Mesh are equal in power. That is, for every T step algorithm on a unit-cost LR-Mesh, there is an $O(T)$ step algorithm on a bends-cost LR-Mesh.

Chapter 9 presented the E-SRGA architecture. This architecture aims to exploit the power of dynamic reconfiguration in an FPGA-like setting. We presented cost-benefit tradeoffs for different dynamic reconfiguration features and developed an algorithmic model for the architecture.

10.1 Future Directions

The work done in this dissertation has opened several other directions for future research. Here we organize these directions along the lines of the main topics of this

work, namely, (a) CST analysis, (b) CST configuration, (c) the bends cost measure, and (d) the E-SRGA.

CST Analysis

In Chapter 3 we have derived a lower bound on the number of steps for scheduling a set of one-to-one communications on the CST. We showed that this lower bound is tight for communication sets with disjoint incompatibles, oriented well-nested sets and oriented monotonic communication sets. The natural question is "are there other classes of communications for which this bound is tight as well?" In other words, are there other classes that are width partitionable? Can the methods developed be used for other communication structures (besides the CST)?

Chapter 4 characterizes the simplest communication sets that are not width partitionable. How does this characterization relate to a characterization of larger sets that are not width partitionable? Simply requiring a subset of a communication set to not be width partitionable is not sufficient for the entire set to not be width partitionable.

CST Configuration

In Chapter 5 we showed that the CST can accommodate any one-to-one communication set of width 1. For some communication sets (as in edge-exclusive sets and segmentable bus communications), the binary tree can be configured to establish the paths of these communications in a single step. The following questions arise: Are there other classes of one-to-one communications of width 1 for which the tree can be configured in a single step?

Also in Chapter 5 we presented an algorithm that decomposes any width-1 communication set into at most three edge-exclusive sets. This decomposition algorithm requires compile time knowledge of the communications and so cannot be used for run-time reconfiguration. Is it possible to perform this decomposition at run time?

Bends Cost Measure

In Chapter 7, we presented simulation algorithms for the unit-cost LR-Mesh on a bends-cost LR-Mesh with semimonotonic configurations. Are there other configurations that could be simulated in the same manner?

Is it possible to extend the bends-cost measure to other reconfigurable models, for example the unrestricted R-Mesh? Cyclic buses are an important issue here as they can cause a circuit with feedback or sequential circuits. Would these definitions change if a different technology, for instance, optical buses, were used?

All algorithms of Chapter 7 use a word-model, bends-cost LR-Mesh (as processors need to handle indices). Can these algorithms run on a bit model (in which processors cannot handle indices)? This requires the ability to transform the shapes of buses without using the indices of the end points.

Our result on the relative powers of the unit-cost LR-Mesh and bends-cost LR-Mesh hinges on the ability to tolerate polynomial delay. Can this condition be relaxed? Or conversely, is it possible to show that the $\Theta\left(\frac{\log N}{\log D - \log \Delta}\right)$ time overhead cannot be avoided?

A deterministic method was used to cut a bus to size in Chapter 7. Can randomization help, reduce the simulating model size? Randomization is possible only if one could assume a mechanism to flag buses that are too long. All this may require a change in the concept of power to include the cost of the bus delay.

E-SRGA

In Section 9.3 we have presented some of our simulation results for the E-SRGA architecture. Based on these, one direction would be to optimize the architecture to improve its clock rate and reduce the area. We showed that the configuration hardware contributes to lowering the clock rate. Also we observed the need for different hardware for each class of communication sets. Can we implement the configuration hardware in a manner that reduces its effect on the clock rate? One possible approach is to use configurable logic (possibly LUTs) to implement this hardware. Another approach is to use two different clocks in the architecture. The E-SRGA can operate

on one clock rate to configure the switches, while operating on a higher clock rate for other operations [37].

Since the number of contexts is a dominant factor for the area of the E-SRGA the following questions arise. How many contexts are needed for a function/algorithm? Can we reduce the width of each context so that the area is reduced? To load a new context, do we really need to load the whole context or we need only one part of it (such as LUT contents)? Answers to these questions will lead to better use of the chip area.

Another direction for the work on the E-SRGA is to implement a suite of primitive functions on the architecture and compare it to known hardware solutions that target FPGAs and ASICs. We expect that the implementation of these primitive functions (individually or collectively) on the E-SRGA (which uses dynamic reconfiguration) will have an advantage over the hardware implementation using FPGA and ASICs.

Bibliography

[1] J. A Anderson, *Discrete Mathematics with Combinatorics*, Prentice Hall, New Jersey, 2001.

[2] Atmel Corp., "AT6000 Series Configuration," configuration guide, 1997.

[3] Y. Ben-Asher, D. Gordon and A. Schuster "Efficient Self Simulation Algorithms for Reconfigurable Arrays," *J. Parallel & Distributed Computing*, vol. 30, 1995, pp. 1–22.

[4] Y. Ben-Asher, K.-J. Lange, D. Peleg and A. Schuster, "The Complexity of Reconfiguring Network Models," *Information and Computation*, vol. 121, 1995, pp. 41–58.

[5] Y. Ben-Asher, D. Peleg, R. Ramaswami and A. Schuster, "The Power of Reconfiguration," *J. Parallel & Distributed Computing*, vol. 13, 1991, pp. 139–153.

[6] A. A. Bertossi and A. Mei, "Optimal Segmented Scan and Simulation of Reconfigurable Architectures on Fixed Connection Networks," *Proc. 7th IEEE/ACM Int. Conf. on High Performance Computing (HiPC)*, 2000, pp. 51–60.

[7] K. Bondalapati, P. Diniz, P. Duncan, J. Granacki, M. Hall, R. Jain, and H. Zeigler, "DEFACTO: A Design Environment for Adaptive Computing Technology," *6th Reconfigurable Architectures Workshop*, Springer Verlag Lecture Notes in Computer Sc., vol. 1586, 1999, pp. 570–578.

[8] K .Bondalapati and V. K. Prasanna, "DRIVE: An Interpretive Simulation and Visualization Environment for Dynamically Reconfigurable Systems," *9th Int'l. Workshop Field-Programmable Logic and Applications*, Springer Verlag Lecture Notes in Computer Sc., vol. 1673, 1999, pp. 31–40.

[9] K. Bondalapati and V. K. Prasanna, "Hardware Object Selection for Mapping Loops onto Reconfigurable Architectures," *Proc. Int'l. Conf. Parallel and Distributed Processing Techniques and Applications*, 1999.

[10] K. Bondalapati and V. K. Prasanna, "Loop Pipelining and Optimization for Run Time Reconfiguration," Springer Verlag Lecture Notes in Computer Sc., vol. 1800, 2000, pp. 906–915.

[11] E. Caspi, M. Chu, R. Huang, J. Yeh, J. Wawrzynek and A. DeHon, "Stream Computations Organized for Reconfigurable Execution (SCORE)," *10th International Workshop Field-Programmable Logic and Applications*, Springer Verlag Lecture Notes in Computer Sc., vol. 1896, 2000, pp. 605–614.

[12] B. Beresford-Smith, O. Diessel, and H. ElGindy, "Optimal Algorithms for Constrained Reconfigurable Meshes," *J. Parallel & Distributed Computing* 1996, pp. 74–78.

[13] K. Compton and S. Hauk, "Reconfigurable Computing: A Survey of Systems and Software," *ACM Computing Suveys*, vol. 34, June 2002, No. 2, pp. 171–210.

[14] A. DeHon, R. Huang and J. Wawrzynek, "Hardware-Assisted Fast Routing," *Int. Symp. of Field-Programmable Custom Computing*, Napa, CA, April, 2002.

[15] H. P. Dharmasena, "Multiple-Bus Networks for Binary-Tree Algorithms," Ph.D. dissertation, Dept. of Electrical & Computer Eng., Louisiana State University, 2000.

[16] H. M. El-Boghdadi, R. Vaidyanathan, J. L. Trahan and S. Rai, "Implementing Prefix Sums and Multiple Addition Algorithms for the Reconfigurable Mesh on the Reconfigurable Tree Array," Proc. *Int. Conf. on Parallel and Distributed Processing Techniques and Applications*, vol. 3, 2002, pp. 1068–1074.

[17] J. A. Fernandez-Zepeda, R. Vaidyanathan, and J. L. Trahan, "Using Bus Linearization to Scale the Reconfigurable Mesh," *J. of Parallel & Distributed Computing*, vol. 62, 2002, 704, pp. 495–516.

[18] R. W. Hartenstein, M. Herz, T. Hoffman and U. Nageldinger, "On Reconfigurable Co-processing Units," *Reconfigurable Architectures Workshop*, 1998, Springer Verlag Lecture Notes in Computer Sc., vol. 1388, pp. 67–72.

[19] T. Hayashi, K. Nakano and S. Olariu, "Efficient List Ranking on the Reconfigurable Mesh, with Applications," *Theory of Compter Systems*, vol. 31, no. 5, 1999, pp 593–611.

[20] J. JáJá, *An Introduction to Parallel Algorithms,* Addison-Wesley Publishing Co., 1992.

[21] J. Jang, H. Park and V. K. Prasanna, "A Bit Model of Reconfigurable Mesh," *Proc. 1st Reconfigurable Architectures Workshop*, 1994.

[22] J. Jang and V. K. Prasanna, "An Optimal Sorting Algorithm on Reconfigurable Mesh," *J. Paralle & Distributed Computing*, vol. 25, no. 1, 1995, pp. 31–41.

[23] M. Kunde and K. Gurtzig, "Efficient Sorting and Routing on Reconfigurable Meshes Using Restricted Bus Length" *Int. Parallel Processing Symp.*, 1997.

[24] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays · Trees · Hypercubes*, Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[25] T. Leighton, "Tight Bounds on the Complexity of Parallel Sorting," *IEEE Trans. Computers*, vol. 34, 1985, pp. 344–354.

[26] C. E. Leiserson, "Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing," *IEEE Trans. on Computers*, vol. 34, 1985, pp. 892–901.

[27] H. Li and M. Maresca, "Polymorphic Torus Network," *IEEE Trans. Computers*, vol. 38, 1989, pp. 1345–1351.

[28] R. Lin, S. Olariu, J. L. Schwing, and B.-F. Wang, "The Mesh with Hybrid Buses: An Efficient VLSI Architecture for Digital Geometry," *IEEE Trans. on Parallel & Distributed Systems*, vol. 10, 1999, pp. 266–280.

[29] R. Lin and S. Olariu, "Reconfigurable Shift Switching Parallel Comparators, " *VLSI Design*, vol. 9, 1999, pp. 83–90.

[30] M. Maresca, "Polymorphic Processor Arrays," *IEEE Trans. Parallel & Distributed Systems*, vol. 4, no. 5, 1993, pp. 490–506.

[31] S. Matsumae and N. Tokura, "Simulation Algorithms among Enhanced Mesh Models," *IEICE Trans. Information & Systems*, Oct. 1999, vol. E82-D, no. 10, pp. 1324–1337.

[32] R. Miller, V. Prasanna-Kumar, D. Reisis and Q. Stout "Parallel Computing on Reconfigurable Meshes" *IEEE Trans. Computers*, vol. 42, no. 6, 1993, pp. 678–692.

[33] M. M. Murshed, "The Reconfigurable Mesh: Programming Model, Self-Simulation, Adaptability," Optimality and Applications," Ph.D. Dissertation, Australian National University, 1999.

[34] K. Nakano, "A Bibliography of Published Papers on Dynamically Reconfigurable Architectures," *Parallel Processing Letters*, vol. 5, 1995, pp. 111–124.

[35] K. Nakano and S. Olariu, "An Efficient Algorithm for Row Minima Computations on Basic Reconfigurable Meshes," *IEEE Trans. Parallel & Distributed Systems*, vol. 9, no. 6, 1998, pp. 561–569.

[36] S. Olariu, J. L. Schwing, and J. Zhang, "Fundamental Algorithms on Reconfigurable Meshes," *Proc. Allerton Conf. on Communication, Control & Computing*, 1991, pp. 811–820.

[37] "Reconfigurable Array Media Processor (RAMP)," *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, 2000, pp. 287–288.

[38] S. M. Scalera, J. J. Murray and S. Lease, "A Mathematical Benefit Analysis of Context Switching Reconfigurable Computing," *Reconfigurable Architectures Workshop*, 1998, Springer Verlag Lecture Notes in Computer Sc., vol. 1388, pp. 73–78.

[39] D. B. Shu and J. G. Nash, "The Gated Interconnection Network for Dynamic Programming," in *Concurrent Computations*, S. K. Tewksbury *et al.*, eds., Plenum Publishers, New York, 1988, pp. 645–658.

[40] R. Sidhu, A. Mei, and V. K. Prasanna, "Genetic Programming using Self-Reconfigurable FPGAs," *Int. Workshop on Field Programmable Logic and Applications*, Sept. 1999.

[41] R. Sidhu, A. Mei, and V. K. Prasanna, "String Matching on Multicontext FPGAs using Self-Reconfiguration," *Int. Symp. on Field-Programmable Gate Arrays*, Feb. 1999.

[42] R. Sidhu and V. K. Prasanna, "Efficient Metacomputation Using Self-Reconfiguration," *Proc. Field Programmable Logic* 2002, Springer Verlag Lecture Notes in Computer Sc., vol. 2438, 2002, pp. 698–709.

[43] R. Sidhu, S. Wadhwa, A. Mei, and V. K. Prasanna, "A Self-Reconfigurable Gate Array Architecture," *Int. Conf. on Field Programmable Logic and Applications*, 2000, Springer Verlag Lecture Notes in Computer Sc., vol. 1896, pp. 106–120.

[44] M. Slater, *Microprocessor Based Design-A Comprehensive Guide to Hardware Design*, Prentice Hall Inc., 1989.

[45] L. Snyder, "Introduction to the Configurable Highly Parallel Computer," *IEEE Computer*, vol. 15, 1982, pp. 47–56.

[46] "Tighter and Broader Complexity Results for Reconfigurable Models," *Parallel Processing Letters*, special issue on Bus-based Architectures, vol. 8, no. 3, pp. 271–282, 1998.

[47] "Constant Time Graph Algorithms on the Reconfigurable Multiple Bus Machine," *J. Parallel & Distributed Computing*, vol. 46, pp. 1–14, 1997.

[48] J. L. Trahan, R. Vaidyanathan and R. K. Thiruchelvan, "On the Power of Segmenting and Fusing Buses," *J. Parallel and Distributed Computing*, vol. 34, 1996, pp. 82–94.

[49] R. Vaidyanathan, C. R. P. Hartmann and P. K. Varshney, "Running ASCEND, DESCEND and PIPELINE Algorithms in Parallel Using Small Processors," *Information Processing Letters*, vol. 46, no. 1, pp. 31–36, April 1993.

[50] R. Vaidyanathan and A. Padmanabhan, "Bus-Based Networks for Fan-in and Uniform Hypercube Algorithms," *Parallel Computing*, vol. 21, 1995, pp. 1807–1821.

[51] J. F. Wakerly, *Digital Design , Principles & Practices*, Prentice Hall, New Jersey, 2001.

[52] M. J. Wirthlin and B. L. Hutchings, "DISC: The Dynamic Instruction Set Computer," *Proc. Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing*, J. Schewel, ed., *Proc. SPIE*, vol. 2607, 1995, pp. 92–103.

Vita

Hatem Mahmoud El-Boghdadi is a native of Egypt. He received his bachelor of science in electrical engineering (Computers and Control) in 1991 with grade of Distinction with honor degree, and master of science in electrical engineering in 1994, both from Assiut University, Egypt. Since 1992 he has been with the Electrical Engineering Department, Assiut University, as a demonstrator, and as an assistant lecturer in 1994. In 1998, he joined the Faculty of Computers and Informatics, Cairo University, Egypt, as an assistant lecturer. In the Fall of 1999, he joined the graduate program in the Department of Electrical and Computer Engineering at Louisiana State University, United States of America. He is expected to receive the degree of Doctor of Philosophy in electrical and computer engineering in May 2003.