TIME-MULTIPLEXED INTERCONNECTION NETWORK FOR FIELD-PROGRAMMABLE GATE ARRAYS

Xiaolei Chen

(*B. Sc. University of Science & Technology of China*)

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

NATIONAL UNIVERSITY OF SINGAPORE

February 2012

Acknowledgements

I would like to thank my supervisor Dr. Ha Yajun for his support and guidance. He dedicated much time to consistently building up my background and patiently leading me to the right direction in my research. Also, he consistently shows great confidence on me, so that I have been able to come all the way through these six years. I would like to thank Dr. Vaugh Betz, the author of the VPR tool and the book *Architecture and CAD for Deep-Submicron FPGAs*. I have never met him personally, and the only one interaction between him and me was an Email exchange. I "got to know" and learned from him through my experiences with the source codes and the book, which have had a profound effect on my PhD studies and my life. I would like to express my sincere thanks to Dr. Akash Kumar. He has been keeping an eye on my project and consistently giving constructive suggestions and comments for the past years. He also kindly invited me to attend his weekly network-on-chip (NoC) seminars in academic year 2010 - 2011. In this seminar series, I had the opportunity to regularly present my work and get helpful feedbacks and comments from a group of bright and enthusiastic fellow students.

I would like to thank Mr. Hanyu Liu. In the Fall of 2008, he quickly completed the first version of a global router for FPGAs with time-division multiplexed (TDM) interconnects. That global router, for the first time, enabled me to get a sense of

what the pros and cons of TDM interconnects are. Also I would like to thank Mr. Syed Rizwan, Mr. Loke Wei Ting, and Mr. Shakith Fernando. I have learned a lot from research discussions with them.

It has been a great pleasure to spend my PhD years at Signal Processing & VLSI Lab. I would like to acknowledge all in the Lab for your friendship. Thanks to Wei Ying, Tian Xiaohua, Li Yanhui, Yu Heng, Zhang Wenjuan, Shakith Fernando, Dong Bo, Pu Yu, He Lin, Chen Jiangzhong, Yang Zhenglin, Zhou Xiaodan, Tan Jun, Cheng Xiang, Zhang Xiaoyang, Wang Lei and Li Yong Fu. Since I take longer time to finish my PhD than most others, my thank-list is correspondingly longer than normal. My thanks also go to our dear lab officers: Mr. Teo Seow Miang and Ms. Zheng Huan Qun.

I would like to thank my parents and brother for their love and support. If humans had a second life, I wish that I could be their son and his brother for one more time. When I started on this project, I just got to know Ping. By today, I am almost convinced that she is *the one* with whom I want to spend the rest of my life. Thanks a lot for her accompany and love, which have proved to be crucial in the later stages of my PhD studies.

Last but not the least, my studies and research here are financially supported by National University of Singapore and Ministry of Education, Singapore. I would also like to thank Singapore people, numerous of which have displayed great friendship and kindness to me during the past six years.

Summary

In FPGAs, interconnects account for a large part of the area and timing budget. Given the significant intra- clock cycle idleness of wire segments in conventional architecture, we in this work propose TM-ARCH, a time-multiplexed architecture for FPGA interconnects. In this architecture, a wire can be multiplexed among multiple nets within one clock cycle. Specially designed time-multiplexing switches (TM switches) are used to enable the multiplexing of wires. Correspondingly, we present a time-multiplexing -aware timing-driven routing algorithm. Based on the VPR 5 timing-driven routing algorithm, this algorithm actively identifies nets that can be scheduled to multiplex wires. This routing algorithm accepts placement results from conventional placement tool, and requires no changes to the upstream EDA tools in FPGA design flow. This is the first timing-driven routing algorithm that performs combined global and detailed routing on FPGA architectures with time-multiplexed interconnects.

Our experiments with MCNC 20 benchmark show that, average minimum channel width required by the proposed TM-ARCH architecture is 20% less than that of conventional island-style architectures. Also, the average circuit critical path delay is 1.7% smaller. However, these improvements come at the expense of a 46% increase in routing area. As a result, TM-ARCH exhibits 10% larger area-delay

product compared with conventional island-style architectures. The area overhead is largely due to TM switches.

Our further investigation finds that partial depopulation can reduce the number of TM switches required, hence mitigate the area overhead. Correspondingly, we propose TM-ARCH(a), a family of FPGA architectures with partially populated time-multiplexed interconnects. This architecture family is extended from TM-ARCH architecture. In this architecture, only a portion of tracks in routing channels can be time-multiplexed with the aid of TM switches. We define an architecture parameter, a, to parameterize this portion. Our experimental results show that, TM-ARCH(a) architecture with small a values can achieve up to 10% smaller area-delay product than conventional island-style architectures.

This thesis demonstrates that the technique of time-multiplex can be applied to FPGA interconnects. Our proposed architectures show that time-multiplexed interconnect reduces channel width and improves circuit critical path delay. Our architectures also show that FPGA architectures with time-multiplexed interconnects can have area-delay product advantages, if area overhead of time-multiplexing is controlled appropriately.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Field-programmable gate arrays (FPGAs), as an important media to implement digital circuits, have been becoming increasingly popular. This trend is bound to continue, as the manufacturing technology of integrated circuits keeps on advancing. Although its programmability gives it some key advantages over application-specific integrated circuits (ASICs) technology, FPGA's programmability also causes significant timing, area, and power overhead compared with ASICs. If gaps between FPGAs and ASICs in these key metrics could be closer, FPGAs would be a more competitive technology.

It has been understood that FPGA interconnection network is a main contributor to the overall timing, area, and power budget. Hence, this thesis focuses on FPGA interconnection network exclusively. Traditionally, three factors are identified as determining the performance of an FPGA at large: quality of the FPGA architecture, quality of the computer-aided design (CAD) tools, and electrical design of the FPGA. This thesis attempts to examine and optimize the performance of FPGA interconnection network mainly by looking at the former two factors, i.e.,

architecture and CAD.

The technique of time-multiplex has been applied to FPGAs before. Originally this technique was applied to FPGA logic blocks so as to improve utilization of logic blocks. This is reasonable because logic resources used to be at a premium in early years of FPGAs. Given that nowadays FPGA interconnect resources are more costly than the logic, this thesis proposes that time-multiplex be applied to FPGA interconnects; hence the title of this thesis.

In this thesis, we present TM-ARCH, an FPGA architecture with fully populated time-multiplexed interconnects. This architecture is based on the classical island-style architecture [6]. All wires in routing channels can be time-multiplexed with the aid of specially designed switches. We also present TM-ARCH(a), a family of FPGA architectures with partially populated time-multiplexed interconnects. TM-ARCH(a) is extended from TM-ARCH architecture, and the parameter a is used to parameterize the portion of routing tracks that can be time-multiplexed. At CAD side, we present a time-multiplexing -aware timing-driven routing algorithm, which is based on VPR 5 timing-driven routing algorithm [20]. We implement this algorithm as our routing tool. With this routing tool, we employ a standard CAD flow and a set of benchmark circuits to experimentally evaluate TM-ARCH and TM-ARCH(a) architecture.

This thesis demonstrates that the technique of time-multiplex can be applied to FPGA interconnects. Our proposed architectures show that time-multiplexed interconnect reduces channel width and improves circuit critical path delay. Our architectures also show that FPGA architectures with time-multiplexed interconnects can have area-delay product advantages, if area overhead of time-multiplexing is controlled appropriately.

The remainder of this chapter is organized as follows. Section 1.1 and 1.2 give an overview of FPGAs at large and FPGA interconnection network, respectively. Section 1.3 proposes the use of time-multiplex technique for FPGA interconnects. Section 1.4 briefly describes our approaches and some key results. Section 1.5 and 1.6 describes contributions and organizations of this thesis, respectively.

1.1 Overview of FPGAs

Since their introduction in 1980s, field-programmable gate arrays (FPGAs) have gained wide popularity. Some of the historical key application fields of FPGAs include rapid prototyping, custom computing, and design emulation. Besides, with the technology scaling enabling high degree of integration, FPGAs nowadays contain rich logic, as well as a variety of customized functional blocks, such as block memory, digital signal processing (DSP) blocks, and Ethernet controller. Hence, FPGA has also become an important medium to implement user designs.

Compared with application-specific integrated circuits (ASICs), FPGAs have two desirable qualities: low non-recurring engineering (NRE) cost and fast turn-around time. While the IC technology is progressing towards 28nm technology node, it is expected that the cost and complexity of ASICs will skyrocket. This will make FPGA implementation more attractive for IC designers working on low-to-middle volume productions.

The above-mentioned advantages of FPGAs come with a price. It has been well known that, for the same user design, the FPGA implementation will consume more area and power, yet achieve lower performance than the ASIC implementation. A recent work [11] which measures the gap between FPGA implementation

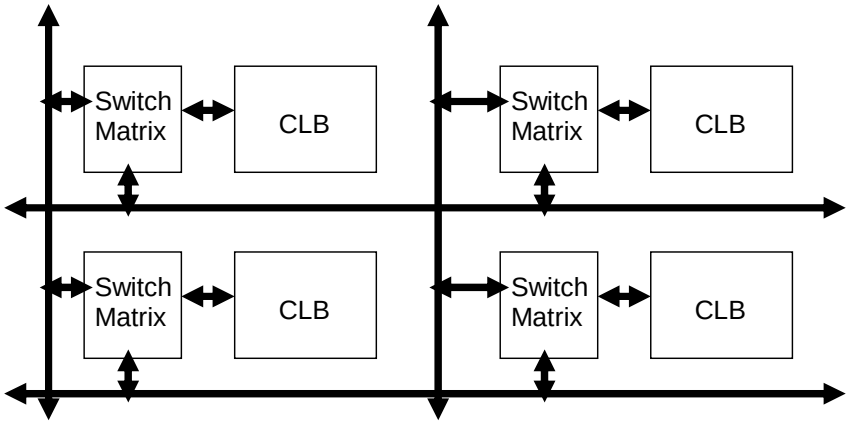


Fig. 1.1: A generic FPGA architecture (from [37])

and ASIC implementation points out that the required silicon area of the FPGA implementation is about 18 times that of ASIC implementation, the critical path delay about three to four times, and the dynamic power consumption about 14 times.

1.2 FPGA Interconnection Network

Resources in an FPGA device can be broadly divided into two parts: logic resources and interconnect resources. Logic resources are used to implement user logic. Logic resources in FPGAs are usually organized as arrays (or columns) of logic blocks. Interconnect resources are used to implement connections between different logic blocks. Interconnect resources usually include routing wires, grouped into channels, and routing switches. Figure 1.1 illustrates a generic FPGA architecture. FPGA interconnects (or interconnection network) can be defined as the programmable network of signal pathways between inputs and outputs of logic blocks within FPGA.

Both analysis and measurements reveal that interconnection network is the main contributor to area, delay, and power. To achieve high routability leading to successful implementation of user design, and also to ease routing task for CAD tools leading to reduced compile time, FPGA vendors usually devote much silicon area to routing tracks and programmable switches used to connect tracks. This explains why interconnects out-weight logic in terms of area and power budget. Routing a net to connect pins at different logic blocks usually means traversing a series of tracks connected by switches. This explains why interconnects in FPGA implementation are slow. Given that the interconnection network has a profound effect on FPGA's overall performance, optimizing the interconnection network is essential.

1.3 Time-Multiplexed Interconnects for FPGAs

Previous research [35] has shown that time-multiplexing can improve the utilization of logic resources in FPGA. In our work, we apply time-multiplexing to interconnection networks in FPGA. The idea is based on the observation that most interconnect wires are only used for a short period in a clock cycle. That is, the delay for a signal to propagate along a wire segment is only a small portion of the clock cycle. Take the 65nm FPGA architecture file *n10k04l04.fc15.area1delay1.cmos65nm.bptm* from iFAR [23]. The delay of a wire spanning four logic blocks (plus the delay of the switch driving the wire) is around $70ps$. Assuming this same architecture file, the average post-route critical path delay of MCNC 20 benchmark circuits is around $4ns$. Dividing $70ps$ by $4ns$ gives a percentage of 1.8%. This means that, in a clock cycle, a wire is effectively used to propagate a signal in only 1.8% of the time. In

other words, a wire remains idle for more than 90% of the time. An intuitive way to understand this fact is that the circuit critical path usually spans a number of logic blocks and nets. If we examine one wire segment used along this critical path, we may find that its delay is far less than the whole critical path delay.

By time-multiplexing signals on interconnects, we can better utilize the interconnect resources. This could translate to area savings, as well as performance improvement, at the cost of routing circuitry complexity. As the FPGA architecture keeps evolving, time-multiplexed interconnection network could be a viable solution to the scalable FPGA architecture.

1.4 Approaches and Key Results

In this thesis, we first present TM-ARCH, an FPGA architecture with fully populated time-multiplexed interconnects. This architecture is based on the classical island-style architecture [6]. All wires in routing channels can be time-multiplexed. Specially designed switches, time-multiplexing switches (TM switches), replace conventional switches to enable time-multiplexing of wires. Following Trimberger *et al*'s terminology, we define the architecture parameter, K, as the number of micro-cycles in a user clock cycle. That is, the number of time slots for time-multiplexing. It is worth to mention here that this architecture does not time-multiplex logic blocks.

We then present a time-multiplexing -aware timing-driven routing algorithm. This routing algorithm is based on VPR 5 timing-driven routing algorithm [20]. It employs a multiplexing-aware congestion cost function so as to identify nets for time-multiplexing. We implement this algorithm as our routing tool. By assuming

standard FPGA CAD flow and replacing the conventional router with our router, we are able to implement circuits onto our proposed TM-ARCH FPGAs. More importantly we are able to evaluate experimentally TM-ARCH architecture by mapping a set of benchmark circuits to the architecture and measuring area and timing results.

In our evaluation of TM-ARCH, we use MCNC benchmark circuits and VPR flow, which are the common practices in the research community. We compare TM-ARCH with the conventional island-style architecture based on four metrics: minimum channel width, routing area, circuit critical path delay, and area-delay product. Our evaluation shows that TM-ARCH generally can achieve smaller minimum channel widths. For example, with K=4, TM-ARCH achieves average 20% reduction in minimum channel widths over 20 MCNC circuits. But TM-ARCH is also shown to exhibit significant routing area overhead although it reduces channel widths. In the case of K=4, TM-ARCH requires 46% larger routing area. This significant area overhead is mainly due to TM switches, which consume much more area than their conventional counterpart. Our evaluation shows that TM-ARCH can achieve similar or slightly better critical path delays with smaller channel widths than the conventional island-style architecture. In the case of K=4, TM-ARCH improves critical path delay by 1.7% while using 20% smaller channel widths. Finally, our evaluation shows that TM-ARCH generally has larger area-delay product. With K=4, TM-ARCH exhibits 10% larger area-delay product.

Our evaluation of TM-ARCH architecture reveals that area overhead of TM switches can be significant. This motivates us to propose a family of FPGA architectures with partially populated time-multiplexed interconnects. This architecture family is extended from TM-ARCH architecture. In this architecture, only a portion

of tracks in routing channels can be time-multiplexed with the aid of TM switches. We define a second architecture parameter, a, to parameterize this portion. We denote this family of architecture by TM-ARCH(a). With a=1.0, TM-ARCH(a) is equivalent to TM-ARCH.

Again, we compare TM-ARCH(a) with the conventional island-style architecture. Our evaluation shows that TM-ARCH(a) architecture with small a values can achieve smaller area-delay product than the conventional island-style architecture. For example, with K=4 and a=0.1, TM-ARCH(a) achieves 10% smaller area-delay product.

1.5 Contributions of This Thesis

This thesis investigates two issues related to employing time-multiplexed interconnects for FPGAs. First, it investigates the proper FPGA interconnect architectures which support time-multiplexing. Second, it investigates the corresponding routing algorithm which schedules signals to achieve time-multiplexing. Main contributions of this thesis are:

1. An FPGA architecture with fully populated time-multiplexed interconnects is presented. This architecture is based on VPR island-style architecture. All the wires in the routing channels can be time-multiplexed with the aid of specially designed switches, TM switches. This architecture differs from existing FPGA architectures with time-multiplexed interconnects mainly in two aspects. First, only interconnect resources can be multiplexed, and logic blocks cannot be multiplexed (compared with [35]); Second, TM switches provide signal latching capability (compared with [18] and [10]). We believe that our work is the first to

examine the feasibility of applying time-multiplexed interconnects to an academic island-style FPGA architecture.

This architecture is evaluated against and compared with island-style architecture experimentally. Our experimental results show that this architectural generally can achieve smaller channel widths. This acknowledges findings of related work in literature ([18] [10]). As part of the architecture evaluation, an important architectural parameter K, number of microcycles in a user clock cycle, is investigated. The investigation shows that $K = 2$ and $K >= 8$, exhibiting significant area overhead, are inappropriate. This finding, to some extent, challenges practices in related work, which either use $K = 2$ ([18]) or $K >= 8$ ([35] [10]). The architecture evaluation also reveals that time-multiplexed interconnect could achieve comparable or slightly better timing *and* use smaller channel widths than its conventional counterpart. This finding is the first to demonstrate that, in terms of circuit timing, time-multiplexed interconnect is competitive with its conventional counterpart.

2. A time-multiplexing -ware timing-driven routing algorithm is presented. This algorithm is based on VPR 5 timing-driven routing algorithm. Multiplexing-aware congestion cost function is used so as to identify signals for time-multiplexing. This routing algorithm is implemented as a routing tool, which is used for experimental evaluation of our proposed time-multiplexed interconnect architectures stated in contribution 1 and 3. This is the first timing-driven routing algorithm that performs combined global and detailed routing on FPGA architectures with time-multiplexed interconnects. This algorithm is important for us to demonstrate that time-multiplexed interconnect is competitive with its conventional counterpart in terms of circuit timing.

3. A family of FPGA architectures with partially populated time-multiplexed interconnects is presented. This architecture is also based on VPR island-style architecture, and extended from our proposed architecture stated in contribution 1. A portion of tracks in the routing channels can be time-multiplexed with the aid of TM switches, while the remaining tracks cannot be multiplexed. An architectural parameter a, multiplex-able track population, is used to parameterize this portion. This family of architectures is evaluated against and compared with island-style architecture experimentally. The experimental results demonstrate that small values for a help this architecture achieve good tradeoff between channel width reduction and area overhead. This finding, to some extent, acknowledges a previous finding in the literature ([10]). The results also show that our proposed architecture with small a values could achieve comparable or better area-delay product than the conventional island-style FPGA architecture. This finding is important, for it demonstrates that time-multiplexed interconnect is practical for FPGAs.

1.6 Organization of This Thesis

The remainder of this thesis is organized as follows. The next chapter provides background information and reviews some of the previous work in the area of FPGA interconnect architecture. Chapter 3 presents our proposed FPGA architecture with time-multiplexed interconnects. Chapter 4 presents our time-multiplexing - aware timing-driven routing algorithm. Architecture and routing algorithm are the two facets of our work. Then the proposed architecture is evaluated, and experimental results are given in Chapter 5. Chapter 6 presents the proposed FPGA architecture with partially populated time-multiplexed interconnects. In this same

chapter, we evaluate this partially populated architecture and give experimental results. Chapter 7 concludes this thesis and provides suggestions for future work.

Chapter 2

Background and Previous Work

The first half of this chapter gives background information about island-style FPGA architecture and VPR routing tool. The second half of this chapter reviews some of the previous research on FPGA interconnect architecture.

2.1 Island-Style FPGA Architecture

In their classical book *Architecture and CAD for Deep-Submicron FPGAs* [6], Betz *et al* classified FPGAs into three groups according to their routing architecture: island-style, row-based, and hierarchical. Lemieux and Lewis largely followed this classification in their book *Design of Interconnection Networks for Programmable Logic* [15]. The island-style architecture is certainly the most popular one. Almost all modern commercial FPGAs employ the island-style architecture. And in academia, research and literature on the island-style FPGA architecture also dominate.

In our work, we propose a new FPGA architecture by extending the classical

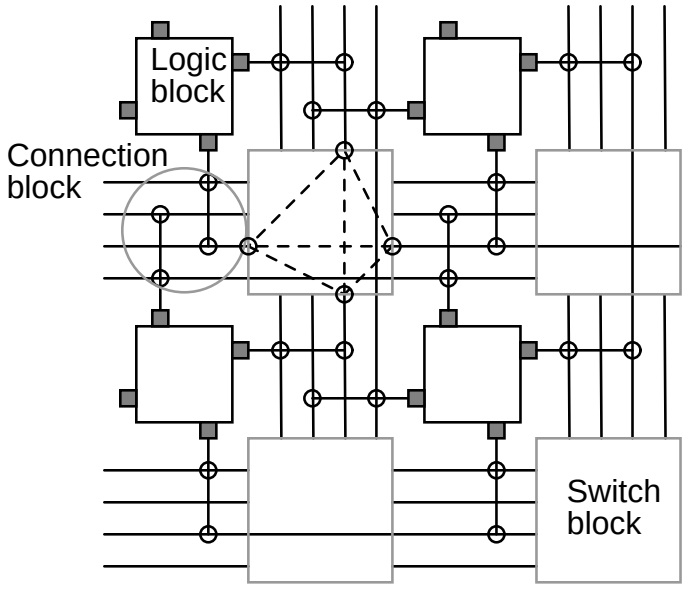


Fig. 2.1: An island-style FPGA (from [6]).

island-style FPGA architecture, as defined in Betz *et al*'s book. Hence, in the following, we give a short summary of key features of the island-style FPGA architecture. Readers are referred to Betz *et al*'s book for a detailed treatment of the island-style architecture.

Island-style architecture is also called mesh architecture in Lemieux's book. In island-style FPGAs, logic resources are organized as a two dimensional array of logic blocks. Logic blocks are surrounded by routing channels on four sides. A routing channel, either horizontal or vertical, usually contains a number of routing tracks. A routing track usually consists of a series of wire segments. Figure 2.1 illustrates an island-style FPGA.

Input and output pins of a logic block can connect to the peripheral routing channels via connection blocks. And at the intersection of a horizontal channel and a vertical channel, there is a switch block. Both connection blocks and switch

blocks are made up of routing switches. A routing switch contains a one-bit memory cell, and can be turned on or off by appropriately configuring the memory cell. It is the routing switches inside connection blocks and switch blocks that achieve programmable interconnections between logic blocks.

Logic blocks themselves are also programmable, thus allowing users to implement different logic functions. A logic block usually contains a cluster of look-up tables (LUTs) and registers. A k-input LUT can implement any Boolean function with k inputs and one output. Registers enable the implementation of sequential logic.

2.1.1 Uni-Directional and Single Driver Wiring

Major commercial FPGAvendors have shifted away from using bi-directional and multiple-driver wires. They use uni-directional and single-driver wires, instead. Figure 2.2 and Figure 2.3 illustrate these two paradigms, respectively. In Figure 2.2, the horizontal wire w_1 spanning two logic blocks can route a signal either from left to right or from right to left. Correspondingly, the switches used are bi-directional. A pass transistor switch connects wire w_1 with another horizontal wire w_2, and two back-to-back tri-state buffers connect wire w_1 with a vertical wire w_3. Also notice that wire w_1 can be driven from multiple points: it can be driven via the tri-state buffer b_3 by wire w_3, or via tri-state buffer b_1 by a logic block, or via tri-state buffer b_2 by a second logic block. But in Figure 2.3, the horizontal wire w_1 can route a signal from left to right only. And it can only be driven via the buffer b_1 from the left endpoint. A wide multiplexer is used to select from all possible sources.

It has been shown in [14] that uni-directional and single-driver wiring reduces

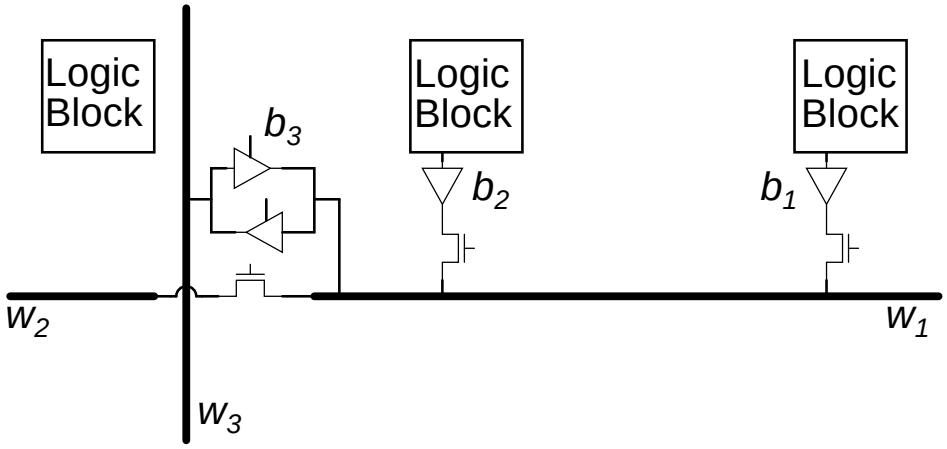


Fig. 2.2: Bi-directional wires with multiple drivers (adapted from [19]).

area and improves delay over bi-directional and multiple-driver wiring. This may explain the trend towards uni-directional and single-driver wiring in the industry.

2.1.2 Direct Drive Mux Switch

In the context of uni-directional and single-driver wiring, one switch widely used is direct drive mux switch [17].

This type of switch consists of two parts: a multiplexer used to select input lines and a buffer to drive the wire. The multiplexer usually is constructed using NMOS pass transistors. As a result, a logic-1 signal at the selected input line will produce a weak-1 signal at the output of the multiplexer. A level-restoring PMOS transistor is integrated with the buffer to restore the voltage level of the weak-1 signal [25] [4]. Figure 2.4 illustrates a direct drive mux switch.

Assuming that we only use NMOS pass transistors to construct the multiplexer, there can be three kinds of topology: tree, flat, and two-level hybrid [12]. A comparison of these three topologies can be found at [12]. The two-level hybrid

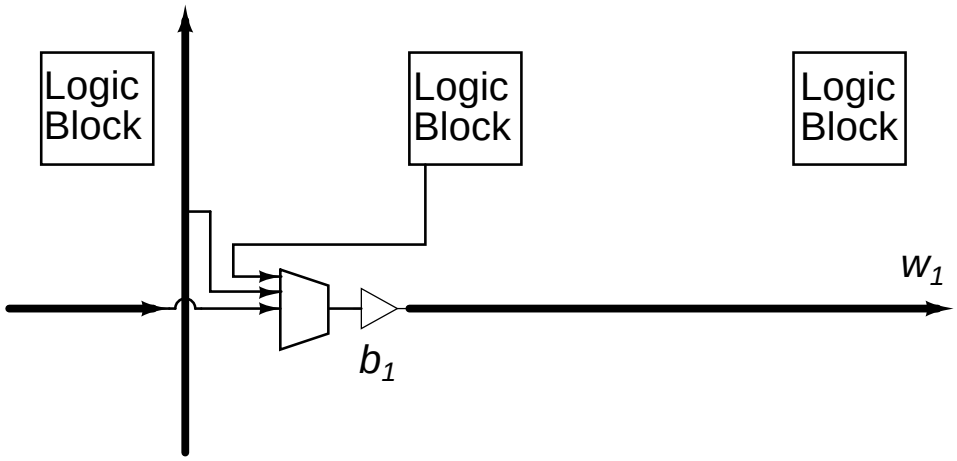


Fig. 2.3: Uni-directional and single-driver wires (adapted from [19]).

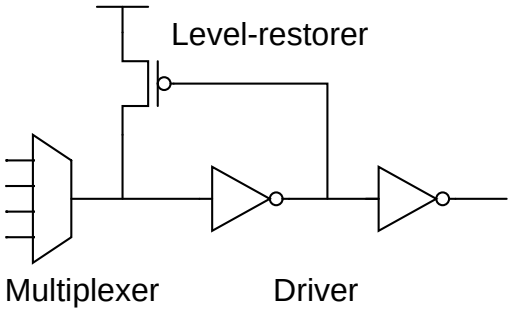


Fig. 2.4: A direct drive mux switch.

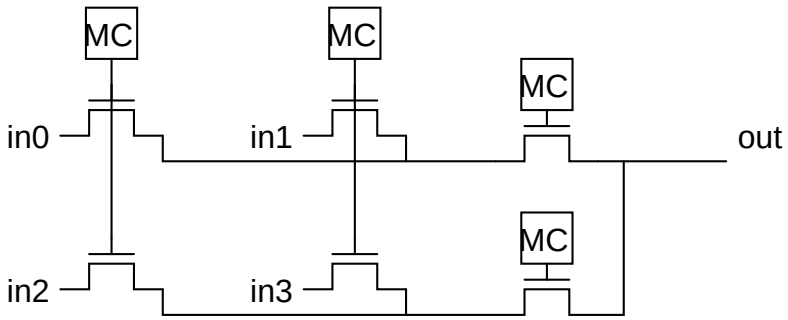


Fig. 2.5: A 4:1 multiplexer assuming the two-level hybrid topology. Each "MC" represents one-bit memory cell.

topology, which strikes a good balance between area and performance, is employed in [25] [4] [12] [16]. We also use the two-level hybrid topology in our work. Figure 2.5 shows a transistor-level implementation of a 4:1 multiplexer, which assumes the two-level hybrid topology.

2.2 VPR Router

VPR (Versatile Pace and Route) CAD tool [5] was developed by the FPGA research group at University of Toronto. It has been well maintained and regularly updated by the same research group. Since its release in 1997, it has become the standard tool among the FPGA researchers, due to its free availability and high quality. In our work, we mainly use VPR version 5.0 [20], which we will refer to as "VPR 5" hereafter.

VPR 5 includes a timing-driven router, which is based on the Pathfinder negotiated congestion-delay algorithm [22]. Section 2.2.1 briefly introduces Pathfinder algorithm. In our work, we make extensive use of VPR 5 timing-driven router's capabilities of delay modeling and static timing analysis. Section 2.2.2 briefly describes how VPR router models the delay of a routing path. Section 2.2.3 gives background knowledge on static timing analysis.

2.2.1 Pathfinder Routing Algorithm

Pathfinder negotiated congestion-delay routing algorithm is based on an iterative approach [22]. It rips-up and re-routes each net in each iteration. During each iteration, it finds the shortest path for each sink s_{ij} of each net N_i using directed search, and allows resource overuse. Figure 2.6 presents pseudo-code of Pathfinder

routing algorithm.

```
     Q(i): Priority queue used while routing net N_i
     RT(i): Routing tree of net N_i
     Source(i): Source of net N_i

1    Crit(i,j) = 1.0 for all i and j;
2    while (overused resources exist) {
3      for (each net N_i) {
4        Rip-up RT(i), and update p(n) for all nodes n in RT(i);
5        RT(i) = Source(i);
6        for (each sink j of net N_i in decreasing Crit(i,j) order) {
7          Q(i) = RT(i);
8          while (sink(i,j) not found) { /* Wave expansion */
9            Remove lowest cost node, m, from Q(i);
10           for (all fanout nodes n of node m) {
11             Add n to the Q(i);
12           }
13         } /* Routing of one sink is finished. */
14         for (all nodes, n, in path from RT(i) to sink(i,j)) {
15           Update p(n);
16           Add n to RT(i);
17         }
18       } /* Routing of one net is finished. */
19     } /* Routing of all nets are finished. */
20     Update h(n) for all nodes n;
21     Update Crit(i,j);
22   } /* End of one routing iteration*/
```

Fig. 2.6: Pseudo-code of Pathfinder routing algorithm. From [6]

A key innovation of Pathfinder algorithm is its cost function. By using a cost function as shown by Eq. 2.1, it gradually resolves the overuse after multiple iterations, and at the same time optimizes the delay. The cost to include a node n into the routing path is

$$c(n) = Crit(i,j) \cdot d(n) + [1 - Crit(i,j)] \cdot b(n) \cdot h(n) \cdot p(n) \tag{2.1}$$

$b(n)$, $p(n)$ and $h(n)$ are nodes n's base cost, present congestion cost and historical congestion cost, respectively. $p(n)$ and $h(n)$ both increase as a node n is overused. $d(n)$ is node n's intrinsic delay. $Crit(i,j)$, the criticality of the connection from net N_i source to sink s_{ij}, is defined as

$$Crit(i,j) = 1 - \frac{Slack(i,j)}{T_{crit}} \tag{2.2}$$

T_{crit} is the circuit critical path delay, and $Slack(i,j)$ the connection's timing slack.

The first term in right hand side of Eq. 2.1 is called *delay sensitive term*, and the second term *congestion sensitive term*. When a source-sink connection lies on a critical path, the congestion sensitive term becomes zero. This means that Pathfinder will ignore congestion and route this connection for minimum delay.

2.2.2 Delay Modeling of Routing Path

VPR version 4.30 employs the Elmore delay model to compute the delay of a route from a net source to any of its sinks. Pass transistors and wires are modeled as RC trees. And a buffer is modeled by a constant delay and a resistor. The Elmore delay of a source-sink path is:

$$T_{path} = \sum_{i \in path} (R_i C_{ds,i} + d_i) \tag{2.3}$$

Basically, the Elmore delay of the path is a summation of the Elmore delay of each node over all the nodes along the path. A node i can be a wire, a pass transistor, or a buffer. R_i is node i's equivalent resistance, and d_i is node i's intrinsic delay. $C_{ds,i}$ is the nodes i's downstream capacitance.

VPR 5 recommends that FPGAs assume uni-directional single driver wiring. As a result of that, it favors a constant delay model over the Elmore delay model [26]. Hence, the delay of a source-sink path now is:

$$T_{path} = \sum_{i \in path} d_i \tag{2.4}$$

Again, delay of the path is a summation of delay of each node over all the nodes along the path. A node i can be a wire, or a switch. d_i is a constant delay value specified in architecture files.

Note that VPR 5 can also compute Elmore delay for old-style VPR version 4.30 routing architectures. And it handles these two different cases transparently. In our research, we have worked with both the old-style routing architecture and the uni-directional single driver routing architecture. VPR 5 computes and reports delay values based on the Elmore delay model for old-style architectures; and it computes and reports delay values based on the constant delay model for uni-directional single driver architectures. Hereafter, we will simply use the same single word "delay" for routing path delay values computed based on both delay models.

VPR contains a delay extractor which can compute the delay of any routed net. The delay extractor can also incrementally compute the delay from the net source to a node in the current routing tree while a net is being routed.

2.2.3 Static Timing Analysis

Usually a timing graph is used to perform static timing analysis. In a timing graph, nodes represent input and output pins of basic circuit elements. In the case of FPGAs, the basic circuit elements include LUTs, registers, and IO pads. Edges

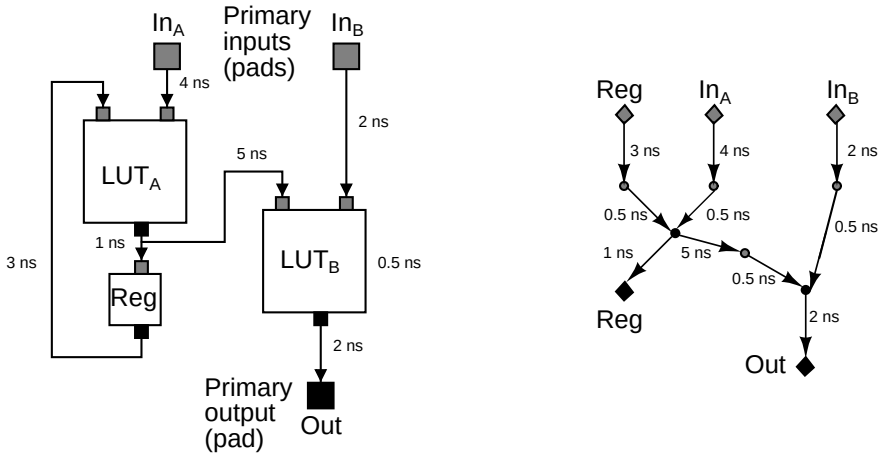


Fig. 2.7: A simple circuit, and its timing graph (from [6]).

are added between input pins of combinational logic blocks (e.g., LUTs) and their outputs. Edges are also added between pins which the circuit netlist specifies are connected. Each edge is annotated with a delay value. This delay value specifies the delay required to pass through the circuit element or routing.

As explained in Section 2.2.2, VPR's delay extractor computes the delay of a net's routing path, after this net is routed. Delay values of all nets' routing paths will be annotated on the edges in the timing graph, after all nets are routed.

Figure 2.7 shows a simple circuit implemented with 2-input LUTs and registers, and its corresponding timing graph.

In sequential digital circuits, we usually assume that signal arrival times at primary inputs are 0. We can start from the primary inputs, breadth-first traverse the timing graph, and compute arrival time for each node in the graph.

$$T_{arrival}(i) = Max_{j \in fanin(i)}\{T_{arrival}(j) + delay(j,i)\} \tag{2.5}$$

, where $delay(j, i)$ is the delay value annotated on the edge joining node j to i.

The node with the largest arrival time defines the maximum delay T_{crit} through the circuit. The required clocked period of this circuit should be no less than T_{crit}.

We can do a backward breadth-first traversal of the timing graph, and determine required times at all nodes. Required times of all primary outputs we set to T_{crit}. The required time of any node with fanouts is

$$T_{required}(i) = Min_{j \in fanout(i)}\{T_{required}(j) - delay(i, j)\} \tag{2.6}$$

Timing slack of the connection from node i to node j is then

$$Slack(i, j) = T_{required}(j) - T_{arrival}(i) - delay(i, j) \tag{2.7}$$

A connection with a slack of zero is said to be on the circuit critical path. Any delay increase of such a connection will result in increase of T_{crit}. It is easy to prove that the delay along the critical path equals to T_{crit}.

Note that one can find the circuit critical path by using a traceback method. The traceback starts from the node with the largest arrival time.

VPR contains a path-based static timing analyzer which can compute circuit critical path delay after all nets of a circuit have been routed.

2.3 Previous Work

The FPGA community have long realized that an optimized interconnect architecture is important to the overall performance of an FPGA. This can partly explain the extensive research activities on FPGA interconnect architecture. The

remainder of this section reviews some of the previous work related to this thesis.

2.3.1 Pipelined Interconnect

The idea of pipelined interconnect is mainly motivated by the fact that interconnect delay dominates logic delay and hence limits user designs' clock frequency. If interconnect between two logic blocks is pipelined and the delay of each interconnect pipeline stage is comparable to the delay inside logic blocks, interconnect delay will cease to be the bottleneck.

Architecture of pipelined interconnect has been reported in [36], [31], and [32]. HSRA [36] assumes a hierarchical routing architecture, instead of island-style architecture. Given a target clock frequency, the authors calculate (at HSRA design time) the length of interconnect that can be travelled within one clock cycle. At the end of this length, a register is placed. Figure 2.8 illustrates a switch block in this architecture. As can be observed, a register is present in the switch block. As a result of this register pipelined interconnect, signal route that consists of long interconnects takes more than one clock cycle. An HSRA prototype achieving 250MHz frequency was implemented with a 0.4um DRAM process. The authors have shown that HSRA architecture allows many pipelineable designs to run at frequencies of 2-17x the un-pipelined frequencies. However, HSRA architecture exhibits 50% area overhead per logic block, compared with un-pipelined architecture. The area overhead is due to pipeline registers in switch blocks and retiming registers in logic blocks.

In [32], Singh and Brown assume the island-style routing architecture. Figure 2.9 illustrates a registered switch in switch blocks. Note that this switch contains both a multiplexer and a register. Hence the route can be either pipelined or un-

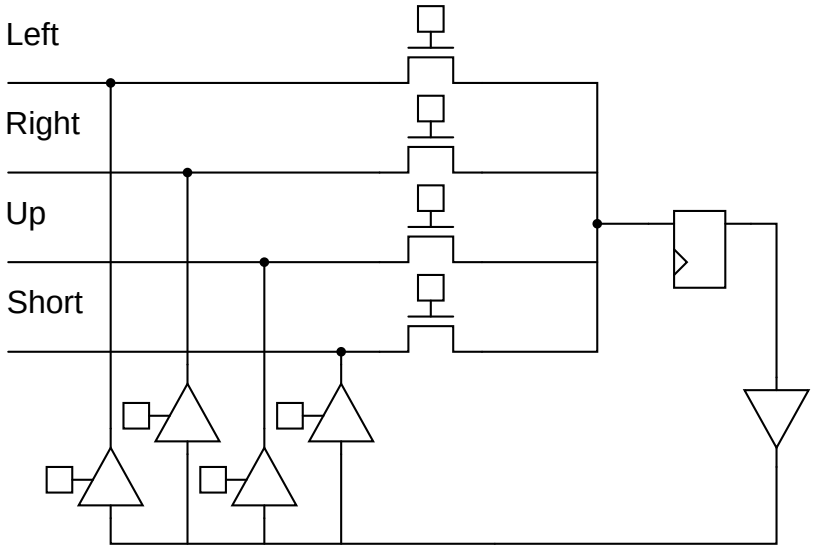


Fig. 2.8: Switch boxes with registers in HSRA architecture. From [36]

pipelined. Also an extra input register per LUT is added for retiming purpose. A noticeable feature of Singh and Brown's architecture is that the number of registered tracks is parameterizable. All wire segments on a registered track have registered switches at their ends. For the architecture in which 25% of all tracks are registered, the authors reported 12% - 25% speedup for circuit critical path delay, at the price of around 10% area overhead.

Pipelined interconnect architecture poses implications on FPGA design flow. For example, a typical scenario in [36] is that there are more registers between some LUT pairs in the placed and routed design than there were in the original netlist. As a result, retiming the design is necessary to ensure the correct logical behavior. More specifically in [36] the authors retimed all LUTs such that the number of registers between any two LUTs is larger or equal to the number of registers required by the interconnects. In [32], the authors proposed a modified CAD flow in which

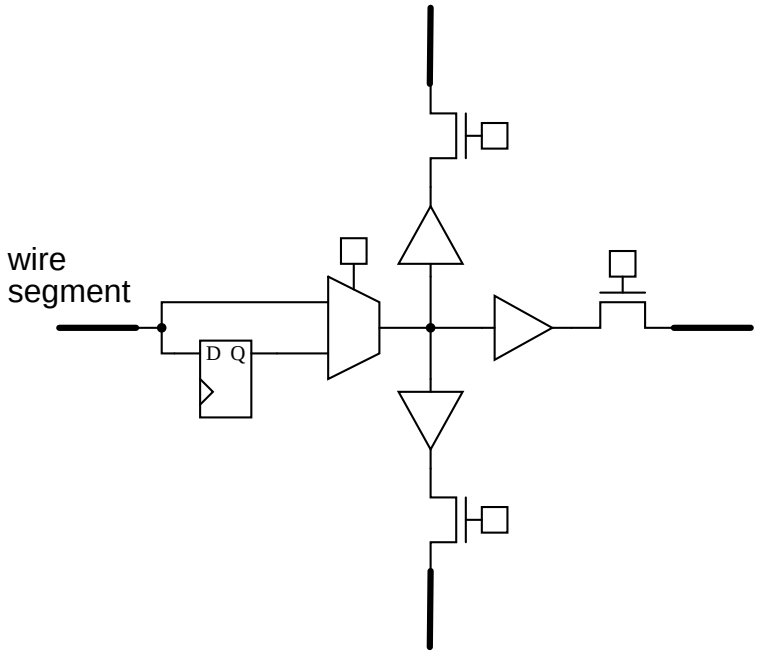


Fig. 2.9: Switch boxes with registers in Singh and Brown architecture. From [32]

the routing phase consists of two separate steps, namely, retiming aware routing and architecturally constrained retiming. In the first step, a conventional Pathfinder-based timing-driven routing algorithm routes all nets as if there were no registers in the connection network at all. Then long routes are shifted to registered tracks. In the second step, the actual retiming is performed.

A pipelining-aware router for FPGA called PipeRoute was presented in [30]. PipeRoute takes retimed netlist and pipelined FPGA architecture as inputs, and gives an assignment of nets (or signals) to routing resources as output. At the core of PipeRoute is an optimal one-delay router, which finds a lowest cost route between a source and a sink that goes through at least one registered switch-point. The registered switch-point is assumed to pick up either one clock cycle delay or no delay at all. This optimal one-delay router is then used to build a heuristic two-terminal

N-delay router. This heuristic recursively builds an N-delay route from a $(N-1)$-delay route by successively replacing each segment of the $(N-1)$-delay route with a one-delay route and then greedily selecting the lowest-cost N-delay route. Finally the two-terminal N-delay router is extended to a multi-terminal N-delay router. The multi-terminal router routes one sink at a time. All sinks are sorted in non-decreasing order of delay separation from the signal source. So the multi-terminal router first finds a route to the sink that is the least number of delays from the source, which is really a two-terminal N-delay routing problem. Then this partial routing tree connecting signal source and the first sink is expanded to include all remaining sinks. PipeRoute was employed for the design space exploration for the pipelined interconnect architecture in [29].

Armada, a timing-driven pipeline-aware router, was presented in [9]. In conventional FPGA routing, a link's timing criticality tends to be consistent between the routing iterations. This is so because the link's two endpoints are already fixed by the placement tool. However, in pipeline-aware routing, a link's criticality can change dramatically, depending on the route it takes. This happens because registers are discovered along the way during the routing. Authors of Armada tackled this problem by using *assumed criticality* breadth first search. To route from a source to a sink, N independent waves are started from the source, each assuming a different criticality from $1/N$ to 1.0. Each of these multiple simultaneous searches emphasizes delay versus congestion in a different way. The first wave exploration to reach the sink is the least expensive, representing a proper balance between congestion and delay. The actual value of N can be tuned to achieve tradeoff between runtime and timing accuracy. Experimental results presented in [9] showed that Armada produces significantly (as much as 60%) better critical path delays than

non- timing-driven router (such as PipeRoute).

2.3.2 Wave-Pipelined Interconnect

Pipelining, which breaks logic gates into stages and inserts storage elements (for example, registers) between stages, is a commonly used technique to improve the system throughput. Wave-pipelining can be considered as a virtual pipelining, in which logic gates serve as the storage elements. Hence, wave-pipelining can achieve high throughput, while avoiding the timing overhead of the storage elements. But for wave-pipelining to operate correctly, additional constraints apply. This can partly explain why wave-pipelining has not gained wide popularity.

Recently, wave-pipelining technique has been applied to FPGA interconnects, and has been proved to be able to improve the bandwidth of interconnect links. Mak *et al* in [21] presented closed-form expressions for throughput of both register-based pipelining and wave-pipelining in the context of FPGA interconnects. Their theoretic analysis showed that, in a 65nm technology and to achieve the same throughput, register-based pipelining requires 49% larger latency and 26% more power consumption than wave-pipelining. Their further SPICE modeling and simulation results confirmed advantages of wave-pipelining. For example, assuming PTM 65nm technology [1], wave-pipelining claims about 22% latency improvement and around 9% power improvement over registered-based pipelining. And wave-pipelining's improvement over register-based pipelining is expected to be even larger as technology scales down: at PTM 32nm technology, the latency and power improvement are shown to be 35% and 13%, respectively. Both the theoretical analysis and simulation results predicted that wave-pipelining could achieve 1.4

[1] Predictive Technology Model. See http://ptm.asu.edu/

Gbps throughput for a data link spanning 75 tiles in 65nm FPGAs.

Teehan *et al* in [34] proposed using wave-pipelining as an on-chip serial signaling technique, especially for data-path FPGA designs which operate on words. Their architecture added dedicated serializer and de-serializer to each logic block. The serializer captures a data word from the logic block. The serialized data is then transmitted along the interconnect wires. At the receiving side, the de-serializer restores the data word. The reference clock signal for serializer and de-serializer is generated from user clock by a ring oscillator at the serializer side. And this reference serial clock is transmitted to the receiving side parallelly with data. This is called source-synchronous signaling. The authors' SPICE simulation results reported that wave-pipelining could achieve 2 to 4 Gbps throughput for a data link spanning 200 tiles in 65nm FPGAs. It is interesting to note that the authors predicted interconnect area saving in spite of the apparent area overhead of serializers and de-serializers. For example, almost 50% interconnect area saving can be achieved, if 512 length-4 channel wires are replaced with 8-to-1 bit-serial wires. The reasons, the authors argue, are that the number of wires is greatly reduced and connection block area shrinks significantly. One problem of Teehan's interconnect architecture is power consumption. The authors reported that wave-pipelined interconnect exhibits at least 6-8x power penalty compared with parallel-bus, due to increased data activity and high-frequency toggling of serial timing strobe. As for reliability, the authors showed that wave-pipelining is sensitive to PVT (process, temperature, and voltage) variations as well as clock jitter and skew. Although it could achieve throughput as high as 5Gbps for short links, wave-pipelined interconnect has to run much slower than that (2 - 4 Gbps) due to reliability issue.

Wave-pipelined interconnect requires neither new routing algorithm nor modi-

fications to conventional FPGA design flow. However, as put forward in [34], better supply noise modeling and accounting is essential to achieve robust communication over wave-pipelined interconnect. Hence, CAD tool support in this regard is valuable.

2.3.3 Three-Dimensional Architecture

Three-dimensional architecture for FPGAs has been reported in [1], [13], and [24]. In these architectures, logic blocks are organized into a three-dimensional mesh array. These architectures have shorter average interconnect length than the conventional two-dimensional architecture. Also, they provide switch blocks with larger flexibilities. Besides, analytical results based on predicting models in [24] show that three-dimensional integration can improve interconnect delay by as much as 45% - 60%.

Alexander *et al* are possibly the first to propose a three-dimensional FPGA architecture [1]. Alexander's three-dimensional FPGA architecture is a generalization of the basic island-style architecture. In Alexander's architecture, each switch block has six immediate neighbors, as opposed to four in two-dimensional architecture. Three-dimensional switch blocks are analogous to their two-dimensional counterparts. They allow each channel segment to connect to a subset of channel segments incident on the other five faces of the switch block.

The authors' first-order analysis showed that, their proposed three-dimensional architecture achieves a shorter average interconnect length. For example, for FPGAs with 525 switch blocks (a modest size at the time of their work), average interconnect length in three-dimensional architecture is only half of that in two-dimensional architectures.

The authors assumed the subset switch block generalized to three dimensions: a segment connects to a single segment in each of the five adjacent channels. That is, $F_s = 5$, while $F_s = 3$ in two-dimensional architecture. As a result, a three-dimensional switch block requires more transistors than its two-dimensional counterpart. A more refined analysis, which takes this overhead into account, showed that, for FPGAs with more than 250 switch blocks, the benefits from reduced interconnect length outweigh the overhead of three-dimensional switch block. Hence the authors concluded that an average net would consume less routing transistors in three-dimensional FPGAs.

Another early three-dimensional FPGA architecture is Rothko [13]. Rothko architecture is based on Triptych architecture: a layer of Rothko architecture is similar to Triptych, and inter-layer connections are provided. In Triptych architecture, the basic logic element is RLB (routing and logic block)[2] [7]. Rothko architecture extends the Triptych RLB by allowing an RLB receive outputs of the neighboring RLBs in adjacent layers. The authors' experimental results from manually mapping two designs to a two-layer Rothko architecture showed that Rothko mapping's footprint is about half of that of Triptych.

In spite of their claimed advantages, these three-dimensional architectures have not been adopted by major commercial vendors, possibly due to challenges involved in the fabrication of three-dimensional chips. Fabrication of 3D chips is usually complex, requiring additional process steps. Alexander *et al* proposed that multi-chip module (MCM) technology is used to vertically stack a number of 2D FPGA dies [2]. Solder bumps are used to bond dies to an underlying substrate containing wires. In this way vertical interconnections are established. A problem with this

[2]An RLB in Triptych architecture can perform both logic implementation and signal routing. Hence the name.

vertical stacking technique is that there is a non-zero probability of defect when joining two dies. A second problem is heat dissipation. 3D architectures have a higher power-to-area ratio; hence the thermal issues are more challenging. Leeser *et al* proposed a thin film transfer approach for fabricating Rothko architecture [13]. Take a two-layer Rothko for example. First processed is a bulk silicon wafer containing half of the circuit. This is followed by the process of a second silicon-on-insulator (SOI) wafer containing the other half of the circuit. Then the SOI circuit is transferred face-down onto the top of the bulk silicon wafer. An adhesive bonds the transferred circuit to the bulk. Vertical interconnections are enabled by 3D vias, which connect metals at both layers to a third common metal.

In [28], the concept of extra-dimensional FPGA, in which *logical* third and fourth dimension are mapped to standard two dimensional IC, is introduced. Logic blocks are grouped into *xy planes*. A conventional 2D FPGA can be considered as a *xy* plane. Logic blocks only connect to wires on the same *xy* plane. Extra dimensions are formed by interconnecting planes. The experiments demonstrated that a four-dimensional FPGA provides better scalability. For example, as the design size increases from 20 to 585 CLBs [3], minimum channel width of 4D architecture remains constant, while that of 2D architecture needs to steadily increase.

Mostly, placement and routing algorithms for 2D FPGAs can be extended for 3D FPGAs. For example, the placement, global routing, and detailed routing algorithms used in [1] are extended from their 2D versions presented in [3]. More specifically, the 3D placement uses a 3D partitioning template (an $m * n * r$ grid), while the 2D version uses a 2D template (an $m*n$ grid). Another example is the 3D FPGA design flow presented in [8], which extends VPR. The author modified the

[3]This work assumes that a CLB consists of a four-input LUT and a bypass-able register.

VPR simulated annealing algorithm by introducing a non-adaptive schedule and a two-stage annealing. In fact, the first modification may not be strictly necessary, although the second modification is intended for optimizing circuit critical path delay. The author re-used VPR routing algorithm without any modification, as long as a routing resource graph can be generated from the 3D FPGA architecture description.

Stacked Silicon Interconnect (SSI) technology recently introduced by Xilinx represents another approach to achieve scalable interconnect for FPGAs [39]. SSI technology combines multiple FPGA die slices and a passive silicon interposer to create a die stack. The multiple die slices are placed side-by-side. This avoids the power and reliability issues that could result from stacking dies on top of each other. The passive silicon interposer interconnects the die slices by providing more than 10,000 traces between multiple die slices. Finally the die slices/ interposer stack is mounted on package substrate. Through-silicon vias (TSVs) are employed to prove connection between stack and package substrate. SSI technology is employed in Xilinx's Virtex-7 FPGAs.

FPGAs with SSI technology claim several advantages, compared with the traditional approach of connecting multiple conventional FPGAs to obtain a larger logic capacity. First, the interposer can provide as many as 10,000 connections, while IO pin count of a conventional FPGA is limited to around 1,200; Second, die-to-die latency of interposer is one fifth of pin-to-pin latency of standard IOs; Third, die-to-die connections implemented by interposer consume far less power.

Xilinx claims that, in terms of design flow, FPGAs with SSI technology can be treated like monolithic devices. There is no need of partitioning. And routing between die slices is transparent to users. Xilinx's ISE design tool supports the

Virtex-7 family.

2.3.4 Time-Multiplexed Interconnect

The seminal work on time-multiplexed FPGA is [35] by Trimberger *et al.* Based on the Xilinx XC4000E FPGA family, the introduced architecture features both time-multiplexed configurable logic blocks (CLBs) and time-multiplexed interconnects. The whole architecture is backed by one active configuration and eight inactive configurations. All the configuration bits of CLBs and interconnects can be flash reconfigured from one of the eight inactive configurations. Logic engine mode is one of the operation modes proposed by the authors for the time-multiplexed FPGA. In this mode, the FPGA sequences through multiple configurations. One configuration is called one *microcycle*. One pass through all the microcycles is called a *user cycle*. Therefore, the FPGA is reconfigured multiple times inside a user cycle. Our proposed FPGA architecture in this thesis is different from Trimberger's architecture, in that we time-multiplex interconnects only. We do not time-multiplex logic blocks. Our architecture is similar to Trimberger's architecture, in that we use the same concept of user cycle and microcycle. In particular, our architecture assumes that a user cycle is divided evenly into K microcycles. K is an architecture parameter, which in our work is chosen to be 2, 4, 6, or 8.

Recently a commercial time-multiplexed FPGA architecture was introduced by a start-up vendor Tabula. The architecture is named Spacetime [33]. A user clock cycle is divided to eight *folds* (fold 0, fold 1, ..., and fold 7). The FPGA device starts from fold 0. In each fold, the FPGA device performs a portion of user logic, and stores results in place. Then the device gets re-configured, and moves to the next fold, in which it uses the locally stored data to perform another portion

of user logic. At the completion of fold 7, the entire user logic has been performed, and a user clock cycle has been finished. After that, the device gets re-configured and starts again from fold 0, repeating the procedure described above. The re-configuration between folds is ultra-rapid. In fact, the first generation of Tabula device employs a 1.6 GHz clock for this reconfiguration. This enables a user clock of 200 MHz divided into eight folds.

Advantages claimed by Spacetime architecture include greater logic density, higher performance, as well as greater memory density. The memory density advantage stems from the fact that Spacetime architecture uses single-port memory cells while still providing fold-based multi-port capability.

One can see that, in many ways, Spacetime architecture resembles Trimberger's time-multiplexed FPGA architecture. However, Spacetime architecture does have some unique features. For example, it allows tradeoff between speed and area. If a higher clock frequency is desired from user design, less number of folds should be used. Hence more area of the device is committed. Take a 200 MHz user design with eight folds for instance. To get higher performance, the user could run the design at 400 MHz with four folds (or, 800 MHz with two folds only) [4].

Tabula claims that design flow with Spacetime architecture is similar to that with conventional FPGAs. In Tabula's design flow, synthesis optimized for Spacetime architecture is coupled with timing-driven place-and-route. As a result, mapping from RTL to Spacetime architecture is transparent.

The limited information disclosed by Tabula does not explicitly state whether interconnects are also time-multiplexed in Spacetime architecture.

[4] In fact, different sections of Tabular device core can run at different frequencies (and corresponding numbers of folds). The granularity of this tuning can be down to the level of individual tiles. Each tile in Tabula devices is logically equivalent to 16 4-input LUTs.

Another early interesting work by Lin *et al* [18] applied time-multiplexing to FPGA routing resources (i.e., interconnects) only. The proposed architecture has two configuration bits for each routing connection. A phase clock whose frequency is twice that of system clock is used. In operation, the phase clock shifts in a loop the two configuration bits, so that each of these two configuration bits alternately controls the routing switch. As a result, the physical routing resources work as two, time-multiplexed routing structures. The authors performed experiments with this architecture based on the Xilinx 4000 architecture, and their routing results reported a 30% reduction of channel density. The authors did not tell how the timing performance would be affected by the time-multiplexed interconnects.

A recent work by Francis *et al* [10] applied the idea of time-multiplexed interconnects to an Altera Stratix-based FPGA architecture. Both a realization of the time-multiplexed wirings and an algorithm to use the time-multiplexed wirings were described in detail. Similar to Trimberger's time-multiplexed FPGAs, Francis' architecture divides design clock cycle into a series of time slots, each representing a interconnect clock cycle. Design clock cycle and interconnect clock cycle here correspond to user cycle and microcycle, respectively, in Trimberger's architecture. Besides the extra configuration bits required for time-multiplexing, the authors also introduced latches to the time-multiplexed wire segments and look-up table inputs. Noticeably, the authors explored three architectural parameters: (1) the number of time slots in a design clock cycle, (2) the length of a time slot (in terms of picoseconds), and (3) the number of time-multiplexed wires per switch box. The third architectural parameter implied the co-existence of time-multiplexed wires and conventional wires. An important feature that Francis' architecture and our proposed architecture share in common is the co-existence of time-multiplexed wires

and conventional wires.

Chapter 3

FPGAs with Time-Multiplexed Interconnects

This chapter presents TM-ARCH, our proposed FPGA architecture with time-multiplexed interconnects. TM-ARCH architecture is based on the classical island-style architecture. Section 3.1 first gives an overview of TM-ARCH architecture. Section 3.2 introduces the term user clock cycle and microcycle. Time-multiplexing always has the technique of time slot. In our work, microcycle is the equivalent to time slot.

A key difference between TM-ARCH and the island-style architecture is that, in TM-ARCH, all wires in routing channels can be time-multiplexed. Section 3.3 explains how a wire in TM-ARCH is time-multiplexed.

Specially designed time-multiplexing switches (*TM switches*) are used in TM-ARCH to enable time-multiplexing of wires. TM switches differs from conventional switches in two aspects. First, a TM switch has multiple contexts (or configurations) associated with it. And it sequences through these multiple contexts within

a user clock cycle. Second, a TM switch is able to latch data. Section 3.4 describes these two aspects.

Finally section 3.5 presents our circuit design for TM switches. It helps understand TM switches' behavior described in section 3.4.

3.1 Overview

Figure 3.1 illustrates the architecture of our proposed FPGAs with time-multiplexed interconnects. This architecture is based on the island-style architecture. As in conventional island-style FPGAs, logic blocks are surrounded by horizontal/ vertical routing channels on all four sides. Each channel can contain multiple metal wires. The total number of wires in a channel is defined as the channel width. There is a switch block at every intersection of a horizontal channel and a vertical channel. Logic block pins connect to wires in routing channels through connection blocks. Programmable switches reside in connection blocks and switch blocks to achieve configurable routing.

At the architecture level, the only difference between this architecture and the conventional island-style architecture is that all wires in the routing channels can be time-multiplexed. Note that in this architecture, logic blocks cannot be multiplexed.

At the circuit level, the only difference between this architecture and the conventional island-style architecture is that, this architecture replaces the conventional switches with TM switches at the connection blocks and switch blocks. Unlike a conventional switch, a TM switch has multiple contexts, and sequences through the contexts in a clock cycle. To visually highlight this difference, we rep-

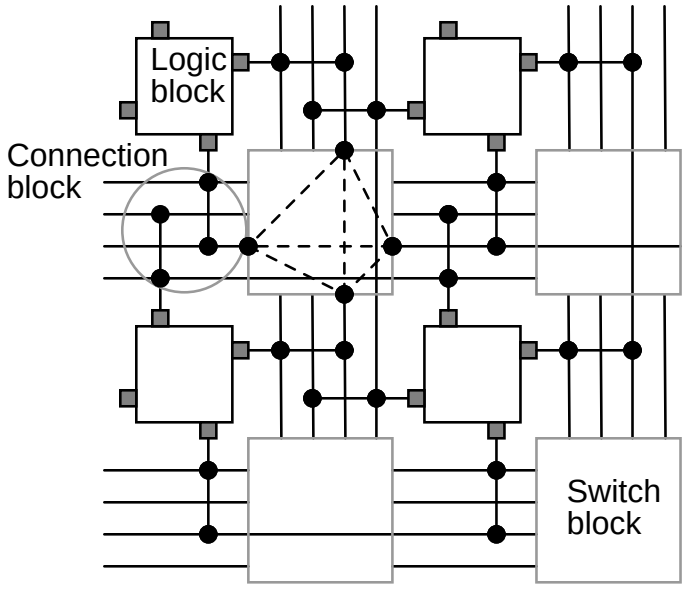


Fig. 3.1: Proposed architecture of FPGAs with time-multiplexed interconnects

resent a switch using a filled circle in Figure 3.1, while we represent a conventional switch using a void circle in Figure 2.1.

3.2 User Clock Cycle and Microcycle

Time-multiplexing generally works by dividing the time domain into time slots of fixed lengths and assigning a communication link to signals on the time slot basis. During time-multiplexing, multiple signals apparently take the same physical link, but they use the link at mutually exclusive time slots.

In TM-ARCH architecture, time-multiplexing works by dividing a user clock cycle evenly into multiple microcycles. Within a user clock cycle, multiple signals can use a same wire given that they use this wire at mutually exclusive microcycles.

We borrow the terms user clock cycle and microcycle from Trimberger *et al*'s

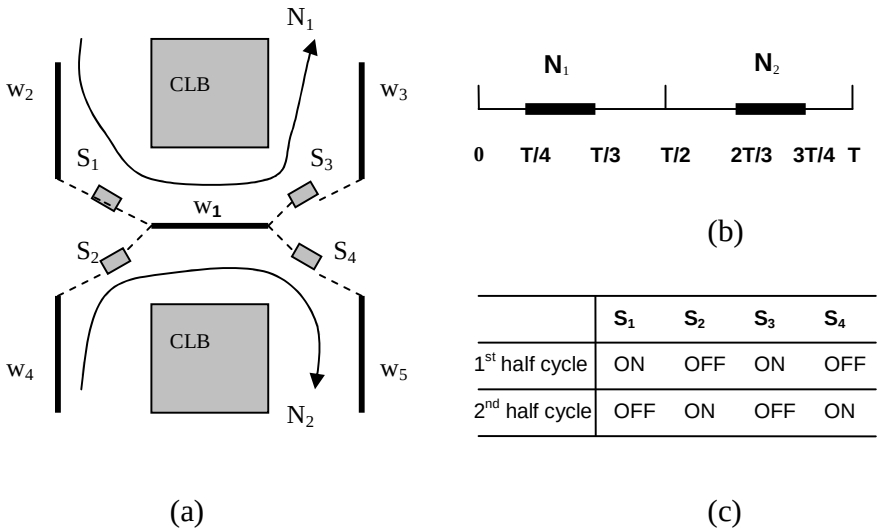
work [35]. User clock cycle is the clock cycle constrained by the critical path delay of users' design. For example, user clock cycle period must be no less than 1 ns, if the user's design has a critical path delay of 1 ns.

In our work, we define a parameter K for TM-ARCH architecture. This parameter denotes the number of microcycles in a user clock cycle. For we divide a user clock cycle evenly into K microcycles. Hence, microcycle period is $1/K$ of user clock cycle period.

3.3 Time-Multiplexed Wires

In FPGAs, route of a net very often consists of multiple wire segments. When the signal transition propagates from net source to net sinks, a wire in the route is idle before the signal transition arrives and after the transition leaves. That is, the net occupies the wire only for a short interval in a clock cycle.

In TM-ARCH FPGAs, multiple nets can time-multiplex a wire segment, given that these nets occupy the wire at different microcycles in a clock cycle. This can be best illustrated by the example shown in Figure 3.2(a). Assume that in this TM-ARCH device, a user clock cycle is divided into two microcycles. A wire w_1 is included in the route of both net N_1 and net N_2. This is allowed because N_1 and N_2 occupy the wire at different microcycles in a clock cycle. N_1 uses this wire at interval $[T/4, T/3]$, hence resides in the 1st microcycle. And N_2 uses this wire at $[2T/3, 3T/4]$, hence resides in the 2nd microcycle. Figure 3.2(b) illustrates the time intervals in which N_1 and N_2 occupy the wire.



	S_1	S_2	S_3	S_4
1st half cycle	ON	OFF	ON	OFF
2nd half cycle	OFF	ON	OFF	ON

Fig. 3.2: (a) Signals of N_1 and N_2 time-multiplex a wire; (b) N_1 and N_2 do not overlap in the time domain; (c) On/off state of the TM switches.

3.4 TM Switch

In TM-ARCH architecture, it is TM switches that provide hardware support for time-multiplexing of wires. Compared with switches in conventional FPGAs, TM switches have two features. First, a TM switch is associated with multiple contexts. Second, a TM switch is able to latch data. Section 3.4.1 and 3.4.2 present these two features respectively.

3.4.1 Multiple Contexts

In conventional FPGAs, a switch has only one context, and assumes that context until the whole FPGA is re-programmed [1]. This re-programming usually requires that the circuit's operation has to be suspended. A switch's context de-

[1]Some modern commercial FPGAs support an advanced feature called partial reconfiguration. In the case of partial reconfiguration, a switch assumes its context until the region where it resides is re-programmed.

termines its on/off state. For it assumes the one context until re-programming, a switch's on/off state will not change throughout the circuit's operation.

In our proposed architecture, a TM switch has K contexts: one context for each microcycle. As the time ticks through the K microcycles in a user clock cycle, a TM switch sequences through the K contexts. As a result, a TM switch's on/off state can change on-the-fly during the circuit's operation. This unique feature of TM switches provides the support for the time-multiplexing of wires in our architecture.

As an illustration, Figure 3.2(c) lists the on/off states of TM switches (S_1, S_2, S_3, and S_4) that can realize the time-multiplexing shown in Figure 3.2(a). In this case, the TM switches have two contexts, and sequence through the two contexts in a clock cycle. More specifically, the TM switches assume the first context in the first half of the cycle (i.e., 1st microcycle), and the second context in the second half cycle (i.e., 2nd microcycle).

3.4.2 Latching Capability of TM Switch

Another feature of TM switch is that it latches the current logic value when it transits from on state to off state.

The necessity for this capability of latching data is best illustrated by the example shown in Figure 3.2. In the 1st half cycle, TM switches S_1 and S_3 are on. Hence, the connection w_2-w_1-w_3 is maintained for signal of net N_1. In the 2nd half cycle, TM switches S_2 and S_4 turn on, while S_1 and S_3 turn off. Hence, the connection w_4-w_1-w_5 is maintained for signal of net N_2, while the connection w_2-w_1-w_3 is broken. Now consider the wire w_3 in the 2nd half cycle. To prevent w_3 from floating, the switch S_3 needs to serve as the driver. This is when the latching

capability comes to rescue. When it transits from on to off, S_3 latches the current logic vale, and then drives w_3.

Note that in Francis' architecture [10], the latching capability is provided by the wire itself, not the switch. This marks an important difference between Francis' architecture and ours. A derived difference is that our architecture does not need to deploy latches at look-up table inputs.

3.5 TM Switch Design

To help understand TM switch's behavior described in section 3.4, in this section we present our circuitry design for TM switches. Since this thesis does not claim circuit design for TM switches as one of the main contributions, the circuit design presented here may not be optimal, in terms of area or timing.

Section 3.5.1 first presents design of TM pass transistor. TM pass transistor is extended from a simple NMOS pass transistor. It is a basic structure used by us to build TM switches. Section 3.5.2 presents how we use TM pass transistor to build TM switches that have multiple contexts. Section 3.5.3 then presents how we use TM pass transistor to add data latching ability into TM switches.

3.5.1 TM Pass Transistor

A basic component used in both bi-directional and uni-directional wiring is the pass transistor. The gate of pass transistor (usually NMOS) is connected to one bit of configuration memory (or, one memory cell). Hence, the pass transistor's on/off state is controlled by this bit. An input line connects to the **in** terminal; and the **out** terminal connects to the downstream circuitry (another NMOS pass transistor

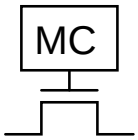


Fig. 3.3: A pass transistor controlled by a SRAM cell

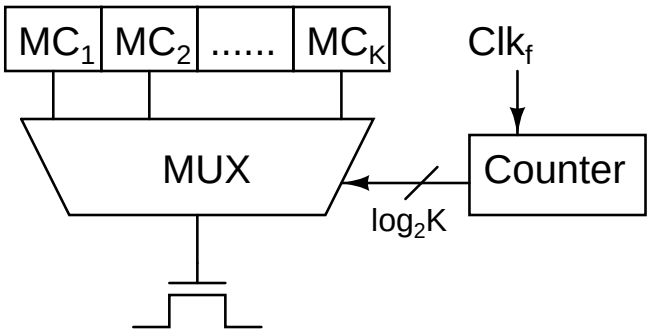


Fig. 3.4: A TM pass transistor

in the case of two-level hybrid multiplexer, as shown in Figure 2.5). Figure 3.3 illustrates this basic component.

In our work, we extend this basic component in the following way. First, we back the pass transistor with K bits of memory. Then, we use a K:1 multiplexer to select one out of the K memory bits. Third, we use a circular counter to generate the select signals. This counter is clocked by a fast clock signal (Clk_f), of which the frequency is K times that of system clock (Clk). The counter counts from 0 upwards to $K-1$ repeatedly. Figure 3.4 illustrates our extension. For the sake of brevity, hereafter we will refer to this extended pass transistor as *TM pass transistor*.

Now we examine how the TM pass transistor works. Without loss of generality, we assume K=2 for this moment. The pass transistor now is backed by two bits,

Table 3.1: Pass transistor's on/off state in 1st and 2nd half cycles for different configurations

MC_1MC_2	1st half cycle	2nd half cycle
00	Off	Off
01	Off	On
10	On	Off
11	On	On

MC_1 and MC_2. The 2:1 multiplexer's one-bit select signal is generated by the counter. The counter counts 0, 1, 0, 1, and so on. Frequency of Clk_f is two times that of Clk.

Assume that MC_1 will be selected that if the select signal is 0, and MC_2 will be selected otherwise. In the first half cycle of Clk, counter value is 0, so MC_1 controls the on/off state of the pass transistor. In the second half cycle of Clk, counter value is 1, so MC_2 controls the pass transistor. Table 3.1 lists the pass transistor's on/off state for different combinations of MC_1 and MC_2.

Comparing Figure 3.3 with Figure 3.4, one finds that our extension brings in area overhead. The area due to configuration memory increases by a factor of K. Also, an extra K:1 multiplexer is required for the TM pass transistor. However, the counter can possibly be shared by a number of TM pass transistors.

3.5.2 Design for Multiple Contexts

Now we use TM pass transistor to build TM switches with multiple contexts. Switches in conventional island-style FPGAs are of different types. For example, pass transistor switch and tri-state buffer switch are the two common types for bi-directional wiring; direct drive mux switch is widely used for uni-directional wiring. For major commercial FPGA vendors are now using uni-directional wiring, here

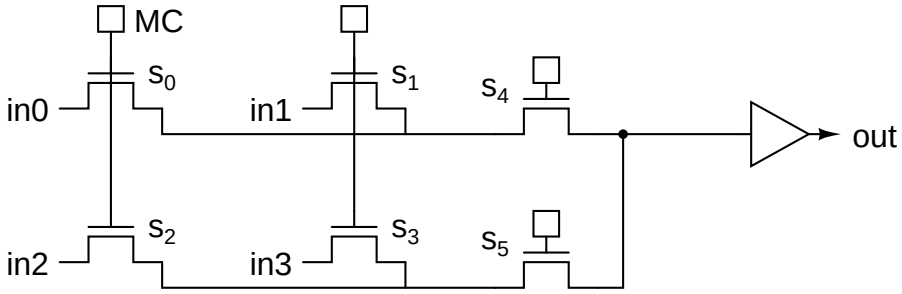


Fig. 3.5: A direct drive mux type switch. Its multiplexer selects one from four input lines.

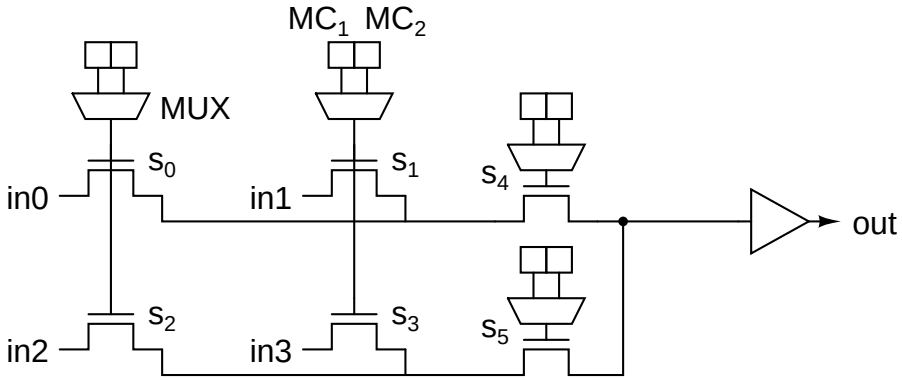


Fig. 3.6: A TM switch of direct drive mux type. For the sake of illustration, circular counters are not shown, and K is assumed to be 2.

we present how we use TM pass transistor to build direct drive mux -type TM switches with multiple contexts.

Figure 3.5 illustrates a direct drive mux switch which selects one from four input lines to drive its output line. The 4-to-1 multiplexer, implemented with NMOS pass transistors, assumes a two-level hybrid topology. Figure 3.6 illustrates a TM switch of direct drive mux type. Comparing Figure 3.6 with Figure 3.5, one can see that TM pass transistors replace NMOS pass transistors. Notice that TM pass transistor s_0 and s_2 share the configuration memory cells. So do s_1 and s_3.

Now we see how we can use this TM switch to enable the time-multiplexing

Table 3.2: Configurations of TM pass transisotrs to achieve the time-multiplexing in Figure 3.2.

	MC_1	MC_2
s_0 & s_2	0	1
s_1 & s_3	1	0
s_4	1	0
s_5	0	1

shown in Figure 3.2. From Section 3.3, we know that net N_1 uses wire w_1 in the 1st microcycle and net N_2 uses wire w_1 in the 2nd cycle. Hence the connection w_2-w_1 needs be on in the 1st microcycle and off in the 2nd microcycle. The connection w_4-w_1 needs be off in the 1st microcycle and on in the 2nd microcycle.

Now we deploy the TM switch at the left endpoint of wire w_1 [2]. Without loss of generality, we assume that wire w_2 connects to input line **in1**, and wire w_4 connects to input line **in2**. Output line **out** connects to wire w_1. Table 3.2 shows configurations of TM pass transistors s_0 - s_5 that could meet requirement of the time-multiplexing in Figure 3.2.

3.5.3 Design for Latching Ability

The direct drive mux TM switch present in the previous section has multiple contexts. But it does not have the ability to latch data. Here we add one more TM pass transistor (s_6) to it and obtain its latching ability by feeding the buffer output back to the buffer input. This is illustrated in Figure 3.7. At any particular microcycle, the feedback loop is enabled (or disabled) if s_6's configuration bit at this microcycle is 1 (or 0).

Now we see how we can use this TM switch to provide the data latching

[2]Note that now the TM switch replace the two switches S_1 and S_2 in Figure 3.2.

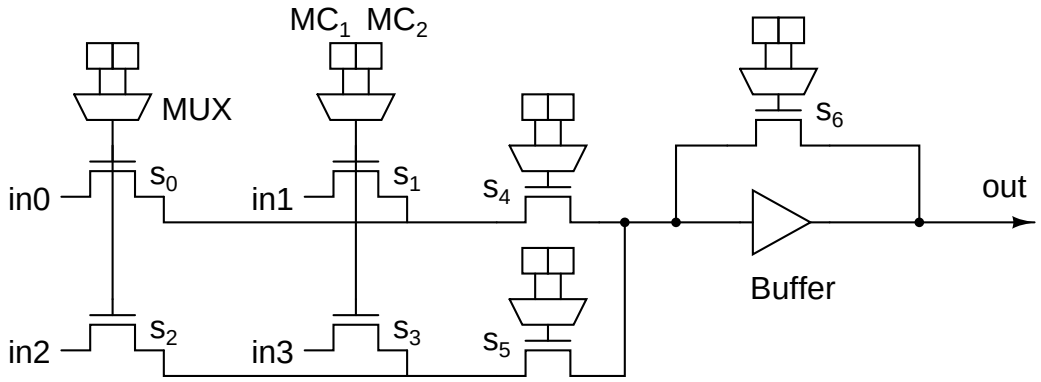


Fig. 3.7: A TM switch of direct drive mux type. For the sake of illustration, circular counters are not shown, and K is assumed to be 2. Notice that this TM switch can latch data.

Table 3.3: Configurations of TM pass transisotrs to achieve the time-multiplexing in Figure 3.2.

	MC_1	MC_2
s_0 & s_2	0	0
s_1 & s_3	1	0
s_4	0	0
s_5	1	0
s_6	0	1

required in Figure 3.2. From Section 3.4.2, we know that connection w_1-w_3 needs be off in the 2nd microcycle. The switch S_3 needs drive w_3 in the 2nd microcycle to prevent w_3 from floating.

Now we deploy the TM switch shown in Figure 3.7 at the lower end point of wire w_3 [3]. Without loss of generality, we assume that wire w_1 connects to input line **in3**, and output line **out** connects to wire w_3. Table 3.3 shows configurations of TM pass transistors s_0 - s_6 that could meet requirement of the time-multiplexing in Figure 3.2.

[3]Note that now the TM switch replaces the switch S_3 in Figure 3.2.

Chapter 4

Time-Multiplexing -Aware Timing-Driven Routing Algorithm

This chapter presents our proposed time-multiplexing -aware timing-driven routing algorithm. This algorithm is intended to route a circuit onto TM-ARCH FPGAs. Section 4.1 presents formulation of the problem. Our algorithm is an extension of VPR 5 timing-driven routing algorithm, which has been well documented [6] [20]. Hence this chapter focuses on our major improvements relative to VPR 5 algorithm.

One technique introduced in our algorithm is signal's occupation bitmap. Occupation bitmap indicates at which microcycle(s) this signal would probably use a wire. Section 4.2 presents this technique and explains how we compute occupation bitmap. A second technique introduced in our algorithm is to compute a wire's congestion penalty on a microcycle basis. This is to cater to TM-ARCH architecture,

in which a wire can be used by nets on a microcycle basis. Section 4.3 presents this technique and explains how we compute a wire's congestion penalty at each microcycle. An important innovation of our algorithm is time-multiplexing -aware congestion cost, which is the key to our algorithm's ability to identify and schedule multiple nets to time-multiplex a wire. Section 4.4 describes formulation of our time-multiplexing -aware congestion cost.

Section 4.5 presents formulation of our overall cost, which includes a congestion sensitive term and a delay sensitive term. This overall cost function gives our algorithm the ability to resolve congestion and optimize timing. Section 4.6 defines how we determine that a routing solution is legal for TM-ARCH architecture. The used criterion is slightly different from that of VPR 5 router.

Finally, Section 4.7 puts all these together and presents pseudo code of our algorithm. And Section 4.8 gives some further details of our algorithm. Section 4.9 presents analysis of our algorithm on its time complexity, memory requirement, and unroutability detection.

4.1 Problem Formulation

A routing-resource graph $G(V, E)$ is used to represent the routing resources in TM-ARCH and their connections. The set of vertices (or nodes) V corresponds to wires or CLB pins, and the set of edges E to switches. Associated with each node and each edge, there is a delay d. A node's capacity Cap is defined as the maximum number of different nets that can use this node at a microcycle. We define that, in TM-ARCH architecture, Cap equals to 1 for the nodes corresponding to wires. This is so, because at any microcycle a wire must be used by at most one

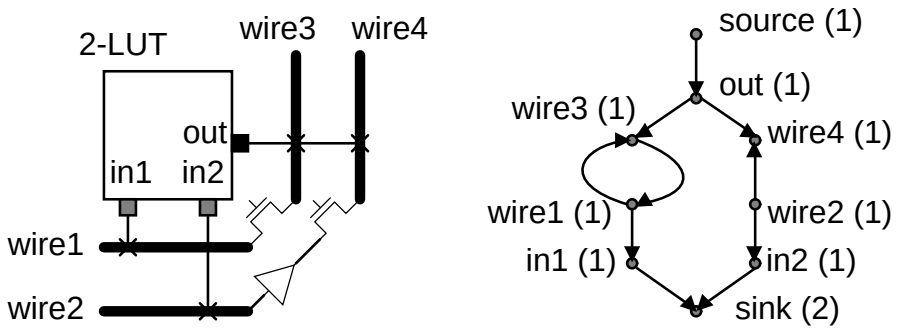


Fig. 4.1: Representing TM-ARCH *architecture as a routing resource graph*

net. Fig. 4.1 shows the routing-resource graph for a portion of a TM-ARCH device with K=2. The number shown next to the node name denotes a node's capacity. Note that **source** and **sink** are two dummy nodes. A capacity of 2 is assigned to **sink** to model the logic equivalence of the two input pins of the 2-input LUT.

For a signal i to be routed in TM-ARCH, its net N_i is a set of terminals, including the source terminal s_i and sink terminals s_{ij}. N_i forms a subset of V. A routing solution to net N_i is a directed routing tree RT_i embedded in G and connecting s_i with all s_{ij}.

The task of a router for our TM-ARCH architecture is to route all the nets and optimize the circuit delay at the same time. Since a wire can be time-multiplexed in TM-ARCH, the router should be able to identify and schedule multiple nets to time-multiplex a wire.

4.2 Signal's Occupation Bitmap

Our time-multiplexing -aware routing algorithm is able to identify a signal (or, net) that may multiplex a wire with other signals. And it does this on-the-fly while

routing the signal net. Our algorithm achieves this capability by actively tracking a signal's arrival and leave time at a wire. Based on these two timing values, it computes occupation bitmap for this signal. This occupation bitmap indicates at which microcycle(s) this signal would probably use this wire.

Section 4.2.1 describes how we compute signal arrival and leave times and record them internally in our data structure. Section 4.2.2 describes how we compute and record occupation bitmap.

4.2.1 Arrival Time and Leave Time

At the core of Pathfinder algorithm, VPR 5 routing algorithm, and our algorithm is a maze router. The maze router routes a net by starting from the net source and wave-expanding over the routing-resource graph until the wave-front reaches the destination sink of the net. This process is called wave expansion. The maze router then back-traces and records the path from net source to sink.

In our algorithm for each wire being expanded, we compute two timing values, arrival time ($t_{arrival}$) and leave time (t_{leave}). As its name suggests, $t_{arrival}$ is the time at which the signal arrives at this wire. And t_{leave} is the time at which the signal leaves this wire. They can be formally defined as:

$$t_{arrival}(n) = \sum_{m \in path\ from\ source(i)\ to\ n} d_m + T_{arrival}(source(i)) \tag{4.1}$$

$$t_{leave}(n) = t_{arrival}(n) + d_n \tag{4.2}$$

In Equation 4.1 the first term on the right hand side evaluates to the delay from source of net i to the node n. The second term is signal arrival time at source of

net i, as computed according to Equation 2.5.[1] The sum of these two terms gives the signal arrival time at the node n.

In Equation 4.2, d_n is node n's delay. That is, it takes the signal d_n time to pass through node n. The value of d_n is specified in FPGA architecture files. Hence, the sum of $t_{arrival}$ and d_n gives the signal leave time.

Like VPR timing-driven router, our router uses a heap data structure to facilitate wave expansion. When a node n is added to the heap, we store its t_{leave}. Later when any of n's fan-out nodes is being expanded, we compute the fan-out node's arrival and leave time as:

$$t_{arrival}(m) = t_{leave}(n) \tag{4.3}$$

$$t_{leave}(m) = t_{arrival}(m) + d_m \tag{4.4}$$

, in which m is a fan-out node of n.

In this way, we incrementally compute a node's arrival and leave time, rather than start from the net source and compute all the way down to the current node.

For each wire included in a net's route, we record the signal arrival time and leave time. During the routing, at sometime a wire may be included in multiple nets' routes. In this case, we record all these nets' arrival/ leave time at the wire. Internally we implement this using a linked list. Each list node stores the net index, signal arrival time, and signal leave time. We put the linked list under the wire. Going through this list, we know all the nets currently using the wire, and their

[1]Equation 2.5 computes signal arrival time for nodes in the timing graph. VPR actually does not create a node for a net source in timing graph. But it does create a node for the logic block output pin which drives the net. Hence we set $T_{arrival}$(source(i)) to be the arrival time at the driving node.

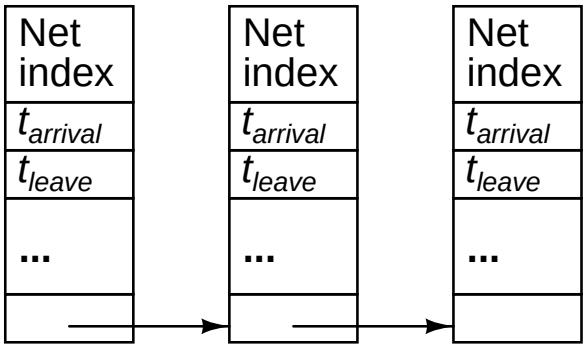


Fig. 4.2: A linked list to record all the nets currently using the wire.

arrival/ leave times.

Figure 4.2 illustrates such a linked list. In this case, three nets are currently using the wire in their routes.

4.2.2 Occupation Bitmap

For each wire being expanded, we use the above calculated $t_{arrival}$ and t_{leave} values to compute what we call occupation bitmap. This bitmap is used by us to represent occupation of a wire n by a net i at difference microcycles. It is an array consisting of K elements. Each element corresponds to a microcycle: the first element corresponds to the 1st microcycle, and the last to the K-th microcycle.

Each element in the bitmap array takes a binary value (0 or 1). A value of "1" means that the net occupies the wire at the corresponding microcycle. A value of "0" means that the net does not occupy the wire at the corresponding microcycle.

The formula we use to calculate each element in the array is as follows:

$$Bitmap[k] = \begin{cases} 0, & \text{if } t_{arrival} > kT_{ucycle} \text{ or } t_{leave} < (k-1)T_{ucycle} \\ 1, & \text{else} \end{cases} \tag{4.5}$$

$$T_{ucycle} = T_{crit}/K \tag{4.6}$$

, in which k is in the range [1, K]. The basic idea behind this equation is that the net does not occupy the wire at a particular microcycle, if the net's signal arrives after this microcycle or the signal leaves before this microcycle.

Figure 4.3 presents the pseudo code of the algorithm we use to compute bitmaps. First, we reset all K elements in the bitmap array (line 1 - 3). Then we determine the very microcycle in which the signal arrives, and denote it with a variable *begin* (line 5 - 9). After that, we determine the very microcycle in which the signal leaves, and denote it with a variable *end* (line 10 - 14). Finally, we set all the microcycles between *begin* and *end* to 1 (line 15 - 17), for these are the microcycles in which the net will occupy the wire.

For each wire included in a net i's route, we also record the bitmap array. Recall from Section 4.2.1 that, for each wire included in a net i's route, we have recorded net i's signal arrival and leave time at the wire. We use these recorded $t_{arrival}$ and t_{leave} to compute the bitmap. And the computed bitmap is recorded along with $t_{arrival}$ and t_{leave}. As in the case of $t_{arrival}$ and t_{leave}, we record multiple bitmaps if a wire is included in multiple net's routes.

Figure 4.4 shows the same linked list as that in Figure 4.2, but with the corresponding bitmap computed and recorded in each list node.

4.3 Congestion Penalties at Microcycles

Pathfinder algorithm uses occupancy to record the number of nets currently using a wire. This occupancy indicates the degree of congestion at the wire: a larger occupancy value means more congestion. Suppose that Pathfinder is currently

$t_{arrival}$: Signal arrival time at wire n;
t_{leave} : Signal arrival time at wire n;
T_{crit} : Circuit critical path delay;
K : Number of microcycles in a clock cycle;

```
1.  for (i=1; i <=K; i++) {
2.      Bitmap[i] = 0;
3.  }
4.  T_ucycle = T_crit / K;
5.  for (i=1; i <=K; i++) {
6.      if (t_arrival > (i-1)* T_ucycle && t_arrival < i* T_ucycle) {
7.          begin = i;
8.          break;
9.  }
10. for (i=1; i <=K; i++) {
11.     if (t_leave > (i-1)* T_ucycle && t_leave < i* T_ucycle) {
12.         end = i;
13.         break;
14. }
15. for (i=begin; i <=end; i++) {
16.     Bitmap[i] = 1;
17. }
```

Fig. 4.3: Pseudo code of our algorithm to compute occupation bitmaps.

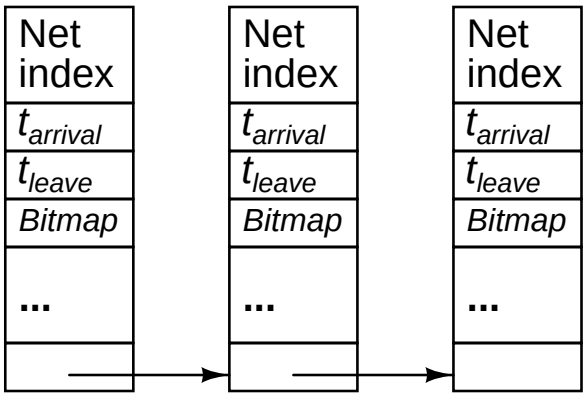


Fig. 4.4: A linked list to record all the nets currently using the wire.

routing a net. When its wave-front arrives at a wire with a large occupancy during wave expansion, Pathfinder assigns a large congestion penalty to this wire. In the end from the standpoint of resolving congestion, Pathfinder is unlikely to include this wire in the routing path of the net.

In TM-ARCH FPGA devices, a wire can be used by nets on a microcycle basis. Hence our algorithm uses micro occupancy to record the number of nets currently using a wire at each microcycle. This micro occupancy indicates the degree of congestion at the wire's each microcycle. Now suppose that our algorithm is currently routing a net on a TM-ARCH FPGA. When its wave-front arrives at a wire, our algorithm then computes congestion penalties at the wire's each microcycle based on the wire's micro occupancy values. At a microcycle, a larger micro occupancy leads to a larger congestion penalty.

Section 4.3.1 details how we compute and maintain micro occupancy values for a wire. Section 4.3.2 details how we compute a wire's two congestion penalties (present congestion penalty and historical congestion penalty) based on the wire's micro occupancy.

4.3.1 Micro Occupancy

We use micro occupancy to record the number of nets that are currently using a wire at each microcycle. Defined for each wire, micro occupancy is an array consisting of K elements. Each element corresponds to a microcycle: the first element corresponds to the 1st microcycle, and the last to the K-th microcycle.

Each element in the array takes an integer value. The value represents the number of nets that are currently using the wire at the corresponding microcycle. In short, micro occupancy records a wire's occupancy at each microcycle.

Initially all elements of each wire's micro occupancy are zeros. When a wire n is included in a net i's route, n's micro occupancy is updated based on the bitmap as computed in section 4.2.2.

$$uOcc[k] = \begin{cases} uOcc[k] + 1, & \text{if } Bitmap[k] = 1 \\ uOcc[k], & \text{else} \end{cases} \tag{4.7}$$

The idea behind Equation 4.7 is simple. The k-th element of bitmap being 1 means that the net occupies the wire in the k-th microcycle. Hence, the k-th element of wire n's micro occupancy increases by one, as wire n is now included in net i's route.

When a net i is being ripped-up, micro occupancy of all the wires in net i's route are also updated.

$$uOcc[k] = \begin{cases} uOcc[k] - 1, & \text{if } Bitmap[k] = 1 \\ uOcc[k], & \text{else} \end{cases} \tag{4.8}$$

The idea here is similar. Since wire n was occupied by net i in the k-th microcycle, and net i is now being ripped-up, we need to decrease micro occupancy's k-th element by one.

4.3.2 Present and Historical Congestion Penalty

For we have micro occupancy records, we can define and compute a wire n's present and historical congestion penalty in each microcycle.

The formulas we use to compute present and historical congestion penalty are

Table 4.1: Routing schedule of our time-multiplexing -aware routing algorithm

Routing Schedule	Value
p_{fac}	0.5 in the first and the second routing iteration; 1.3 times its previous value from the third iterations onwards.
h_{fac}	1.0 for all the iterations

listed as follows:

$$p[k] = 1 + max(0, p_{fac}[uOcc[k] + 1 - Cap]) \tag{4.9}$$

$$h[k]^i = \begin{cases} 1, & i = 1 \\ h[k]^{i-1} + max(0, h_{fac}[uOcc[k] - Cap]), & i > 1 \end{cases} \tag{4.10}$$

Note that these two formulas are essentially the same as those used by VPR authors. In VPR router, the formulas are used to compute a wire's present and historical congestion penalty. And here the formulas are used to present and historical congestion penalty in each microcycle.

Like VPR authors, we update $p[k]$ for all the affected wires whenever any net is ripped-up and re-routed. And we update $h[k]$ for all wires only after an entire routing iteration.

Routing schedule defines the value of h_{fac} and p_{fac} in each routing iteration. Table 4.1 lists the routing schedule used by us. This is the default routing schedule of VPR 5.

4.4 Multiplexing-aware Congestion Cost

A key innovation of our algorithm is time-multiplexing -aware congestion cost. The idea behind this congestion cost is simple. If a wire in TM-ARCH FPGAs is currently used by two signals *and* these two signals use this wire at mutually exclusive microcycles, our algorithm should not see congestion at this wire. This is so, because these two signals time-multiplex this wire, hence cause no congestion.

We have shown in Section 4.2 that, while it is routing a signal, our algorithm computes this signal's occupation bitmap at a wire being expanded. This occupation bitmap indicates at which microcycle(s) this signal would probably use this wire. Then we have shown in Section 4.3 that, our algorithm computes congestion penalties at a wire's each microcycle based on the wire's micro occupancy values. We in this section show how our algorithm computes the time-multiplexing -aware congestion cost based on a signal's occupation bitmap and a wire's congestion penalties at microcycles.

Figure 4.5 presents the pseudo code of our algorithm to compute a wire's congestion cost during wave expansion. Here we assume that we have already computed the bitmap array, as described in Section 4.2. Further more, we assume that the present and historical congestion penalties in each microcycle are up-to-date.

Line 1 - 4 loops over all the microcyles, and records the largest historical congestion penalty. Line 5 - 10 looks for the first "1" value in the bitmap array, and sets the *begin* index. This index indicates from which microcycle the wire's occupation by the wire starts. Similarly, line 11 - 16 sets the *end* index, which indicates the microcycle until which the occupation ends. Line 17 - 20 records

```
Bitmap[1..K] : Bitmap array of net i at wire n;
p_n[1..K] : Present congestion penalty array of wire n;
h_n[1..K] : Historical congestion penalty array of wire n;
b_n : Base cost of wire n;

1.  accCost = 0.;
2.  for (i=1; i <=K; i++) {
3.      accCost = max(accCost, h_n[i])
4.  }
5.  for (i=1; i <=K; i++) {
6.      if (Bitmap[i] == 1) {
7.          begin = i;
8.          break;
9.      }
10. }
11. for (i=begin; i <=K; i++) {
12.     if (Bitmap[i] == 0) {
13.         end = i-1;
14.         break;
15.     }
16. }
17. presCost = 0.;
18. for (i=begin; i <=end; i++) {
19.     presCost = max(presCost, p_n[i])
20. }
21. cCost = presCost*accCost*b_n
```

Fig. 4.5: Pseudo code of our algorithm to compute congestion cost.

the largest present congestion penalty in the microcycles between *begin* and *end* inclusive. Finally line 21 computes the congestion cost, which is the product of the wire's base cost, the largest historical congestion penalty found, and the largest present congestion penalty found.

$$cCost = p_n * h_n * b_n \tag{4.11}$$

Note that Equation 4.11 is modeled after the equation that VPR router uses to compute congestion cost. The differences here are

- For p_n, we use the largest present congestion penalty found in the microcycles during which the wire is occupied.
- For h_n, we use the largest historical congestion penalty found in all the microcycles.

The whole idea here is to make the congestion cost multiplexing-aware. A net occupying a wire from the *begin*-th microcycle until the *end*-th microcycle needs be aware of congestion in these microcycles only. It needs not concern about congestion in the other microcycles.

4.5 Overall Cost Function

Like that of VPR timing-driven router, the overall cost function of our router is the sum of two terms. One is the congestion sensitive term, which is the congestion cost as computed in Section 4.4. The other is the delay sensitive term. We follow VPR authors' practice which uses the wire intrinsic delay as the delay cost, and

weighs the two terms based on the connection's timing criticality. Equation 4.12 presents the overall cost function used in our routing algorithm.

$$c(n) = Crit(i,j) * d(n) + [1 - Crit(i,j)] * cCost(n) \tag{4.12}$$

4.6 Legal Routing Solution

Like the VPR router, after each routing iteration our time-multiplexing -aware router checks if the current routing solution is legal. If yes, our router exits and returns this routing solution. If not, the router starts another iteration. By default our router will try at most 50 iterations before giving up.

A legal routing solution contains no overused routing resources. A wire that is multiplexed by multiple nets is not overused, as long as it is occupied by at most one net in any microcycle. Mathematically, our router checks if the following condition holds in each microcycle.

$$uOcc[k] \leq 1 \quad k \in [1, K] \tag{4.13}$$

4.7 Pseudo Code

Figure 4.6 presents pseudo code of our time-multiplexing -aware timing-driven routing algorithm. This algorithm largely follows VPR 5 timing-driven routing algorithm. Hence here we only point out our major improvements relative to VPR 5 timing-driven routing algorithm.

Lines 13, 22, and 26 reflect the fact that our algorithm computes and updates signal arrival and leave times. This has been detailed in Section 4.2.1. Lines 14,

```
     G_t: Circuit timing graph
     Q(i): Priority queue used while routing net N_i
     RT(i): Routing tree of net N_i
     Source(i): Source of net N_i

1    Back-annotate placement delays of all nets into G_t;
2    Propagate timing in G_t, and compute T_crit;
3    Crit(i,j) = 1.0 for all i and j;
4    while (overused resources exist) {
5      for (each net N_i) {
6        Rip-up RT(i), and update p(n) for all nodes n in RT(i);
7        RT(i) = Source(i);
8        for (each sink j of net N_i in decreasing Crit(i,j) order) {
9          Q(i) = RT(i);
10         while (sink(i,j) not found) { /* Wave expansion */
11           Remove lowest cost node, m, from Q(i);
12           for (all fanout nodes n of node m) {
13             Calculate t_arrival and t_leave for n;
14             Calculate B(i,n) using current T_crit;
15             Evaluate c(n);
16             Add n to the Q(i);
17           }
18         } /* Routing of one sink is finished. */
19         for (all nodes, n, in path from RT(i) to sink(i,j)) {
20           Update p(n);
21           Add n to RT(i);
22           Calculate t_arrival and t_leave for n;
23           Calculate B(i,n) using current T_crit;
24         }
25         Update Elmore delay of RT(i);
26         Update t_arrival and t_leave for all n in RT(i);
27         Update B(i,n) all n in RT(i);
28       } /* Routing of one net is finished. */
29     } /* Routing of all nets are finished. */
30     Back-annotate Elmore delays of RT(i) of all nets i into G_t;
31     Propagate timing in G_t, and compute T_crit;
32     Update B(i,n) for all nets N_i on all nodes n;
33     Update h(n) for all nodes n;
34     Update Crit(i,j);
35   } /* End of one routing iteration*/
```

Fig. 4.6: Time-multiplexing -aware timing-driven routing algorithm pseudo-code

23, 27, and 32 reflect the fact that our algorithm computes and updates occupation bitmap of wires. This has been detailed in Section 4.2.2. Lines 6 and 20 are where our algorithm computes present congestion penalties at microcycles for wires. And line 33 is where our algorithm computes historical congestion penalties at microcycles. Present and historical congestion penalties at microcycles have been detailed in Section 4.3. Line 15 is where our algorithm computes time-multiplexing -aware congestion cost for a wire and then evaluates this wire's overall cost. Time-multiplexing -aware congestion cost and the overall cost have been detailed in Section 4.4 and Section 4.5, respectively. Finally, line 4 is where our routing algorithm checks if there are any congested routing resources remaining. This has been detailed in Section 4.6.

4.8 Further Details

Section 4.2.2 has described how we compute occupation bitmap. For our algorithm relies on this occupation bitmap to determine at which microcycle(s) this signal would probably use this wire, it is desirable that the occupation bitmap is accurate. From Equation 4.5 in Section 4.2.2, we can see that occupation bitmap is a function of both signal arrival/ leave times ($t_{arrival}$ & t_{leave}) and circuit critical path delay (T_{crit}). Hence accuracy of occupation bitmap depends on that of $t_{arrival}$, t_{leave}, and T_{crit}.

Our algorithm contains some improvements relative to the VPR 5 algorithm so as to ensure that the computed $t_{arrival}$, t_{leave}, and T_{crit} values are accurate enough. This section describes these improvements.

4.8.1 Accuracy of $t_{arrival}$ and t_{leave}

In our time-multiplexing -aware routing algorithm, there are two cases when we need to compute $t_{arrival}$ and t_{leave} for a node n:

- The node n is being expanded during the wave expansion from source of net i to one of its sink s_{ij};
- The node n has been included in a routing path from source of net i to one of its sink s_{ij}.

In both cases, we compute $t_{arrival}$ and t_{leave} based on a partial routing tree. But this partial routing tree will change subsequently. In the first case, more nodes will be expanded and possibly included, at the downstream of n, in the routing path from net source to s_{ij}; in the second case, n may be included in routing paths from net source to its other sinks.

This has implications on the accuracy and validity of computed $t_{arrival}$ and t_{leave} values. In some FPGA architectures, delay of a node depends on the topology of the routing tree. For example, this is the case with the old-style VPR version 4.30 routing architectures. As a result, computed $t_{arrival}$ and t_{leave} values will become inaccurate and invalid, as the partial routing tree's topology gets changed.

To address this problem, we update $t_{arrival}$ and t_{leave} for all the nodes in the current partial routing tree, after routing for each sink is done. This ensures that the latest routing tree topology is taken into consideration, hence $t_{arrival}$ and t_{leave} values are as accurate as possible.

Note that this problem does not exist for the constant delay model. In constant delay model, delay of a node does not depend on topology of the routing tree.

Also recall that values of $t_{arrival}$ and t_{leave} depend on $T_{arrival}(source(i))$, signal arrival time at source of net i. $T_{arrival}(source(i))$ is computed by the static timing analyzer, when the timing analyzer traverses the timing graph to compute T_{crit}. Due to Pathfinder's nature of routing nets one after one, the timing analyzer does its task only after all nets are routed. Indeed, in VPR timing-driven router, the timing analyzer is triggered only once in one iteration.

Hence, we can only use $T_{arrival}(source(i))$ value computed in the last iteration. But this make computed values of $t_{arrival}$ and t_{leave} inaccurate, for $T_{arrival}(source(i))$ actually oscillates over the iterations.

To address this problem, we could call the static timing analyzer and have it compute $T_{arrival}(source(i))$ after each net is routed. This ensures that $T_{arrival}(source(i))$ is as accurate as possible. A minor problem caused by this approach is increased running time.

Fortunately, we can make use of the fact that $T_{arrival}(source(i))$ is mostly affected by critical nets. This means that we need not call the static timing analyzer after each net is routed. Instead we could call the static timing analyzer only after a critical net is routed. This saves considerable running time without compromising the accuracy of $T_{arrival}(source(i))$.

4.8.2 Accuracy of T_{crit}

According to Equation 4.5, we need to know T_{crit} value as well to compute the bitmap. Recall that the value of T_{crit} is determined by the path-based static timing analyzer. The timing analyzer traverses the timing graph and computes T_{crit}, after all nets are routed. As said in Section 4.8.1, in VPR timing-driven router, the timing analyzer is triggered only once in one iteration.

Similar to the case of $T_{arrival}(source(i))$, we can use T_{crit} value computed in the last iteration. But this value may be inaccurate, for T_{crit} value also oscillates over the iterations.

Fortunately this problem can be solved by the approach we present in Section 4.8.1 to solve the problem of $T_{arrival}(source(i))$ accuracy. When it performs the breadth-first traversal on the timing graph, the timing analyzer computes $T_{arrival}(source(i))$ as well as T_{crit}. Hence, the approach improves the accuracy of both $T_{arrival}(source(i))$ and T_{crit} values.

4.9 Analysis of Algorithm

In this section, we present analysis of our algorithm. We focus on its time complexity, memory requirement, and unroutability detection ability.

4.9.1 Time Complexity

Now we analyze timing complexity of the presented time-multiplexing -aware timing-driven routing algorithm. The algorithm is iteration-based, and the number of iterations is always capped at a fixed number in practice. As a result, it suffices to analyze complexity of one iteration in the algorithm.

By looking at the pseudo-code shown in Figure 4.6, one can observe that, one iteration consists of two parts. The first part is *netlist routing* (line 5 - 29), and the second is *post-processing* (line 30 - 34). Let's examine each of the two parts individually.

Netlist routing part consists of running a *net routing* algorithm for each net in circuit netlist. It has been shown that typical complexity of net routing algorithm

in Pathfinder is $O(k^2 logk)$ [6]. k is the number of terminals that a net has. In our presented algorithm, the extra computations introduced during net routing include:

- Computing signal arrival and leave times (line 13, 22, and 26);
- Computing occupation bitmap (line 14, 23, ad 27);

From Section 4.2.1, we know that it takes only constant time to compute signal arrival and leave times for a routing-resource node n. From Section 4.2.2, we know that it takes $O(K)$ time to compute occupation bitmap. K is the number of microcycles in a user clock cycle. However, we limit K to 8 in our work, as we will see in later chapters. As a result, we can consider that, in practice, it takes only constant time to compute occupation bitmap.

Taking the above into considerations, we can conclude that, in our presented algorithm, routing a k-terminal net typically takes $O(k^2 logk)$ time. This is the same as that in Pathfinder.

Next, let's examine the post-processing part. The main computation here is performing static timing analysis on timing graph to compute T_{crit} (line 31). The critical path method (CPM) algorithm employed and implemented in VPR 5 is re-used here. It has been shown in [27] that this algorithm has a complexity of $O(V + E)$. V is the number of vertices in the circuit timing graph, and E is the number of edges in the timing graph.

As for other operations in post-processing part, back-annotation of routing tree Elmore delay (line 30) takes $O(C)$ time, where C denotes the number of source-sink connections in circuit; Both computing occupation bitmap (line 32) and updating historical congestion penalties for all nodes (line 33) takes $O(R)$ time, where R

denotes the number of nodes in routing-resource graph.

4.9.2 Memory Requirement

For our presented algorithm, memory requirement is mainly due to FPGA routing-resource graph and circuit timing graph.

For each node in the routing-source graph, we store its physical information (for example, coordinates), connectivity information (for example, number and type of outgoing edges), timing information (for example, its resistance and capacitance), congestion information (capacity, micro occupancies, present and historical congestion penalties, etc), and some extra information needed for maze expansion. Generally memory requirement due to routing-resource graph is $O(R)$. R denotes the number of nodes in routing-resource graph.

For each vertex in timing graph, we store connectivity information (for example, number of outgoing edges) and timing information (for example, arrival time of the last input signal). For each edge in timing graph, we also store connectivity information (for example, destination timing node) and timing information (for example, delay). Generally memory requirement due to timing graph is $O(V + E)$. V is the number of vertices in the circuit timing graph, and E is the number of edges in the timing graph.

4.9.3 Unroutability Detection

Our presented algorithm uses a heuristic way to detect unroutability. Like Pathfinder algorithm, our algorithm will only declare that the given placement is un-routable on the given FPGA after 50 iterations. But this process will take a

long time. A possible future work to enhance our algorithm is to add the quick unroutability detection ability.

Chapter 5

Architecture Evaluation

In this chapter, we evaluate our proposed time-multiplexed interconnect architecture TM-ARCH. The evaluation has two major foci: (1) comparing the time-multiplexed architecture with the conventional architecture, and (2) comparing time-multiplexed architectures of different K values. Architectural evaluations of similar foci have been done by previous researchers, as part of their work on time-multiplexed architecture. In the next section, we review some results from the literature.

We evaluate interconnect architectures experimentally. We use the standard CAD flow which is widely used in the FPGA research community. We use various metrics, including minimum channel width, routing area, circuit critical path delay, and area-delay product. Section 5.2 details our experiment flow and evaluation metrics, as well as the area model, delay model, and our assumptions regarding some significant architectural details.

Table 5.1: Channel width comparison between Trimberger's architecture and XC4000E architecture. Data of the former architecture is from [35] Table 1, and data of the later is from [38] Table 14.

	Trimberger's		XC4000E	
Wire Types	Vertical	Horizontal	Vertical	Horizontal
Singles	8	8	8	8
Doubles	0	0	4	4
Quads	8	8	0	0
Octals	8	8	0	0
Long Lines	0	6	6	6

5.1 Key Results from Related Work

Trimberger *et al* revealed that their proposed time-multiplexed FPGA requires larger channel widths than its conventional counterpart [35]. Their architecture is based on Xilinx XC4000E architecture, and divides a user clock cycle into eight microcycles. It time-multiplexes both logic blocks and interconnects. Table 5.1 compares channel widths between their architecture and XC4000E architecture. Note that singles, doubles, quads, and octals in Table 5.1 correspond to length-1, length-2, length-4, and length-8 lines, respectively, in VPR terminology. We can see that, in both vertical and horizontal channels, their architecture has a larger channel width. The reason, according to them, is that signals routed from micro registers to their destinations represent additional nets that need to be routed. As a result, additional channel width is required.

Lin *et al* demonstrated that their proposed time-multiplexed interconnects could achieve channel width reduction compared with its conventional counterpart [18]. Their architecture is based on Xilinx XC4000 FPGA architecture, and, in terminology of this thesis, divides a user clock cycle into two microcycles. Their

experimental results from five benchmark circuits showed that their architecture reduced channel widths by about 30%.

Francis *et al* showed that significant channel width reduction can be achieved given that the number of time slots is large enough (i.e., K is large enough, in terminology of this thesis) [10]. Their architecture is based on Altera Stratix FPGA architecture, and allows at most 64 time slots in a user clock cycle. For example, their architecture with 24 time slots in a user clock cycle could reduce channel widths by up to 80%, compared with Stratix architecture. But their architecture also exhibited inferior timing performance. Critical path delay of circuits mapped to their architecture is 2x - 4x that of circuit mapped to Stratix. This is likely to be caused by their scheduling algorithm which actually delays signals until a free time slot is available. They also investigated the effect of number of time slots on multiplexing. Their experimental results suggested that, for their architecture, at least 8 time slots are necessary to achieve channel width reduction. If number of time slots was less than 8, their architecture actually required larger channel widths.

Table 5.2 summarizes the key results from the related architectures we reviewed above. We can see that both Lin's and Francis' architecture showed that time-multiplexed interconnects can achieve channel width reduction over their conventional counterparts. Increased channel width of Trimberger's architecture could be mainly due to the increased number of nets that need to be routed. While all the work agreed that time-multiplexed interconnects have area overhead due to extra circuitry, none of them have reported routing area results which take into consideration the two fighting factors, namely, reduced channel width and area overhead. Finally, Trimberger's and Lin's work did not report circuit critical path

Table 5.2: Key results from related architectures in the literature

	Main features			Key Results		
	Baseline architecture	Multiplexing	K	Channel Width	Routing Area	Critical Path Delay
Trimberger's [35]	Xilinx XC4000E	Both logic and interconnect	8	Larger width for horizontal and vertical channel	Unavailable	Unavailable
Lin's [18]	Xilinx XC4000	Interconnect only	2	30% reduction	Unavailable	Unavailable
Francis' [10]	Altera Stratix	Interconnect only	2-64	Up to 80% when K=24	Unavailable	2x-4x larger

delay results of their architectures, while Francis' architecture reported 2x - 4x larger critical path delays.

5.2 Experimental Methodology

5.2.1 CAD Flow

Figure 5.1 illustrates our CAD flow. We start from circuit netlists that have been technology-mapped to LUTs. First, TVPack tool in the VPR package is used to pack LUTs to clustered logic blocks. Next VPR tool is used to place the circuit. Then, based on this same placement result, two routings are performed. The timing-driven router of VPR is used to route the circuit into FPGAs with conventional interconnects. And our time-multiplexing -aware router is used to route the circuit into TM-ARCH FPGAs with time-multiplexed interconnects. We call these two routing braches as conventional routing and time-multiplexing routing, respectively. For the sake of brevity, the VPR timing-driven router and our time-multiplexing-aware router will be referred to as VPR-ROUTER and TM-ROUTER

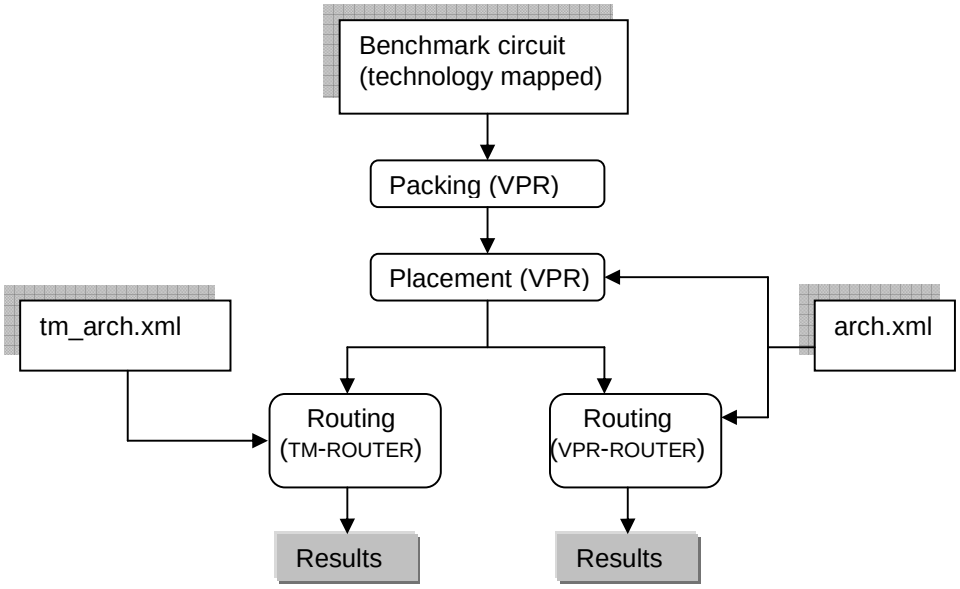


Fig. 5.1: Architecture evaluation flow.

hereafter, respectively.

5.2.2 FPGA Architectural Assumptions

In our work, we only explore homogeneous FPGA architecture. Three main issues in homogeneous FPGA architecture are logic cluster architecture, global routing architecture, and detailed routing architecture. Inside each of these three issues, there are many parameters that can be varied. The design space of FPGA architecture is so huge that we cannot possibly study time-multiplexed interconnects over the entire design space.

To make our work tractable, we first fix on a representative baseline FPGA architecture. This baseline FPGA architecture assumes conventional interconnects. We replace all the wires in the baseline FPGA with time-multiplexed wires, hence get the TM-ARCH FPGA architecture with time-multiplexed interconnects.

Table 5.3: Main features of our used baseline FPGA architecture

Architecture Parameter	**Value/ Specification**	**Comments**
LUT size	4	
Logic block size	10	
Logic block inputs	22	
Amount of bias between horizontal and vertical channels	No bias	
Uniformity of routing channels in the same direction	Uniform	
Aspect ratio	1:1	Assuming square logic blocks
Segmentation distribution	100% length 4 wires	
Switch types used	Uni-directional single driver switches	
Switch block topology	Wilton	
Switch block internal population	100%	
Connection block internal population	100%	

Baseline architecture We choose as baseline FPGA architecture the architecture defined by XML file *n10k04l04.fc15.area1delay1.cmos65nm.bptm* from iFAR [23]. Main parameters of this baseline FPGA architecture are listed in Table 5.3.

5.2.3 Evaluation Metrics

Minimum Channel Width

Minimum channel width, W_{min}, refers to the minimum number of tracks per channel required by a router to successfully route a circuit onto an FPGA. Note that in our work, we assume that channel width is uniform over all the horizontal and vertical channels. This corresponds to architectures with "no directional routing bias" and "uniform channel widths" in [6]. It has been shown in [6] that this kind of routing architecture is the most area-efficient.

Both VPR-ROUTER and TM-ROUTER can perform a binary search for W_{min}.

That is, the router repeatedly routes a circuit using different channel widths, until it finds the minimum channel width required to successfully route the circuit.

By examining and comparing achievable W_{min} given by the conventional architecture and the time-multiplexed architecture, we can evaluate how much wire saving the time-multiplexed architecture can bring in.

Routing Area

FPGA fabrics consist of two parts, logic and interconnect. Routing area, A_R, refers to the part of silicon area devoted to interconnect (or routing network).

Although it lets us quickly assess the possible wire saving brought in by the time-multiplexed architecture, W_{min} is by no means a good indicator of routing area. This is so, because transistors rather than wires determine an FPGA's routing area.

VPR-ROUTER reports FPGAs' routing area by adding all transistor areas in the routing network. It employs minimum-width transistor area model. Following this practice, we let TM-ROUTER report routing area of time-multiplexed architectures. And routing area reported by TM-ROUTER is based on the same area model. This allows us to compare routing areas of time-multiplexed architecture with that of conventional architecture.

By examining routing area of time-multiplexed architectures and comparing it with that of conventional architecture, we can evaluate the area impact of the time-multiplexed architecture.

Circuit Critical Path Delay

At the end of routing, both VPR-ROUTER and TM-ROUTER report a circuit's critical path delay (T_{crit}). The reported delay values are based on PTM 65nm CMOS technology.

Area-Delay Product

Area-delay product is obtained by multiplying an FPGA's overall area (including both logic and interconnect) by critical path delay of a circuit implemented onto this FPGA. It has been used in [6] as a metric to evaluate different FPGA architectures.

5.2.4 TM Switches Area and Delay Assumptions

We assume the implementation of TM switches shown in Figure 3.7. We further assume that multiplexers selecting from multiple configuration bits are implemented with minimum sized NMOS pass transistors [1]. Consistent with VPR 5, we assume that each bit of memory cell takes six minimum-width transistor area. The circular counter generating select signals for multiplexers can be shared by a number of TM switches. For example, a single counter provides select signals for all TM switches in a switch block. Hence we can safely ignore area of the counter.

We assume that delay of a TM switch is the same as that of its conventional counterpart. Comparing Figure 3.7 with Figure 3.5, we can see that TM switch introduces no extra logic along its critical path. Also, the extra capacitive loading seen by the buffer is minimal. Usually this buffer is strong enough to be able to

[1] Gate voltage boosting can be used to compensate voltage drop across NMOS pass transistor.

drive a wire segment spanning a number of logic blocks. So, the minimal extra capacitive loading should not increase delay of TM switch.

5.3 Experimental Results: Minimum Channel Width

In this set of experiments, for each circuit, both conventional routing and time-multiplexing routing perform binary searches to find out the minimum channel widths. More specifically, VPR-ROUTER finds the minimum channel width required to route a circuit onto the FPGA with conventional interconnects, and TM-ROUTER finds the minimum channel width required to route a circuit onto TM-ARCH FPGA with time-multiplexed interconnects. The maximum number of iterations is set to 50 for both VPR-ROUTER and TM-ROUTER.

Table 5.4 presents minimum channel width results for MCNC 20 benchmark circuits [40]. The first column of the table lists name of each benchmark circuit. The values listed under column "W_{min}" are the minimum channel widths achieved by VPR-ROUTER in conventional routing. The values listed under columns "W'_{min}" are minimum channel widths achieved by TM-ROUTER for different K values. We can see that, when K=2, TM-ARCH FPGA actually requires larger channel widths than its conventional counterpart. Averaged over the 20 circuits, the increase of minimum channel width is 14.58%. This result is counterintuitive, for we expected multiplexing wires should result in a reduction of minimum channel width.

A possible reason for the counterintuitive result is that having two microcycles in a clock cycle provides only limited opportunities for TM-ROUTER to achieve time-multiplexing of wires. Table 5.5 tabulates the percentage of wires used in the 1st and the 2nd microcycle in the final routing results for MCNC 20 benchmark

Table 5.4: Minimum channel width for different K values

	W_{min}	W'_{min} K=2	W'_{min} K=4	W'_{min} K=6	W'_{min} K=8
alu4	48	50	30	36	26
apex2	62	74	38	36	22
apex4	64	86	48	34	26
bigkey	44	44	32	32	32
clma	78	138	74	48	36
des	44	40	34	32	32
diffeq	38	36	34	32	30
dsip	38	36	30	30	30
elliptic	62	58	52	N.A.	34
ex1010	74	88	60	40	34
ex5p	68	70	48	34	22
frisc	74	84	44	40	40
misex3	54	60	42	26	22
pdc	90	106	128	60	70
s298	34	76	28	28	20
s38417	48	48	50	26	22
s38584.1	50	52	48	26	22
seq	60	74	48	26	22
spla	74	100	N.A.	52	34
tseng	46	44	40	34	32
Geo. Mean	56	64	44	34	29
Reduction	-	14.58%	-19.92%	-38.11%	-47.74%

Table 5.5: Percentages of wire used in the 1st and the 2nd microcycle for MCNC 20 benchmark circuits. Assume that a user clock cycle is divided into two microcycles.

	1st microcycle	2nd microcycle
alu4	94.59%	4.67%
apex2	97.09%	2.42%
apex4	87.35%	10.56%
bigkey	93.65%	4.77%
clma	96.50%	2.97%
des	90.08%	9.15%
diffeq	95.97%	3.80%
dsip	94.70%	4.28%
elliptic	95.35%	4.46%
ex1010	90.04%	8.67%
ex5p	82.63%	14.85%
frisc	86.87%	12.06%
misex3	93.09%	5.60%
pdc	95.09%	4.36%
s298	85.82%	13.48%
s38417	92.87%	6.60%
s38584.1	97.41%	1.88%
seq	97.50%	2.20%
spla	95.98%	3.60%
tseng	96.48%	3.34%
Geo. Mean	92.85%	5.17%

Table 5.6: Percentages of wire used in the 1st, the 2nd, the 3rd and the 4th microcycle for MCNC 20 benchmark circuits. Assume that a user clock cycle is divided into four microcycles.

	1st microcycle	2nd microcycle	3rd microcycle	4th microcycle
alu4	56.04%	29.64%	4.19%	0.45%
apex2	67.82%	25.32%	2.17%	0.21%
apex4	28.13%	56.96%	9.45%	1.00%
bigkey	61.57%	28.13%	4.77%	0.00%
clma	68.86%	25.69%	2.83%	0.12%
des	63.16%	24.61%	6.50%	1.96%
diffeq	74.31%	18.47%	3.71%	0.10%
dsip	59.70%	31.89%	4.04%	0.18%
elliptic	87.51%	7.36%	3.65%	0.73%
ex1010	23.65%	64.19%	7.98%	0.51%
ex5p	21.70%	56.98%	12.71%	1.99%
frisc	68.23%	18.50%	10.58%	1.41%
misex3	52.19%	33.60%	4.64%	0.80%
pdc	46.93%	42.92%	3.88%	0.42%
s298	63.21%	21.00%	9.76%	3.16%
s38417	65.09%	24.77%	5.83%	0.67%
s38584.1	73.30%	19.91%	1.59%	0.26%
seq	69.24%	24.66%	1.99%	0.17%
spla	49.20%	41.06%	3.31%	0.25%
tseng	88.49%	6.82%	2.35%	0.73%
Geo. Mean	55.83%	26.19%	4.49%	0.51%

circuits. Here we assume that a user clock cycle is divided into two microcycles. The data are produced in the following way. We modify VPR-ROUTER so that it will record signal arrival/ leave times at each used wire segment and write these timing values as a part of routing result output. Then we let VPR-ROUTER perform routing assuming minimum channel width listed under column "W_{min}" in Table 5.4. After that, we use a script to parse the routing result and, as per Equation 4.5, decide in which microcycle each wire segment is used. The script also counts number of wires used in each microcycle and divides the number by total number of wires used in final routing. This gives the percentages listed in Table 5.5.

Table 5.5 reveals that, in average 92.85% of the wires are used in the 1st microcycle, while only 5.17% are used in the 2nd microcycle. This translates to severely limited opportunities for TM-ROUTER to achieve time-multiplexing. As we have seen in Chapter 4, to achieve time-multiplexing, our time-multiplexing -aware routing algorithm matches a net using a wire in the 1st microcycle with a second net using the same wire in the 2nd microcycle. Table 5.5 suggests that a used wire segment is most probably used by a net in the 1st microcycle, and very unlikely used by a second net in the 2nd microcycle. As a result, our time-multiplexing -aware routing algorithm is unlikely to achieve time-multiplexing for this wire. Recall from Chapter 4 that our time-multiplexing -aware routing algorithm does not actively delay signals. This is in contrast to Francis *et al*'s scheduling algorithm, which actively delays signals to achieve more time-multiplexing [10].

For comparison, Table 5.6 tabulates used wires' distribution in microcycles when a user clock cycle is divided into four microcycles. The procedure to produce this table is similar to that for Table 5.5. In this case, the distribution between the 1st microcycle and the 2nd microcycle (55.83% versus 26.19%) is more balanced

than that in Table 5.5. This translates to more opportunities for TM-ROUTER to achieve time-multiplexing.

Besides the reason above, the fact that TM-ROUTER may have produced some noise in the routing results could also partially explain the counterintuitive result. Take **clma** for example. Figure 5.2 illustrates the channel width values tried by TM-ROUTER during the binary search for minimum channel width. For example, the first three channel widths tried by TM-ROUTER are 34, 68 (i.e., 2*34), and 136 (i.e., 2*68), respectively. Considering that W_{min} found by VPR-ROUTER is 78 (see Table 5.4), we would expect that TM-ROUTER could successfully route **clma** using a channel width of 136. However, as shown in Figure 5.2, it failed, and subsequently tried an even larger value of 272 (i.e., 2*136).

Now let us examine one detailed aspect of TM-ROUTER's routing with the channel width 136. Figure 5.3 illustrates the number of remaining congested routing-resource nodes after each iteration during the routing. Recall that TM-ROUTER declares routing successful and exits when no congested nodes remain after an iteration. From Figure 5.3, it can be observed that throughout the first 20 iterations the number of remaining congested nodes decreases steadily (from over 6,000 down to around 30). But after 30 iterations, the number, instead of decreasing to zero, oscillates between 1 and 10. Although they account for a tiny percent of all the nodes (around 100K in this case), these remaining congestions prolong the routing, and eventually cause the routing to fail (for we limit the number of iteration to 50).

The used channel width 136 is sufficiently large, compared with W_{min} achieved by VPR-ROUTER. Hence TM-ROUTER's difficulty to resolve the remaining congestion, as observed in Figure 5.3, is most probably due to some defects in TM-ROUTER.

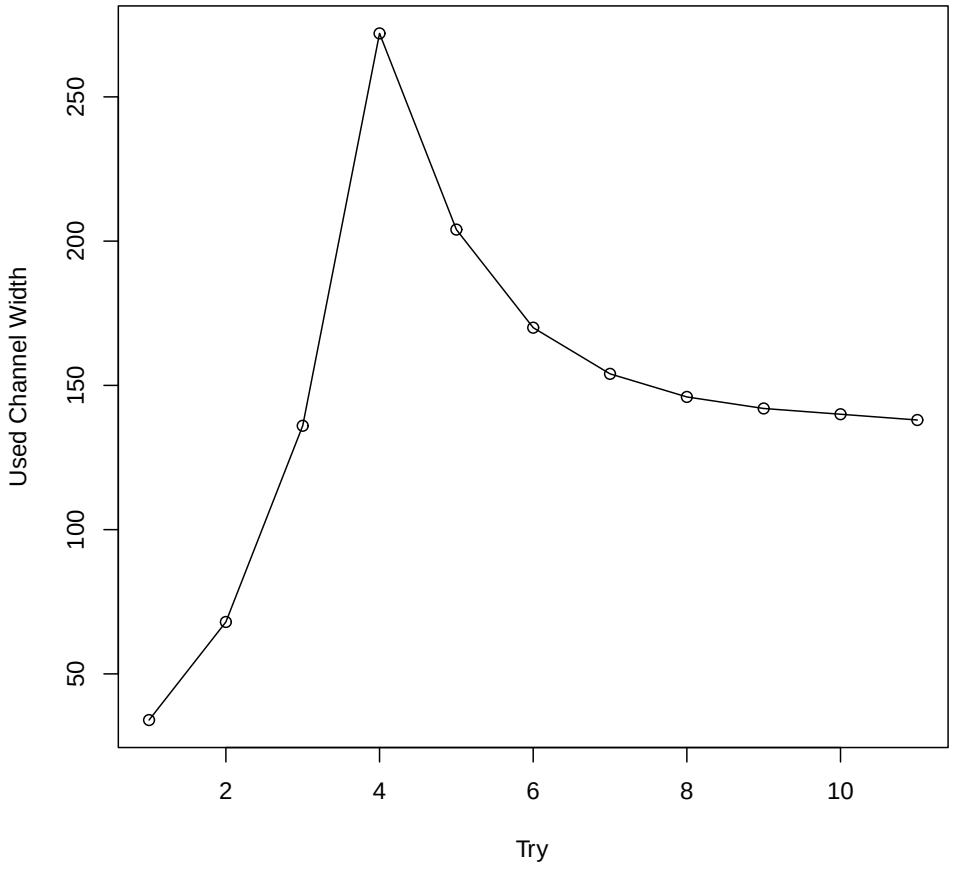


Fig. 5.2: Channel width values that are tried by TM-ROUTER *during the binary search.*

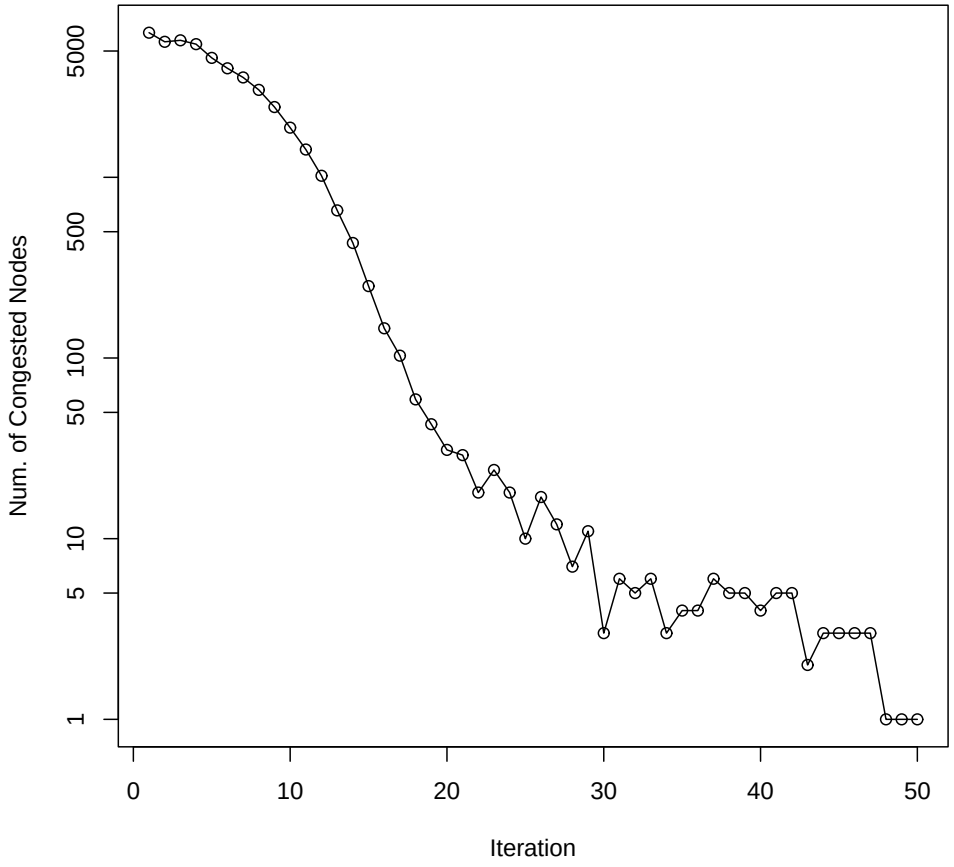


Fig. 5.3: Number of remaining congested nodes after each iteration. (Channel width $W = 136$; y-axis shown in logarithmic scale)

Unfortunately, we are not able to solve this problem in this work.

From K=4 onwards, we see that TM-ARCH FPGA generally requires smaller channel widths. This is in accordance with our expectation that multiplexing wires will result in a reduction of minimum channel width. And the reduction of minimum channel width is more significant when a clock cycle is divided into more microcycles: the percentages of minimum channel width reduction are 19.92% (K=4), 38.11% (K=6), and 47.74% (K=8), respectively. This is due to the nature of time-multiplexing: a same wire can be shared among more nets as K increases, hence less number of wires will be required to route a given number of nets.

5.4 Experimental Results: Routing Area

In the previous section, we have reported the impact of time-multiplexed interconnect on minimum channel width. In this section, we report the impact on routing area. All results presented in this section are reported by the routing tool (VPR-ROUTER and TM-ROUTER) assuming minimum channel width (W_{min} and W'_{min}) for each benchmark circuit. The unit is minimum-width transistor area.

Table 5.7 presents the routing area results. The values listed under column "A_R" are area values reported by VPR-ROUTER in conventional routing. The values listed under columns "A'_R" are area values reported by TM-ROUTER for different K values. We can see that time-multiplexed interconnects consume more routing area that conventional interconnects. For the cases of K=2, 4, 6, and 8, the area overhead, averaged over MCNC20 circuits, is 44.47%, 45.64%, 48.41%, and 59.47%, respectively.

The 44.47% routing area overhead in the case of K=2 can be attributed to

Table 5.7: Routing area reported by routers, assuming minimum channel width. Unit is minimum-width transsitor area.

	A_R	$A_R^{'}$ K=2	$A_R^{'}$ K=4	$A_R^{'}$ K=6	$A_R^{'}$ K=8
alu4	8.04e+05	1.08e+06	9.57e+05	1.46e+06	1.31e+06
apex2	1.20e+06	1.79e+06	1.36e+06	1.68e+06	1.31e+06
apex4	9.19e+05	1.49e+06	1.25e+06	1.19e+06	1.12e+06
bigkey	8.31e+05	1.06e+06	1.16e+06	1.50e+06	1.84e+06
clma	5.95e+06	1.30e+07	1.08e+07	9.26e+06	8.72e+06
des	1.07e+06	1.29e+06	1.62e+06	1.95e+06	2.39e+06
diffeq	6.57e+05	7.99e+05	1.08e+06	1.30e+06	1.50e+06
dsip	7.57e+05	9.22e+05	1.11e+06	1.42e+06	1.74e+06
elliptic	2.39e+06	2.78e+06	3.68e+06	N.A.	3.97e+06
ex1010	3.35e+06	5.03e+06	5.17e+06	4.48e+06	4.80e+06
ex5p	8.09e+05	1.07e+06	1.06e+06	1.00e+06	8.19e+05
frisc	2.52e+06	3.57e+06	2.78e+06	3.36e+06	4.14e+06
misex3	7.68e+05	1.11e+06	1.09e+06	9.14e+05	9.71e+05
pdc	4.00e+06	5.92e+06	1.04e+07	6.74e+06	9.60e+06
s298	7.06e+05	1.83e+06	1.02e+06	1.32e+06	1.21e+06
s38417	3.08e+06	3.95e+06	5.97e+06	4.19e+06	4.47e+06
s38584.1	3.24e+06	4.28e+06	5.70e+06	4.19e+06	4.47e+06
seq	1.18e+06	1.79e+06	1.69e+06	1.24e+06	1.31e+06
spla	2.79e+06	4.69e+06	N.A.	4.81e+06	3.97e+06
tseng	5.41e+05	6.69e+05	8.86e+05	1.00e+06	1.15e+06
Geo. Mean	1.44e+06	2.08e+06	2.10e+06	2.14e+06	2.30e+06
Overhead	-	44.47%	45.64%	48.41%	59.47%

two factors. The first is the larger channel width required. As we have seen in the previous section, when K=2, TM-ARCH requires 14.58% larger channel widths than its conventional counterpart. A larger channel width means more routing transistors used in connection blocks and switch blocks, hence larger routing area. The second is that a TM switch consumes more area than a conventional switch.

Routing area overhead exhibited by TM-ARCH when K=4, 6, and 8, however, is entirely due to the fact that TM switches consume more area. In fact, as we have seen in the previous section, TM-ARCH requires smaller channel widths. As TM switches incurs significant area overhead, TM-ARCH requires larger routing area, in spite of the reduction in channel width.

5.5 Experimental Results: Circuit Critical Path Delay

In this section, we look at the timing metric, namely, circuit critical path delay. In real life applications, FPGAs routing resources tend to be sufficient compared with the requirements by the user designs, so as to improve circuit timing performance. As a result, delay values presented in this section are based on low-stress routing. That is, given a circuit, the router performs routing using a channel width which is 20% more than the minimum channel width. Table 5.8 presents the channel width values used in the low-stress routing. The minimum channel width values we have presented in Table 5.4 at Section 5.3.

Table 5.9 presents the critical path delay values reported from low-stress routing for MCNC 20 benchmark circuits. The "T_{crit}" column corresponds to conventional architecture, while the "T'_{crit}" columns to TM-ARCH architecture. It can be

Table 5.8: Channel width values used in low-stress routing.

	W_{ls}	W'_{ls} K=2	W'_{ls} K=4	W'_{ls} K=6	W'_{ls} K=8
alu4	58	60	36	44	32
apex2	74	88	46	44	26
apex4	76	104	58	40	32
bigkey	52	52	38	38	38
clma	94	166	88	58	44
des	52	48	40	38	38
diffeq	46	44	40	38	36
dsip	46	44	36	36	36
elliptic	74	70	62	N.A.	40
ex1010	88	106	72	48	40
ex5p	82	84	58	40	26
frisc	88	100	52	48	48
misex3	64	72	50	32	26
pdc	108	128	154	72	84
s298	40	92	34	34	24
s38417	58	58	60	32	26
s38584.1	60	62	58	32	26
seq	72	88	58	32	26
spla	88	120	N.A.	62	40
tseng	56	52	48	40	38
Geo. Mean	66	76	53	41	35
Reduction	-	14.99%	-19.86%	-37.74%	-47.89%

Table 5.9: Circuit critical path delays reported from low-stress routing (In unit of nano-seconds).

	T_{crit}	T'_{crit} K=2	T'_{crit} K=4	T'_{crit} K=6	T'_{crit} K=8
alu4	3.31	3.31	3.31	3.23	3.45
apex2	3.90	3.69	3.69	3.69	3.65
apex4	3.80	3.13	3.13	3.20	3.20
bigkey	1.80	1.87	1.80	1.80	1.80
clma	6.74	N.A.	6.63	6.70	6.70
des	2.86	N.A.	2.78	2.85	2.85
diffeq	4.44	4.51	4.37	4.44	4.37
dsip	1.73	1.73	1.80	1.80	1.80
elliptic	6.22	5.52	5.66	N.A.	5.37
ex1010	4.42	4.49	4.49	N.A.	4.42
ex5p	3.55	3.30	3.34	3.23	3.37
frisc	7.63	7.46	7.42	7.42	7.49
misex3	3.13	3.13	3.06	3.20	3.27
pdc	5.26	4.60	4.49	4.49	4.49
s298	N.A.	N.A.	6.14	6.17	6.07
s38417	4.68	4.47	4.47	4.61	4.40
s38584.1	3.71	3.99	N.A.	3.64	3.65
seq	3.13	3.13	3.06	3.20	3.06
spla	4.46	4.14	N.A.	4.14	4.07
tseng	4.43	4.43	4.43	4.43	4.43
Geo. Mean	3.90126	3.71999	3.83374	3.7541	3.85156
Reduction	-	-4.65%	-1.73%	-3.77%	-1.27%

observed that TM-ARCH achieves a 4.65% reduction in critical path delay, when K is 2. However, there is nothing remarkable here, for TM-ARCH has used, in average, 14.99% larger channel widths than the conventional architecture. See Table 5.8.

In the cases of K=4, 6, and 8, TM-ROUTER uses 19.86%, 37.74%, and 47.89% smaller channel widths, respectively. And TM-ROUTER achieves modest reduction in critical path delay: 1.73%, 3.77%, and 1.27%, respectively.

5.6 Experimental Results: Area-Delay Product

Finally we examine the metric of area-delay product. For each circuit, we multiply the critical path delay by FPGA's total area. The delay and area values are reported by VPR-ROUTER and TM-ROUTER from low-stress routing. FPGA's total area is the sum of logic area and routing area. Since VPR-ROUTER and TM-ROUTER report both logic and routing area in unit of minimum-width transistor area, the total area values are also in unit of minimum-width transistor area. The unit of delay values is nano-seconds (ns).

Figure 5.4 plots the area-delay product results. Each data point is the geometric average value over 20 MCNC benchmark circuits. The x-axis is the value of K, the number of microcycles in a user clock cycle. Here K being 1 is used to refer to conventional architecture.

It can be observed from Figure 5.4 that TM-ARCH FPGAs exhibit larger area-delay products. For K=2, 4, 6, and 8, the increase of area-delay product is 6.66%, 10.27%, 10.24%, and 23.12%, respectively.

In Section 5.5, TM-ARCH has been shown to be able to achieve modest reduction in critical path delay. However, the significant area overhead of TM-ARCH

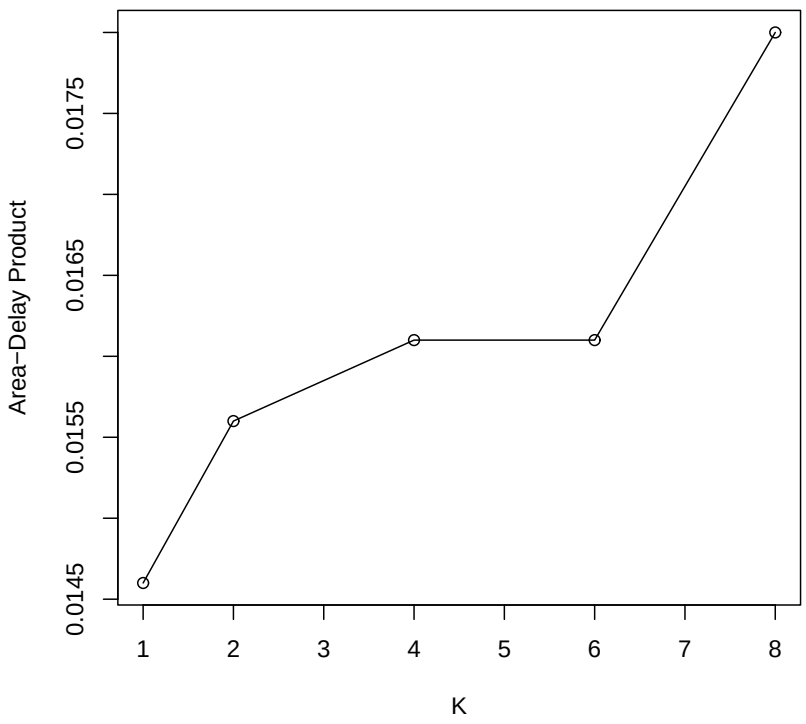


Fig. 5.4: Area-delay product results averaged over 20 MCNC benchmark circuits versus K.

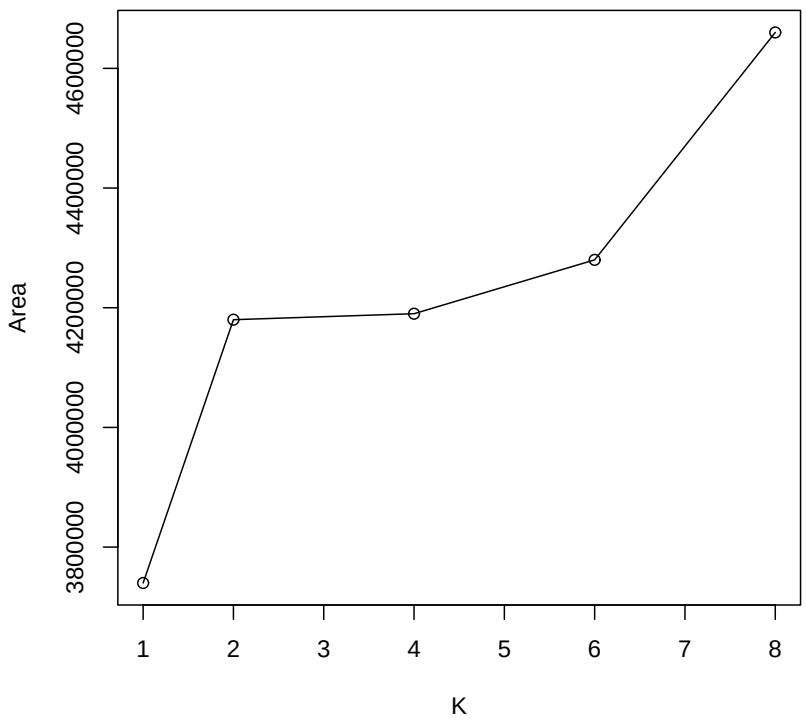


Fig. 5.5: Total area results averaged over 20 MCNC benchmark circuits versus K.

architecture outweighs the delay reduction. As a result, TM-ARCH produces larger area-delay products. Figure 5.5 plots the total area reported by VPR-ROUTER/ TM-ROUTER for low-stress routing. Again, each data point is the average value over 20 MCNC circuits. And K being 1 is used to refer to conventional architecture. It can be seen that TM-ARCH architecture's area overhead is indeed significant. In the cases of K=2, 4, 6, and 8, the overhead, compared with the conventional architecture, is 11.86%, 12.21%, 14.56%, and 24.71%, respectively.

Chapter 6

Partial Depopulation of Time-Multiplexed Interconnects

6.1 Motivation

Two observations motive us to partially depopulate time-multiplexed interconnects.

- Time-multiplexed interconnects require TM switches. Compared with their conventional counterparts, TM switches incur significant area overhead. By partially depopulating time-multiplexed interconnects, we could have less TM switches. Hence, overhead due to TM switches can be reduced.

- The actual utilization of time-multiplexed interconnects is low. In the final routing results, most of the wires are not time-multiplexed. Although they can be multiplexed, these wires are used in only one net's route.

6.2 Overview

Figure 6.1 illustrates the architecture of our proposed FPGA architecture with partially populated time-multiplexed interconnects. It is based on the architecture we have presented in Chapter 3. Compare with Figure 3.1. At the architecture level, the difference is that this architecture partially depopulates time-multiplexed wires. That is, not all the wires in the routing channels can be time-multiplexed: some wire can be time-multiplexed (or, are time- multiplex-able), while the others, just like wires found in conventional FPGAs, cannot. At the circuit level, the difference is that this architecture partially depopulates TM switches. In connection and switch blocks, some switches are TM switches, while the others are conventional switches. Figure 6.1 illustrates the co-existence of TM switches and conventional switches. Again, a TM switch is denoted by a filled circle, while a conventional switch by a void circle.

Figure 6.2 shows a portion of a routing channel in this architecture. This channel contains both multiplex-able wires (shown in bold lines) and conventional wires. In this work, we assume that a routing track consists entirely of either multiplex-able wires or conventional wires. A routing track consisting of multiplex-able wires is called a multiplex-able track. Otherwise, it is called a conventional track. For example, the routing channel illustrated in Figure 6.2 consists of four conventional tracks and one multiplex-able track.

6.3 Multiplex-able Track Population

We define an architectural parameter a, *multiplex-able track population*. This parameter specifies multiplex-able tracks' proportion in a routing channel. Take

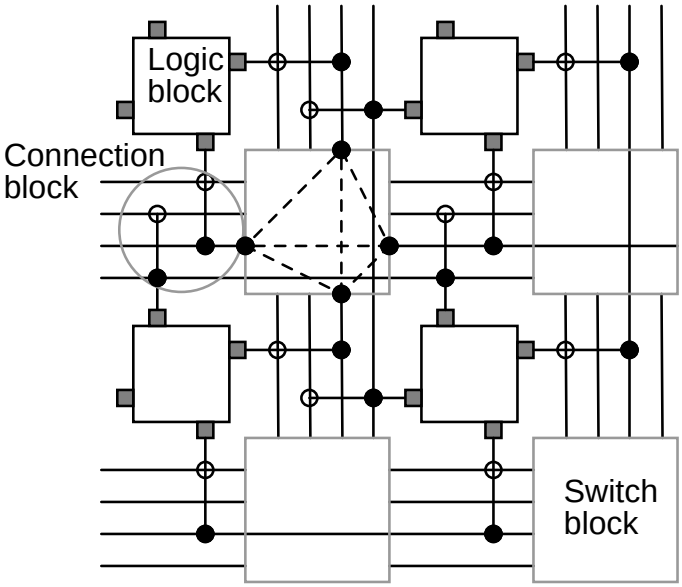


Fig. 6.1: Proposed architecture of FPGAs with partially depopulated time-multiplexed interconnects.

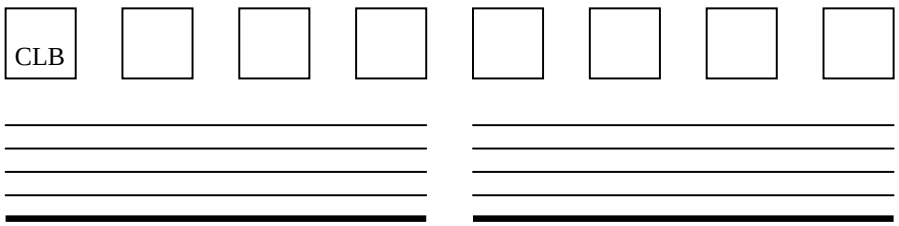


Fig. 6.2: A routing channel consisting of four conventional tracks and one multiplex-able tracks. All wire segments span four logic blocks.

Table 6.1: Switch of choice to implement the configurable connection from wire segment w_A to wire segment w_B

Is w_A multiplex-able?	Is w_B multiplex-able?	Switch of choice
No	No	Conventional
No	Yes	TM
Yes	No	TM
Yes	Yes	TM

the channel shown in Figure 6.2. Out of the total five tracks, one is multiplex-able. That is, multiplex-able track population of this channel is 0.2 or 20%.

6.4 Co-Existence of TM Switches and Conventional Switches

In this architecture with partially populated time-multiplexed interconnects, TM switches and conventional switches co-exist. Whether a TM switch or a conventional switch should be used to implement the configurable connection from a wire segment w_A to a wire segment w_B depends. Table 6.1 lists all the four possible scenarios and the corresponding switch of choice, based on whether w_A and w_B are multiplex-able or not.

It is easy to understand that a conventional switch is sufficient if both wires are conventional wires, and a TM switch must be used if both wires are multiplex-able. These two scenarios correspond to the conventional FPGA architecture and our proposed TM-ARCH architecture presented in Chapter 3, respectively.

When a conventional wire w_A connects to a multiplex-able wire w_B, a TM switch should be used. Figure 6.3(a) helps understand why this is so. In this case,

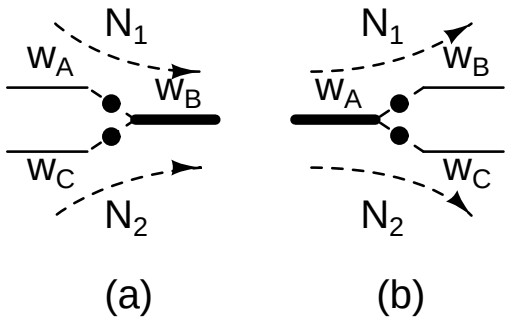


Fig. 6.3: A TM switch should be used for (a) a connection from a conventional wire w_A to a multiplex-able wire w_B, and (b) a connection from a multiplex-able wire w_A to a conventional wire w_B.

w_B is included in routes of two nets, N_1 and N_2. Connection w_A-w_B is for N_1, and w_C-w_B for N_2. At any given time in a clock cycle, only one of the two connections can be on. Hence, connection w_A-w_B should be able to turn off, when w_C-w_B turns on. A conventional switch cannot support this function. As a result, a TM switch is needed.

Figure 6.3(b) helps understand why a TM switch should be used for the connection from a multiplex-able wire w_A to a conventional wire w_B. In this case, wire w_A is time-multiplexed by two nets N_1 and N_2, and only one of the two connections (w_A-w_B and w_A-w_C) can be on at any given time. Hence, connection w_A-w_B should be able to turn off, when w_A-w_C turns on. A conventional switch cannot support this function. As a result, a TM switch is needed. Also note that, after the connection w_A-w_B turns off, the TM switch can serve as the driver so that w_B would not be floating. We have seen TM switch's latching capability in the previous chapter.

6.5 Architecture Evaluation

In this section, we evaluate the proposed architecture with partially populated time-multiplexed interconnects. The two major foci of our evaluation are: (1) comparing time-multiplexed interconnects with different values of multiplex-able track population, and (2) comparing partially populated time-multiplexed interconnect with conventional interconnect architecture. We do the evaluation by exploring two architectural parameters, namely, the number of microcycles in a user clock cycle (K) and multiplex-able track population (a).

6.5.1 Key Results from Related Work

In the literature, the only work that has investigated an FPGA interconnect architecture containing both multiplex-able wires and conventional wires is [10] by Francis *et al.* Their FPGA interconnect architecture with multiplex-able wires is based on Altera Stratix interconnect architecture. Unlike our architecture which uniformly deploys multiplex-able wires on a track basis in routing channels, their architecture uniformly deploys multiplex-able wires on a switch block basis. One multiplex-able wire deployed at a switch block means that this wire's left or lower endpoint is at this switch block. Since their work consider wires of six different types [1], they use an array notation $[x_1, x_2, x_3, x_4, x_5, x_6]$ to represent number of multiplex-able wires deployed at a switch block [2]. For example, $[2, 1, 1, 1, 1, 1]$

[1] The six different wire types are from Stratix interconnect architecture, which has length-4, length-8, and length-24 wires in horizontal channels, and length-4, length-8, and length-16 wires in vertical channels.

[2] One wire at a switch block, as meant in Francis' work, corresponds to L wires in the routing channel. L is the length of the wire segment type. This is so, because Stratix architecture staggers wire segments, and this notation counts only wire segments with their left or lower endpoints at a particular switch block.

means that, at a switch block, there are two multiplex-able wires of the first type, one of the second type, one of the third type, one of the fourth type, one of the fifth type, and one of the sixth type. They explore interconnect architectures containing both multiplex-able wires and conventional wires by varying the values of individual elements in the array notation. In their work, the authors constrained that their time-multiplexed interconnect architecture contains at least one multiplex-able wire at a switch block for each wire type (i.e., $[1, 1, 1, 1, 1, 1]$).

Their experimental results showed that only a few multiplex-able wires make a very large difference to the channel width required to successfully route a user design. The majority of channel width reduction is achieved by having only a few multiplex-able wires at a switch block (for example, $[1, 1, 1, 1, 1, 1]$ and $[2, 2, 1, 1, 1, 1]$). In terminology of this thesis, $[1, 1, 1, 1, 1, 1]$ is approximately equivalent to a between 0.15 and 0.22; and $[2, 2, 1, 1, 1, 1]$ is approximately equivalent to a between 0.17 and 0.25. Notice that this observation is based on their assumption that a user clock cycle is divided into at least 24 time slots. That is, $K >= 24$, in terminology of this thesis.

6.5.2 Experimental Methodology

CAD Flow

Figure 6.4 illustrates our CAD flow. Similar to the flow we used in Chapter 5, we start from circuit netlists that have already been technology-mapped to 4-input LUTs. First, TVPack tool in VPR package is used to pack LUTs to clustered logic blocks. Next, VPR placement tool is used to place the circuit. Then, the same placement result is used for two separate branches of routing. On the left branch,

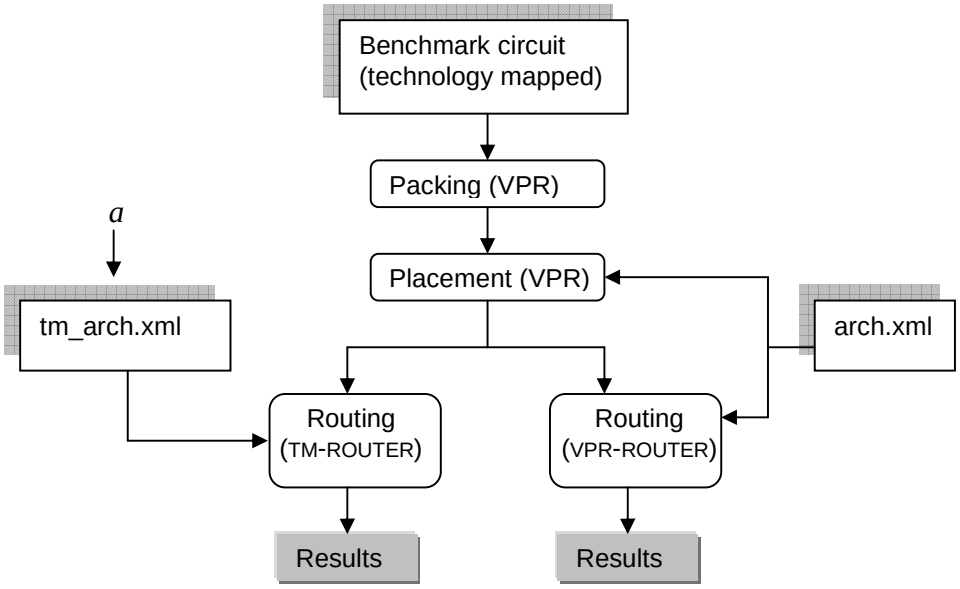


Fig. 6.4: Architecture evaluation flow.

our time-multiplexing -aware router is used to route a circuit into FPGAs with time-multiplexed interconnects; and on the right branch, VPR timing-driven router is used to route a circuit into FPGAs with conventional interconnects. Consistent with Chapter 5, we refer to these two routing braches as time-multiplexing routing and conventional routing, respectively. And we refer to our time-multiplexing -aware router and VPR timing-driven router as TM-ROUTER and VPR-ROUTER, respectively.

FPGA Architectural Assumptions

Similar to the approach in Chapter 5, we fix on a baseline FPGA architecture, which assumes conventional interconnects. We choose this baseline architecture to be the same as that in Chapter 5. That is, the architecture defined by XML file *n10k04l04.fc15.area1 delay1.cmos65nm.bptm*. Main features of this architec-

ture can be found in Table 5.3. For the sake of brevity, we refer to this baseline architecture as ARCH, hereafter.

To get FPGA architectures with time-multiplexed interconnects, we replace a portion of tracks in each routing channel of the baseline FPGA with multiplex-able tracks. We use the parameter a to denote this portion. For the sake of brevity, we refer to this architecture with time-multiplexed interconnects as TM-ARCH(a).

Evaluation Metrics

In our evaluation of partially populated time-multiplexed interconnects, we use the same four metrics that we used in Chapter 5. They are minimum channel width, routing area, circuit critical path delay, and area-delay product.

6.5.3 Experimental Results

Minimum Channel Width

In this set of experiments, for each circuit, both conventional routing and time-multiplexing routing perform binary searches to find out minimum channel width required. Specifically, in conventional routing, VPR-ROUTER finds out the minimum channel width required to route the circuit into architecture ARCH. And in time-multiplexing routing, TM-ROUTER finds out the minimum channel width required to route the circuit into architecture TM-ARCH(a). The maximum number of iterations is set to 50 for both VPR-ROUTER and TM-ROUTER.

To compare time-multiplexed interconnects with different values of multiplex-able track population, we sweep a by using four values: 0.1, 0.2, 0.5, and 1.0. Note that a=1.0 means that all tracks in the routing channels are multiplex-able. Hence

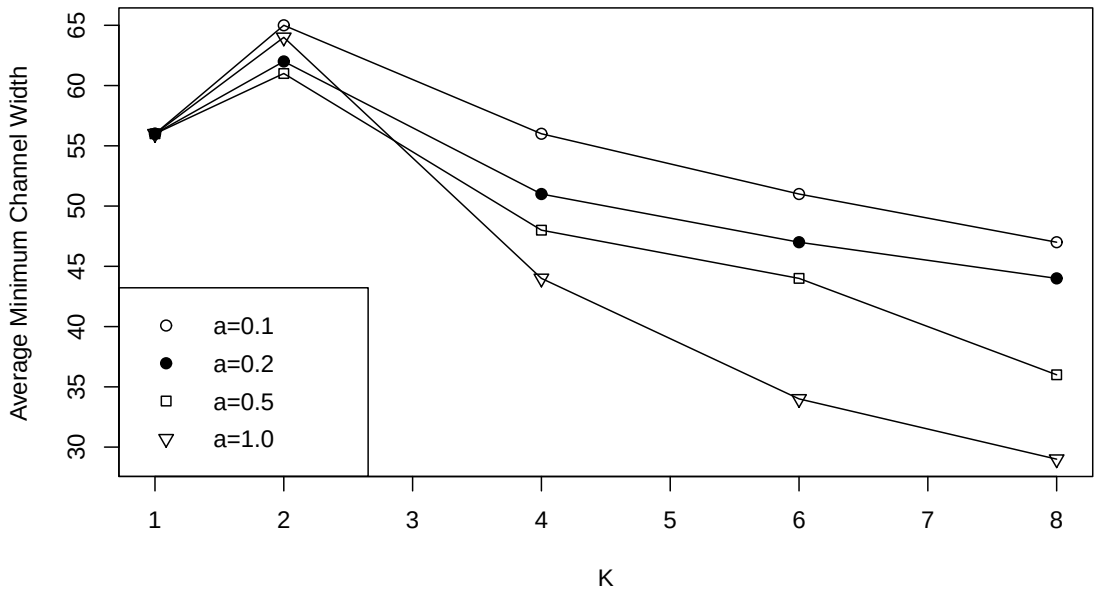


Fig. 6.5: Average minimum channel widths versus multiplex-able track population, a, and number of microcycles in a user clock cycle, K.

the architecture TM-ARCH(a) with a=1.0 is equivalent to the architecture we have presented in Chapter 3.

Figure 6.5 presents minimum channel width results. The x-axis is number of microcycles in a user clock cycle. The y-axis is the achieved minimum channel width. Each curve corresponds to a different value of parameter a, and plots average minimum channel width versus K. Each data point is the geometric average over 20 MCNC benchmark circuits. Note that K=1 denotes conventional interconnect architecture ARCH. That is, there are no multiplex-able tracks at all. So this architecture is insensitive to the value of parameter a. As a result, left end-points of all four curves are at the same coordinate.

Table 6.2: Average minimum channel width reduction of architecture TM-ARCH(a) *at different K values.*

a	K=1	K=2	K=4	K=6	K=8
0.1	-	16.88%	0.90%	-8.79%	-14.99%
0.2	-	10.85%	-7.27%	-16.01%	-21.07%
0.5	-	10.18%	-12.89%	-21.28%	-35.77%
1.0	-	14.58%	-19.92%	-38.11%	-47.74%

Table 6.2 tabulates the reduction of average minimum channel width achieved by TM-ARCH(a) at different K values. A minus percentage means that TM-ARCH(a) achieves smaller average minimum channel width than ARCH. A positive percentage means that TM-ARCH(a) requires larger average minimum channel width. For example, Table 6.2 shows that, with K=2, TM-ARCH(0.1) requires 16.88% larger average minimum channel width than ARCH. Note that K=1 denotes conventional interconnect architecture ARCH.

A first observation from Figure 6.5 is the poor performance of time-multiplexed interconnects TM-ARCH(a) when K equals to 2. In this case, TM-ARCH(a) requires larger minimum channel widths than its conventional counterpart. For multiplexable track population of 0.1, 0.2, 0.5, and 1.0, the increase of minimum channel widths is 16.88%, 10.85%, 10.18%, and 14.58%, respectively.

From K=4 onwards, TM-ARCH(a) begins to show their advantages by achieving smaller minimum channel widths. Two observations here are (1) the minimum channel width improvement of architecture TM-ARCH(a) is greater as a increases; (2) the improvement of TM-ARCH(a) is always bounded by the architecture TM-ARCH(a) with a being 1.0, i.e., the FPGA architecture with fully-populated time-multiplexed interconnects. This is reasonable, because a larger a means more

Table 6.3: Routing area overhead of TM-ARCH(a) *at different K values.*

a	K=1	K=2	K=4	K=6	K=8
0.1	-	19.91%	14.99%	13.36%	14.24%
0.2	-	19.22%	18.60%	22.92%	31.78%
0.5	-	29.81%	38.64%	46.05%	53.95%
1.0	-	44.47%	48.78%	51.61%	59.47%

multiplex-able wires available in the architecture. This in turn gives TM-ROUTER more opportunities to achieve multiplexing, hence reducing channel width.

If we focus on the individual curve and look at its trend, we can have another observation: the minimum channel width improvement of architecture TM-ARCH(a) is greater as K increases from 4 to 8.

Routing Area

This section reports routing area results collected in the same set of experiments which produces minimum channel width results reported in the previous section. Therefore, area values reported here correspond to FPGA architectures with their channel widths set to minimum channel widths.

Figure 6.6 presents the routing area results. The x-axis is number of micro-cycles in a user clock cycle. The y-axis is routing area per logic tile. Here we use routing area per logic tile as a normalized metric, because it allows averaging of results from circuits of different sizes [3]. Each curve corresponds to a different value of parameter a, and plots average routing area per tile versus K. Each data point is the geometric average over 20 MCNC benchmark circuits. Note that K=1

[3]This metric is recommended and used by Betz *et al* in their work on FPGA detailed routing architecture. See page 158 of their book

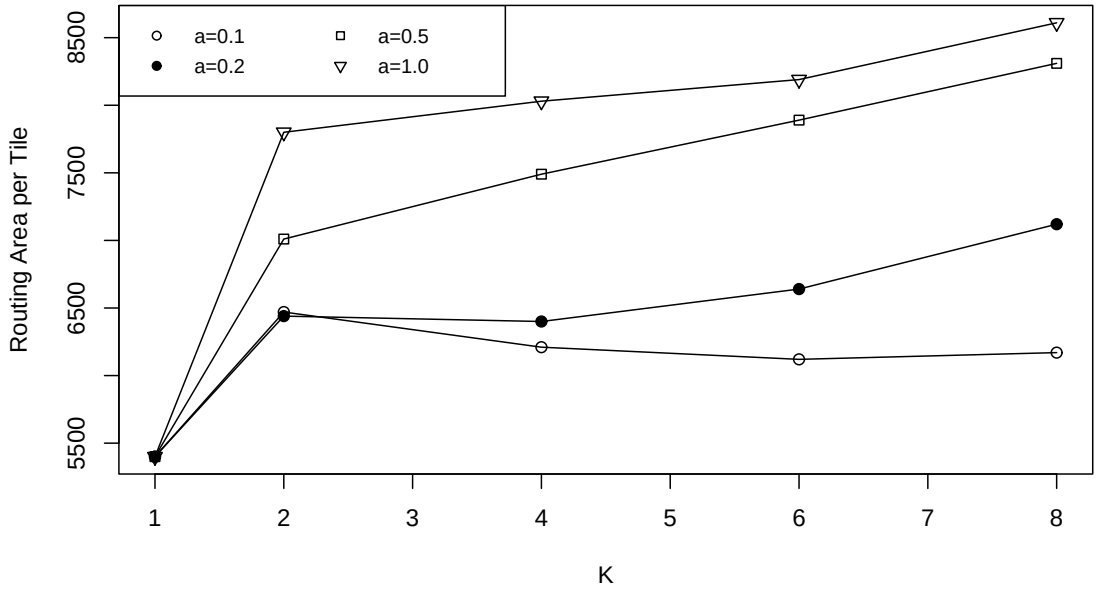


Fig. 6.6: Routing area per logic tile versus multiplex-able track population, a, and number of microcycles in a user clock cycle, K.

denotes conventional interconnect architecture ARCH.

Table 6.3 tabulates routing area overhead of TM-ARCH(a) compared with ARCH. For example, Table 6.3 shows that, when K=2, TM-ARCH(0.1) requires 19.91% more routing area. Also here K=1 column denotes conventional interconnect architecture ARCH.

First, it can be observed that, compared with ARCH, time-multiplexed interconnect architectures TM-ARCH(a) invariably bring in routing area overhead. When K=2, routing area overhead of TM-ARCH(a) with a being 0.1, 0.2, 0.5, and 1.0 is 19.91%, 19.22%, 29.81%, and 44.47% (See Table 6.3). This area overhead is due to two reasons: larger channel width and more expensive TM switches. One can refer Table 6.2 for larger channel widths required by TM-ARCH(a) when K equals to 2.

From K=4 onwards, TM-ARCH(a) still exhibits at least 13% routing area overhead, in spite of smaller channel widths shown in Table 6.2. This means that, from the standpoint of routing area, time-multiplexed interconnect architecture's overhead outweighs its superiority of reducing channel widths. This can be further verified when we compare the four curves in Figure 6.6. The larger value a is of, the larger the overhead is. Routing area overhead of TM-ARCH(a) is upper-bounded by that of TM-ARCH(a) with a being 1.0. We have seen in Section 6.5.3 that TM-ARCH(1.0) has the smallest channel width. But still it consumes the most routing area.

When we examine the trend of each individual curve in Figure 6.6, the curve corresponding TM-ARCH(0.1) is close to flat from K=4 onwards. A possible explanation is as follows: when K increases, area overhead due to TM switches naturally increases. But this is almost completely offset by larger reduction of channel width. As a result, overall routing area overhead is held constantly at around 14%. This

property may make TM-ARCH(0.1) a particularly interesting architecture: one can have tradeoff between more expensive TM switches and smaller channel width, given that the overall 14% routing area overhead is acceptable.

The other three curves corresponding architectures TM-ARCH(a) with a=0.2, 0.5, and 1.0 generally exhibit a trend that, routing area overhead increases as K increases. Given that these same architectures have smaller channel widths as K increases (see Figure 6.5), we may here further conclude that time-multiplexed interconnect architecture's overhead outweighs its superiority of reducing channel widths.

Circuit Critical Path Delay

In the second set of experiments, low-stress routing, instead of minimum channel width routing, is performed on each benchmark circuit. From this low-stress routing, we collect critical path delay and area delay product results. We do in this way, because, as we have explained in section 5.5, real-life applications usually have slightly abundant routing resources to achieve better performance. We report critical path delay results in this section, and area delay product in the next section.

For each circuit, its low-stress routing assumes a channel width which is 20% larger than the minimum channel width. Recall that the minimum channel width has been found out in the previous binary-search routing. The 20% percentage applies to both conventional routing and time-multiplexing routing. Take circuit **alu4** as an example. Table 6.4 lists its minimum channel widths and low-stress channel widths for different combinations of a and K. W_{min} is determined from binary-search, and W_{ls} is derived from W_{min}.

Table 6.4: Minimum channel width and low-stress channel width of circuit ***alu4*** *versus a and K. W_{min} is determined from binary-search routing. W_{ls} is derived from W_{min}.*

	K=1		K=2		K=4		K=6		K=8	
	W_{min}	W_{ls}	W_{min}	W_{ls}	W_{min}	W_{ls}	W_{min}	W_{ls}	W_{min}	W_{ls}
a=0.1	48	58	58	70	46	56	44	52	38	46
a=0.2	48	58	52	62	46	56	40	48	34	40
a=0.5	48	58	54	64	36	44	30	36	28	34
a=1.0	48	58	50	60	30	36	36	44	26	32

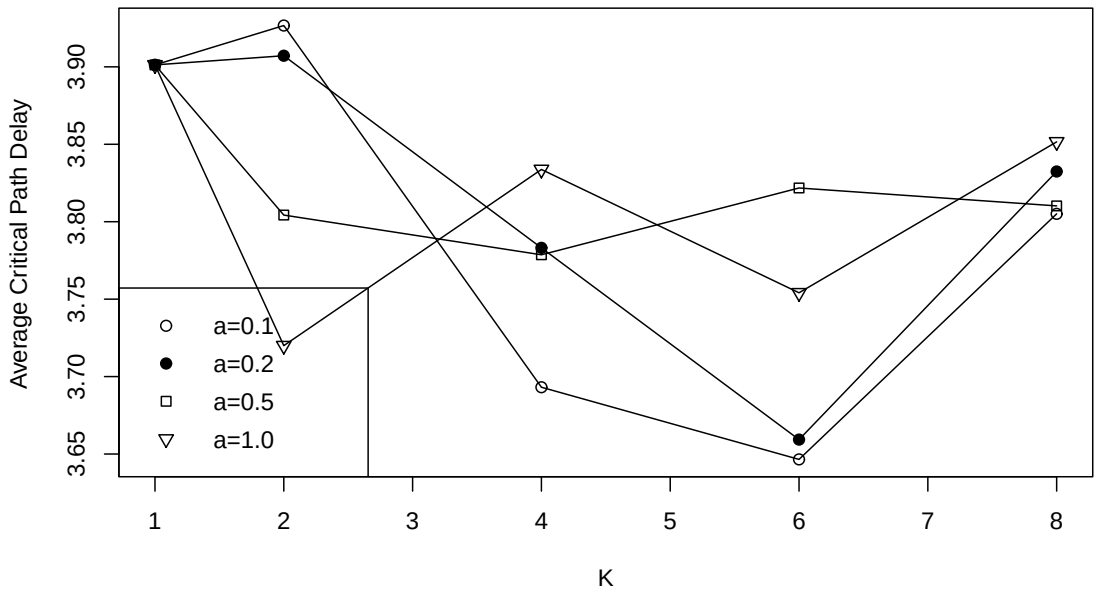


Fig. 6.7: Average circuit critical path delay versus multiplex-able track population, a, and number of microcycles in a user clock cycle, K.

Figure 6.7 presents the circuit critical path delay results. The x-axis is number of microcycles in a user clock cycle. The y-axis is the achieved critical path delay in unit of nano-seconds. Each curve corresponds to a different value of parameter a, and plots average critical path delay versus K. Each data point is the geometric average over 20 MCNC benchmark circuits. Note that K=1 denotes conventional interconnect architecture ARCH.

When K=2, compared with conventional interconnect architecture ARCH, TM-ARCH(0.5) improves critical path delay by 2.49%, and TM-ARCH(1.0) by 4.65%. This improvement is largely due to the fact that larger channel widths are assumed. Table 6.5 lists average low-stress channel width for different combinations of K and a. When K=2, TM-ARCH(0.5) and TM-ARCH(1.0) assumes 4.82% and 8.73% larger channel width than ARCH, respectively.

When K=2, TM-ARCH(0.1) and TM-ARCH(0.2) achieve almost the same delay as ARCH, although they use extra 13.45% and 7.89% channel width, respectively. This is somewhat surprising. Compared with TM-ARCH(0.5) and TM-ARCH(1.0), these two architectures contain less multiplex-able track population, hence are closer to architecture ARCH. But the critical path delay results seem to suggest the opposite: while TM-ARCH(1.0) assuming extra 8.73% channel width improves delay by 4.65%, TM-ARCH(0.1) assuming extra 13.45% channel width does not improve delay at all.

From K=4 onwards, time-multiplexed interconnects demonstrate their performance advantages over conventional interconnect. For example, TM-ARCH(0.2) improves delay by 3.03%, 6.20% and 1.76%, respectively. And TM-ARCH(0.2) achieves these improvements using 13.61%, 16.52%, and 22.86% smaller channel widths (see Table 6.5). Similarly, TM-ARCH(0.5) improves delay by 2% - 3%, assuming at least

Table 6.5: Average low-stress channel widths versus multiplex-able track population, a, and number of microcycles in a user clock cycle, K.

	K=1		K=2		K=4		K=6		K=8	
	W_{ls}	%	W_{ls}	%	W_{ls}	%	W_{ls}	%	W_{ls}	%
a=0.1	68.2	-	77.4	13.45%	65.4	-4.15%	62.0	-9.09%	56.5	-17.16%
a=0.2	68.2	-	73.6	7.89%	59.0	-13.61%	57.0	-16.52%	52.6	-22.86%
a=0.5	68.2	-	71.5	4.82%	58.1	-14.84%	52.3	-23.42%	42.8	-37.34%
a=1.0	68.2	-	74.2	8.73%	53.0	-22.34%	41.0	-39.88%	34.6	-49.27%

14% smaller channel widths. One may say that the percentages improvements are quite small. But the percentages of channel width saving are decent. At least, we can claim that TM-ARCH(a) achieves comparable performance with decent channel width savings.

In Figure 6.7, the four curves do not share a common trend along the x-axis. For example, TM-ARCH(0.1) achieves the best delay at K=6, while TM-ARCH(0.5) at K=4. This is so, because actually different channel widths are assumed at the various data points along each curve. Take the curve correspond to TM-ARCH(0.1) as an example. Average channel width of 77.4, 65.4, 62.0, and 56.5 is assumed for K=2, 4, 6, and 8, respectively. Since the assumed channel width is different, here it is difficult for us to correlate critical path delay with K. Hence the individual curve's trend is not very meaningful here in this figure. Based on this same reason, we cannot correlate critical path delay with a in this figure.

Area-Delay Product

This section reports area-delay product results collected from the low-stress routing experiments. Here area is the total FPGA area including both logic area and routing area. Figure 6.8 presents the area-delay product results. The x-axis

is number of microcycles in a user clock cycle. The y-axis is area-delay product. Each curve corresponds to a different value of parameter a, and plots average area-delay product versus K. Each data point is the geometric average over 20 MCNC benchmark circuits. Note that K=1 denotes conventional interconnect architecture ARCH.

It can be observed that, at a few combinations of K and a, time-multiplexed interconnects can achieve smaller area-delay products than conventional interconnect architecture ARCH. These include (1) TM-ARCH(0.1) with K= 4; (2) TM-ARCH(0.1) with K=6; (3) TM-ARCH(0.2) with K=4; and (4) TM-ARCH(0.5) with K=4. Of these four, TM-ARCH(0.1) with K=4 could be a promising candidate of time-multiplexed interconnect architecture. It achieves 9.56% smaller area-delay product than ARCH. A possible explanation for its superior area-delay product performance could be that it has the advantage of achieving better delay (5.34% smaller) with less channel width (4.15% less) by exploiting time-multiplexing, and at the same time exhibits minimal area overhead of time-multiplexing (0.1 multiplex-able track population). This same can be said on TM-ARCH(0.2) with K=4 and TM-ARCH(0.1) with K=6.

When we examine the trend of each individual curve, all but TM-ARCH(1.0) achieve their optimal area-delay product at K=4. This may suggest that having 4 microcycles in a user clock cycle is optimal for time-multiplexed interconnects. The curve of TM-ARCH(1.0) achieves its optimum at K=1, at which point the architectural is actually ARCH. This means that fully populated time-multiplexed interconnect architectures yield no gain in terms of area-delay product.

When we compare the four curves, we can observe that, from K=4 onwards, area-delay product generally gets worse as a increases. This is reasonable: as a

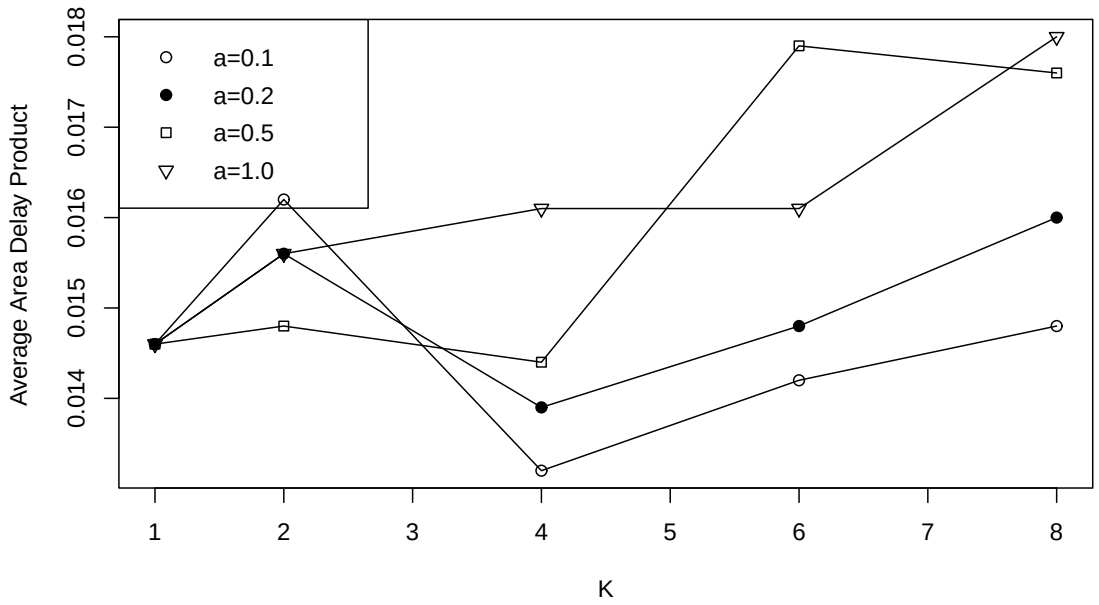


Fig. 6.8: Area delay product versus multiplex-able track population, a, and number of microcycles in a user clock cycle, K.

increases, the increasing area overhead of TM switches dominates both channel width reduction and delay improvement.

Chapter 7

Conclusion

This chapter concludes this thesis by summarizing our work and giving recommendations for future work.

7.1 Summary

The subject of this thesis is time-multiplexed interconnects for FPGAs. Around this subject, this thesis has contributed to two related research areas: FPGA architecture and FPGA routing algorithm.

Chapter 3 presents TM-ARCH, our proposed FPGA architecture with fully populated time-multiplexed interconnects. This architecture is based on the classical island-style architecture. All wires in routing channels can be time-multiplexed with the aid of specially designed TM switches. An architecture parameter, K, is defined to denote the number of microcycles in a user clock cycle. That is, the number of time slots for time-multiplexing.

Chapter 4 presents our time-multiplexing -aware timing-driven routing algo-

rithm. This routing algorithm is based on VPR 5 timing-driven routing algorithm. It employs a multiplexing-aware congestion cost function so as to identify nets for time-multiplexing. We implement this algorithm as our routing tool. By assuming standard FPGA CAD flow and replacing the conventional router with our router, we are able to implement circuits onto our proposed TM-ARCH FPGAs.

Chapter 5 presents our evaluation of TM-ARCH. Following the common practices in the research community, we use MCNC benchmark circuits and VPR flow. We compare TM-ARCH with the conventional island-style architecture based on four metrics: minimum channel width, routing area, circuit critical path delay, and area-delay product. Our evaluation shows that TM-ARCH generally can achieve smaller minimum channel widths. For example, with K=4, TM-ARCH achieves average 20% reduction in minimum channel widths over 20 MCNC circuits. But TM-ARCH is also shown to exhibit significant routing area overhead although it reduces channel widths. In the case of K=4, TM-ARCH requires 46% larger routing area. This significant area overhead is mainly due to TM switches, which consume much more area than their conventional counterpart. Our evaluation shows that TM-ARCH can achieve similar or slightly better critical path delays with smaller channel widths than the conventional island-style architecture. In the case of K=4, TM-ARCH improves critical path delay by 1.7% while using 20% smaller channel widths. Finally, our evaluation shows that TM-ARCH generally has larger area-delay product. With K=4, TM-ARCH exhibits 10% larger area-delay product.

Chapter 6 presents TM-ARCH(a), a family of FPGA architectures with partially populated time-multiplexed interconnects. TM-ARCH(a) is mainly motivated by our evaluation of TM-ARCH architecture, which reveals that area overhead of TM switches can be significant. TM-ARCH(a) family is extended from TM-ARCH

architecture. In TM-ARCH(a) family FPGAs, only a portion of tracks in routing channels can be time-multiplexed with the aid of TM switches. The architecture parameter, a, parameterizes this portion. With a=1.0, TM-ARCH(a) is equivalent to TM-ARCH. We evaluate TM-ARCH(a) family by comparing it with the conventional island-style architecture. Our evaluation shows that TM-ARCH(a) architecture with small a values can achieve smaller area-delay product than the conventional island-style architecture. For example, with K=4 and a=0.1, TM-ARCH(a) achieves 10% smaller area-delay product.

Time-multiplexing in FPGAs has been a recurring topic for more than a decade. This thesis represents a new effort on proposing the technique of time-multiplex for FPGAs. It is hoped that this thesis could re-raise some interests in time-multiplex among the FPGA research community.

7.2 Future Work

Limitations of this study and our recommendations for future research include:

1. This thesis does not evaluate the impact of time-multiplexed interconnects on power consumption. Being able to justify time-multiplexed interconnects from the power consumption perspective is important. Work on the power issue is already underway at our group.

2. In this thesis, we have presented a time-multiplexing -aware routing algorithm, and implemented this algorithm as our routing tool. In the CAD flow of FPGAs with time-multiplexed interconnects, we replace the conventional router with our router. However, the tools used for technology mapping, logic block packing, and placement are not time-multiplexing -aware. Given

that these various stages in the CAD flow (especially placement and routing) are correlated, it is necessary to have a full time-multiplexing -aware flow for FPGAs with time-multiplexed interconnects. Hence, another future work might be to develop time-multiplexing -aware algorithms for placement.

3. This thesis focuses on island-style FPGAs exclusively, and does not consider the column-based and the hierarchical architectures. This is mainly due to two reasons: (1) the island-style architecture is the most studied and the most widely used in academia and industry; (2) the underlying framework of our studies is VPR 5 tool, which targets the island-style architecture. An interesting future work might be to examine appropriate architectures and CAD algorithms to apply time-multiplexed interconnects to column-based and/ or hierarchical FPGAs.

4. Homogeneous FPGA architecture is assumed throughout this thesis. This simplification largely ignores the fact that almost all modern FPGAs contain some hard blocks, such as digital signal processing (DSP) blocks, block memories, and processors. It is important to evaluate time-multiplexed interconnects in the context of heterogeneous FPGAs. An interesting future work might be extending the architectures and algorithms presented in this thesis to heterogeneous FPGAs.

Appendix A

List of Publications

1. X. Chen and Y. Ha, "The optimization of interconnection networks in FPGAs," in *Proc. Dagstuhl Seminar 10281*, Saarland, Germany, 2010.

2. R. Syed, X. Chen, Y. Ha and B. Veeravalli, "sFPGA2 - A scalable GALS FPGA architecture and design methodology," in *Proc. Intl. Conf. on Field Programmable Logic and Applications*, Prague, Czech, 2009.

3. S. Fernando, X. Chen and Y. Ha, "sFPGA - A scalable switch based FPGA architecture and design methodology," in *Proc. Intl. Conf. on Field Programmable Logic and Applications*, Heidelberg, Germany, 2008.

4. H. Liu, X. Chen and Y. Ha, "An architecture and timing-driven routing algorithm for area-efficient FPGAs with time-multiplexed interconnects," in *Proc. Intl. Conf. on Field Programmable Logic and Applications*, Heidelberg, Germany, 2008. (Poster)

5. H. Liu, X. Chen and Y. Ha, "An area-efficient timing-driven routing algorithm for scalable FPGAs with time-multiplexed interconnects," in *Proc. Intl.*

IEEE Symposium on Field-Programmable Custom Computing Machines, Stanford, USA, 2008. (Poster)

6. Y. Li, S. Fernando, H. Yu, X. Chen, Y. Ha and T. T. Tay, "Tighter WCET analysis of input dependent programs with classified-cache memory architecture," in *Proc. IEEE Intl. Conf. on Electronics, Circuits and Systems*, Malta, 2008.

7. X. Chen and Y. Ha, "Tutorial: EDA tools for leakage analysis and optimization," in *Proc. Intl. PhD Student Workshop on SoC*, Taipei, Taiwan, 2007.

Appendix B

Curriculum Vitae

Xiaolei Chen was born in Ezhou, Hubei Province, China on December 26, 1984. He majored in Electronic Information Science and Technology and received his Bachelor of Science degree from University of Science and Technology of China (USTC) in July 2006. From August 2006 until September 2011, he was pursuing his PhD studies with Department of Electrical and Computer Engineering (ECE) at National University of Singapore (NUS). He has been employed with ST-Ericsson Asia Pacific (Singapore) as IC Design Engineer since September 2011.

From 2008 until 2010, he served as MEng/ PhD Student Representative at Department ECE, NUS. From 2009 to 2011, he served in ECE Alumni Relations Committee at Department ECE, NUS. In 2011, he served as the founding president of ECE Graduate Student Council at Department ECE, NUS. In May 2011, he served as the Program Vice Chair of 1st NUS ECE Graduate Student Symposium.

He has been a Student Member of IEEE since 2007. In 2011, he served as treasurer of IEEE GOLD (Graduate Of the Last Decade) Singapore affinity group.

His main research interests include FPGA architecture and computer-aided

design (CAD) for integrated circuits.

Bibliography

[1] Michael J. Alexander, James P. Cohoon, Jared L. Colflesh, John Karro, Edward L. Peters, and Gabriel Robins. Placement and routing for three-dimensional fpgas. pages 11 –18, 1996.

[2] M.J. Alexander, J.P. Cohoon, J.L. Colflesh, J. Karro, and G. Robins. Three-dimensional field-programmable gate arrays. In *ASIC Conference and Exhibit, 1995., Proceedings of the Eighth Annual IEEE International*, pages 253 –256, sep 1995.

[3] M.J. Alexander, J.P. Cohoon, J.L. Ganley, and G. Robins. Performance-oriented placement and routing for field-programmable gate arrays. In *Design Automation Conference, 1995, with EURO-VHDL, Proceedings EURO-DAC '95., European*, pages 80 –85, sep 1995.

[4] Jason H. Anderson and Farid N. Najm. Low-power programmable fpga routing circuitry. *IEEE Trans. Very Large Scale Integr. Syst.*, 17:1048–1060, August 2009.

[5] Vaughn Betz and Jonathan Rose. Vpr: A new packing, placement and routing tool for fpga research. In *Proceedings of the 7th International Workshop on*

Field-Programmable Logic and Applications, FPL '97, pages 213–222, London, UK, UK, 1997. Springer-Verlag.

[6] Vaughn Betz, Jonathan Rose, and Alexander Marquardt, editors. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, Norwell, MA, USA, 1999.

[7] G. Borriello, C. Ebeling, S.A. Hauck, and S. Burns. The triptych fpga architecture. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 3(4):491 –501, dec. 1995.

[8] Vikram Chandrasekhar. Cad for a 3-dimensional fpga. S.M. Thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 2007. Available online.

[9] Ken Eguro and Scott Hauck. Armada: timing-driven pipeline-aware routing for fpgas. In *Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, FPGA '06, pages 169–178, New York, NY, USA, 2006. ACM.

[10] Rosemary Francis, Simon Moore, and Robert Mullins. A network of time-division multiplexed wiring for fpgas. In *Proceedings of the Second ACM/IEEE International Symposium on Networks-on-Chip*, NOCS '08, pages 35–44, Washington, DC, USA, 2008. IEEE Computer Society.

[11] I. Kuon and J. Rose. Measuring the gap between fpgas and asics. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(2):203 –215, feb. 2007.

[12] Edmund Lee, Guy Lemieux, and Shahriar Mirabbasi. Interconnect driver design for long wires in field-programmable gate arrays. *J. Signal Process. Syst.*, 51:57–76, April 2008.

[13] M. Leeser, W.M. Meleis, M.M. Vai, S. Chiricescu, Weidong Xu, and P.M. Zavracky. Rothko: a three-dimensional fpga. *Design Test of Computers, IEEE*, 15(1):16 –23, jan-mar 1998.

[14] Guy Lemieux, Edmund Lee, Marvin Tom, and Anthony Yu. Directional and single-driver wires in fpga interconnect. In *IEEE International Conference on Field-Programmable Technology*, 2004.

[15] Guy Lemieux and David Lewis. *Design of Interconnection Networks for Programmable Logic*. Kluwer Academic Publishers, Norwell, MA, USA, 2004.

[16] David Lewis, Elias Ahmed, Gregg Baeckler, Vaughn Betz, Mark Bourgeault, David Cashman, David Galloway, Mike Hutton, Chris Lane, Andy Lee, Paul Leventis, Sandy Marquardt, Cameron McClintock, Ketan Padalia, Bruce Pedersen, Giles Powell, Boris Ratchev, Srinivas Reddy, Jay Schleicher, Kevin Stevens, Richard Yuan, Richard Cliff, and Jonathan Rose. The stratix ii logic and routing architecture. In *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, FPGA '05, pages 14–20, New York, NY, USA, 2005. ACM.

[17] David Lewis, Vaughn Betz, David Jefferson, Andy Lee, Chris Lane, Paul Leventis, Sandy Marquardt, Cameron McClintock, Bruce Pedersen, Giles Powell, Srinivas Reddy, Chris Wysocki, Richard Cliff, and Jonathan Rose. The stratix routing and logic architecture. In *Proceedings of the 2003 ACM/SIGDA*

eleventh international symposium on Field programmable gate arrays, FPGA '03, pages 12–20, New York, NY, USA, 2003. ACM.

[18] Chih-Chang Lin, D. Chang, Yu-Liang Wu, and M. Marek-Sadowska. Time-multiplexed routing resources for fpga design. In *Custom Integrated Circuits Conference, 1996., Proceedings of the IEEE 1996*, pages 152 –155, May 1996.

[19] Jason Luu, Peter Jamieson, Ian Kuon, and Jonathan Rose. Vpr version 5.0. Talk slides available online (45 pages), University of Toronto.

[20] Jason Luu, Ian Kuon, Peter Jamieson, Ted Campbell, Andy Ye, Wei Mark Fang, and Jonathan Rose. Vpr 5.0: Fpga cad and architecture exploration tools with single-driver routing, heterogeneity and process scaling. In *Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '09, pages 133–142, New York, NY, USA, 2009. ACM.

[21] T. Mak, P. Sedcole, P.Y.K. Cheung, and W. Luk. Wave-pipelined signaling for on-fpga communication. In *ICECE Technology, 2008. FPT 2008. International Conference on*, pages 9 –16, 2008.

[22] Larry McMurchie and Carl Ebeling. Pathfinder: a negotiation-based performance-driven router for fpgas. In *Proceedings of the 1995 ACM third international symposium on Field-programmable gate arrays*, FPGA '95, pages 111–117, New York, NY, USA, 1995. ACM.

[23] University of Toronto. ifar c intelligent fpga architecture repository, June 2011.

[24] A. Rahman, S. Das, A.P. Chandrakasan, and R. Reif. Wiring requirement and three-dimensional integration technology for field programmable gate arrays.

Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, 11(1):44 –54, feb. 2003.

[25] Arifur Rahman and Vijay Polavarapuv. Evaluation of low-leakage design techniques for field programmable gate arrays. In *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, FPGA '04, pages 23–30, New York, NY, USA, 2004. ACM.

[26] Jonathan Rose. Personal communication, 2010.

[27] Sachin Sapatnekar. *Timing*. Kluwer Academic Publishers, 2004.

[28] Herman Schmit. Extra-dimensional island-style fpgas. In Peter Y. K. Cheung and George Constantinides, editors, *Field Programmable Logic and Application*, volume 2778 of *Lecture Notes in Computer Science*, pages 406–415. Springer Berlin / Heidelberg, 2003.

[29] Akshay Sharma, Katherine Compton, Carl Ebeling, and Scott Hauck. Exploration of pipelined fpga interconnect structures. In *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, FPGA '04, pages 13–22, New York, NY, USA, 2004. ACM.

[30] Akshay Sharma, Carl Ebeling, and Scott Hauck. Piperoute: a pipelining-aware router for fpgas. In *Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, FPGA '03, pages 68–77, New York, NY, USA, 2003. ACM.

[31] Amit Singh, Arindam Mukherjee, and Malgorzata Marek-Sadowska. Interconnect pipelining in a throughput-intensive fpga architecture. In *Proceedings of*

the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays, FPGA '01, pages 153–160, New York, NY, USA, 2001. ACM.

[32] Deshanand P. Singh and Stephen D. Brown. The case for registered routing switches in field programmable gate arrays. In *Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*, FPGA '01, pages 161–169, New York, NY, USA, 2001. ACM.

[33] Inc. Tabula. Spacetime architecture white paper. White paper, Tabula, Inc., August 2011. Available online (14 pages).

[34] Paul Teehan, Guy G.F. Lemieux, and Mark R. Greenstreet. Towards reliable 5gbps wave-pipelined and 3gbps surfing interconnect in 65nm fpgas. In *Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '09, pages 43–52, New York, NY, USA, 2009. ACM.

[35] S. Trimberger, D. Carberry, A. Johnson, and J. Wong. A time-multiplexed fpga. In *FPGAs for Custom Computing Machines, 1997. Proceedings., The 5th Annual IEEE Symposium on*, pages 22 –28, April 1997.

[36] William Tsu, Kip Macy, Atul Joshi, Randy Huang, Norman Walker, Tony Tung, Omid Rowhani, Varghese George, John Wawrzynek, and André DeHon. Hsra: high-speed, hierarchical synchronous reconfigurable array. In *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, FPGA '99, pages 125–134, New York, NY, USA, 1999. ACM.

[37] Xilinx. Spartan-6 user guide. User Guide, Xilinx.

[38] Xilinx. Xc4000e and xc4000x series field programmable gate arrays product specification, version 1.6. Product Specification, Xilinx, May 14 1999. Available online (68 pages).

[39] Inc. Xilinx. Xilinx stacked silicon interconnect technology delivers breakthrough fpga capacity, bandwidth, and power efficiency, v1.0. White paper WP380, Xilinx, Inc., October 2010. Available online (10 pages).

[40] S. Yang. Logic synthesis and optimization benchmarks. 1991.