2008

Beyond the arithmetic constraint: depth-optimal mapping of logic chains in reconfigurable fabrics

Michael Todd Frederick
Iowa State University

Beyond the arithmetic constraint: depth-optimal mapping of logic chains in reconfigurable fabrics

by

Michael Todd Frederick

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Engineering

Iowa State University

Ames, Iowa

2008

To my parents for teaching me how to be a better man,

and to my brother for being a better man,

and to my Melissa for making me want to be a better man...

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1. INTRODUCTION

As computing systems have evolved, a myriad of applications and their specific requirements have driven the specialization of architectures. Processors dominate the general purpose computing realm because of their ability to flexibly implement a wide range of applications with different execution styles, but to do so they must sacrifice performance and efficiency. On the other end of the spectrum, application specific integrated circuits (ASICs) perform highly-repetitive tasks with extreme efficiency and performance, but are so specialized that they can rarely be called upon to do anything else. As the societal pervasiveness of computing has increased, the need for architectures that can blend the advantages of general purpose processors (GPPs) with ASICs has grown. Field Programmable Gate Arrays (FPGAs) have emerged as a technology capable of bridging the gap between efficiency and flexibility.

SRAM look-up table (LUT) based FPGAs are designed to flexibly implement a wide range of applications in programmable hardware. They have traditionally been relegated to the realm of prototyping because they lacked the performance necessary to be critical pieces of a production design. ASICs yield designs which are faster, occupy less area, and consume less power than their reconfigurable counterparts, in most cases by many factors of a given performance metric [77][48]. However, advances in codesign, process technology, and innovative architectures have narrowed the performance gap between FPGAs and ASICs to the point where their flexibility and relatively low cost have made them justifiable design choices. Modern FPGAs have embedded dedicated components such as multipliers, RAM modules, and microcontrollers alongside reconfigurable logic in an effort to provide the specialized resources to achieve the necessary performance. One important dedicated structure present in nearly all commercially available architectures is the arithmetic carry chain.

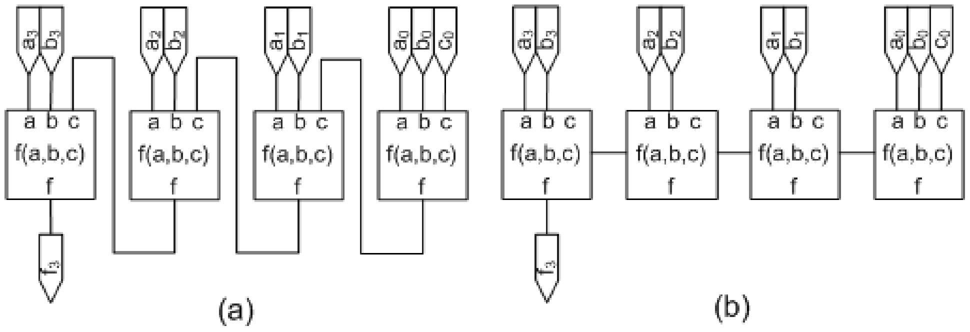


Figure 1.1 Reuse concept, 4 cascaded LEs connected trough (a) general routing nets, and (b) chain nets.

The programmable interconnection array is responsible for connecting logic elements (LEs) within a reconfigurable fabric, tying the output of one LE to the input of another. LEs serve as the basic unit of computation, typically containing a memory structure (flip-flop), look-up table (LUT), and carry chain logic. Simple interconnection arrays are pretty straightforward, connecting any source to any sink using programmable general routing wires. To take advantage of locality, hierarchal routing structures give clusters of LEs the ability to first connect amongst themselves through local routing, and if necessary, connect to LEs outside the cluster through general routing. Regardless of the LE architecture or number of dedicated components of a reconfigurable fabric, the interconnection array is the fabric's greatest strength, but also the limiting factor on performance.

As the need for FPGAs to perform faster and more efficiently has grown, carry chains have been added to accelerate arithmetic computation. Arithmetic operations are characterized by the direct dependence of the computation of cell i on the output of cell $i-1$. Carry chains are an indispensable part of reconfigurable fabrics, as they enable arithmetic functions to be implemented in an efficient manner by avoiding the use of the slower, more flexible general routing array. However, limiting the use of the carry chain to arithmetic operations underutilizes a potentially powerful resource.

Figure 1.1(a) shows how a combinational path spanning through three levels of performance-costly general routing can be converted to use the highly-optimized carry chain instead, Figure 1.1(b). The Xilinx ISE tool set estimates, for its entire Virtex Family of FPGAs, the routing delay encountered when using general routing to typically be about 300-1500 ps and a carry chain net to contribute 0 ps wire delay to a circuit [74]. Clearly, the carry chain is a highly efficient interconnection structure which could potentially benefit a variety of applications, if it is allowed to by the architecture and the computer aided design (CAD) tool flow.

1.1 Motivation

The goal of reusing the carry chain for non-arithmetic operations is to map single fanout nets to a logic chain instead of programmable routing, but only when it benefits the circuit. Such a strategy must be applicable to a large enough number of nets in an optimized netlist to make a significant impact, as per Amdahl's law. To justify potential resource expenditures, the first step is to see how many such nets exist designs. Figure 1.2 depicts a histogram of the fanout degree of nets for the aggregated MCNC [76] benchmark suite. It shows that 80% of nets fanout to only one or two other LUTs. This means that the vast majority of nets connect one LE source to one LE sink in an exclusive relationship. Clearly, this indicates that there is a broad potential impact for carry chain reuse.

There are opportunities to use carry chains in all designs, especially for those that have few carry chains to begin with. Many designs are typically sparsely arithmetic in nature. Table 1.1 shows the design characteristics of a sampling of designs available at OpenCores[62]. As evidenced by the low percentage of arithmetic cells in most designs, the maximum being $cfft$ at slightly over 50% penetrance, there are many designs that drastically underutilize the carry chain and could benefit from reuse. Many designs simply lack hardware description language (HDL) designated arithmetic carry chains.

Design depth is created by the l logic and g programmable general routing connections along paths from primary inputs (PIs) to any node in a network. In Figure 1.3, the PIs have a routing depth $g = 0$, and for each LUT in the chain input logic network $g = 1$, $l = 1$. The first member of the ripple-carry chain has logic depth of $l = 2$, while the last has $l = 5$. Each LUT in the chain network possesses a routing depth of $g = 2$, because the path to any LUT in the chain traverses 2 routing connections and increases only logic depth. Each chain node is a

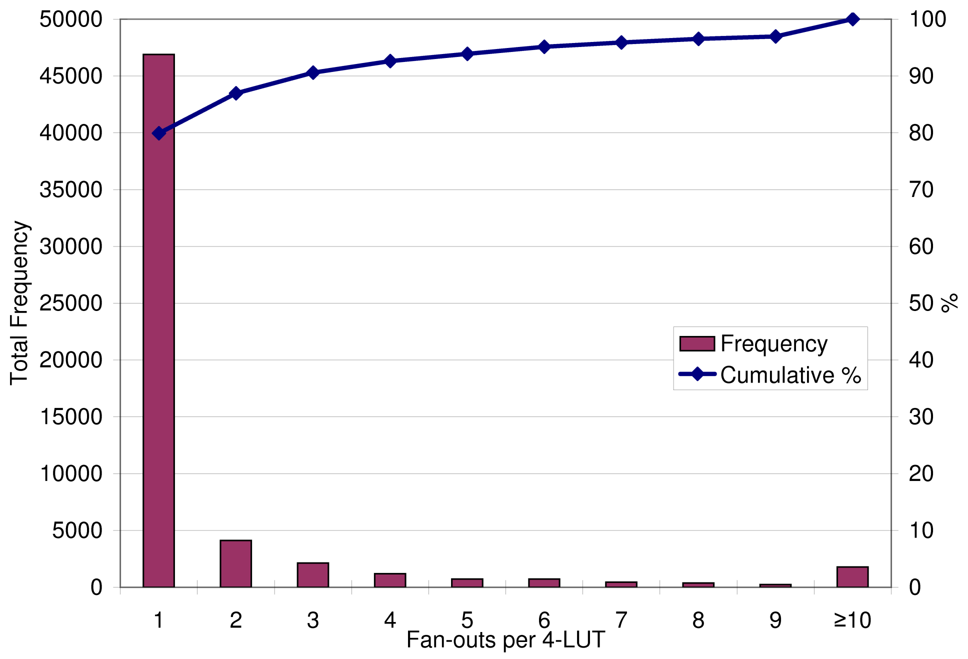


Figure 1.2 Histogram of aggregated netlist fanout for 20 MCNC benchmarks.

Table 1.1 $K = 5$ Design Characteristics

Design	LEs	Arith. LEs	Nets	Ave. LE Fanout	% Arith.
cfft	3360	1753	6758	6.23	52.17
xtea	731	299	1494	6.00	40.90
rsa	945	268	1556	6.17	28.36
sha512	4394	1164	6765	7.23	26.49
jpeg	7015	1715	13976	6.28	24.45
dct	6140	1414	12565	5.82	23.03
ethernet	287	34	363	5.62	11.85
md5	2838	262	3467	5.65	9.23
usb	3141	283	3703	5.91	9.01
ava	12743	1094	14410	5.88	8.59
des3_area	908	10	1268	5.42	1.10
reed_sol	1228	10	1251	6.53	0.81
twofish256	2602	0	2858	5.62	0.00
aes128_fast	12122	0	12255	5.33	0.00
Total	58454	8306	82689	5.92	14.21

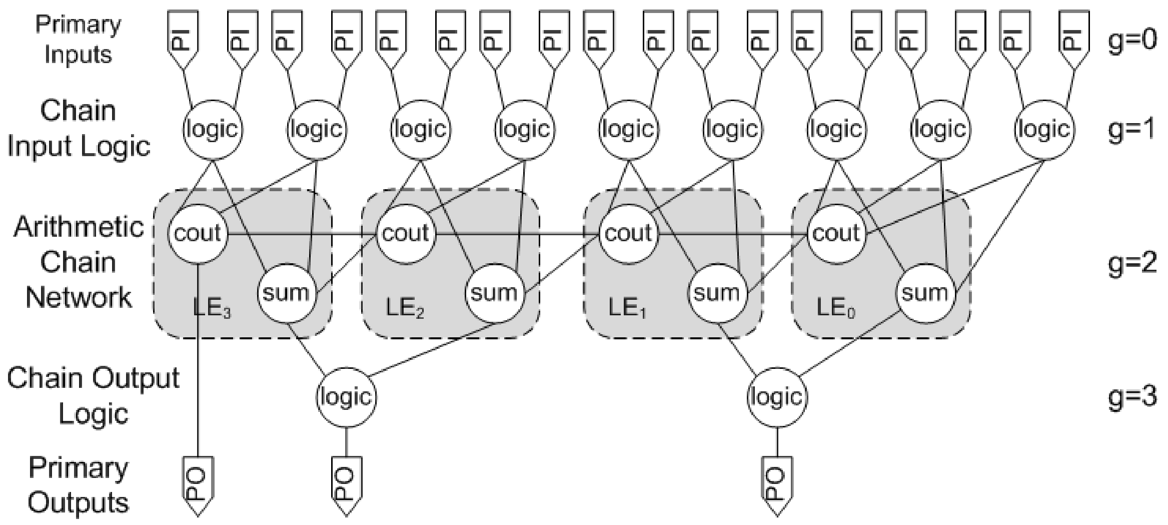


Figure 1.3 Carry chain network with 4 LEs, 8 LUTs contained within a 11 LUT logic network.

depth increasing node, one that increases logic depth without increasing routing depth. Carry chains provide near-zero delay transmission of a carry, but are invoked only through HDL macros. For designs that contain few arithmetic operations and are implemented in a carry-select style architecture, the carry chain is an idle resource. Generic logic chains encompass both arithmetic and non-arithmetic chains and view the carry chain as an equal opportunity resource.

Generic logic chains address two common pitfalls of HDL-based arithmetic chains. First, arithmetic bias can cause some nets to be incorrectly assigned, actually increasing the critical path of the design. Figure 1.4(a) shows an example of an incorrectly assigned carry chain. For this example, assume that an LUT has delay of $250ps$ and that a general routing connection, conservatively, costs twice that at $500ps$. Additionally, all PIs and POs are connected to the network through general routing connections. The critical path in this example is the path from function $f(a_0, b_0, c_0)$ through five general routing connections and four LUTs, $delay = 5 \cdot g + 4 \cdot l \approx 3.5ns$. However, if the carry chain is remapped to the actual critical path in Figure 1.4(b), i.e. that which traverses the most general routing connections, $delay = 4 \cdot g + 3 \cdot l \approx 2.75ns$. Note that Figure 1.4(b) reports the logic depth of the critical path, not

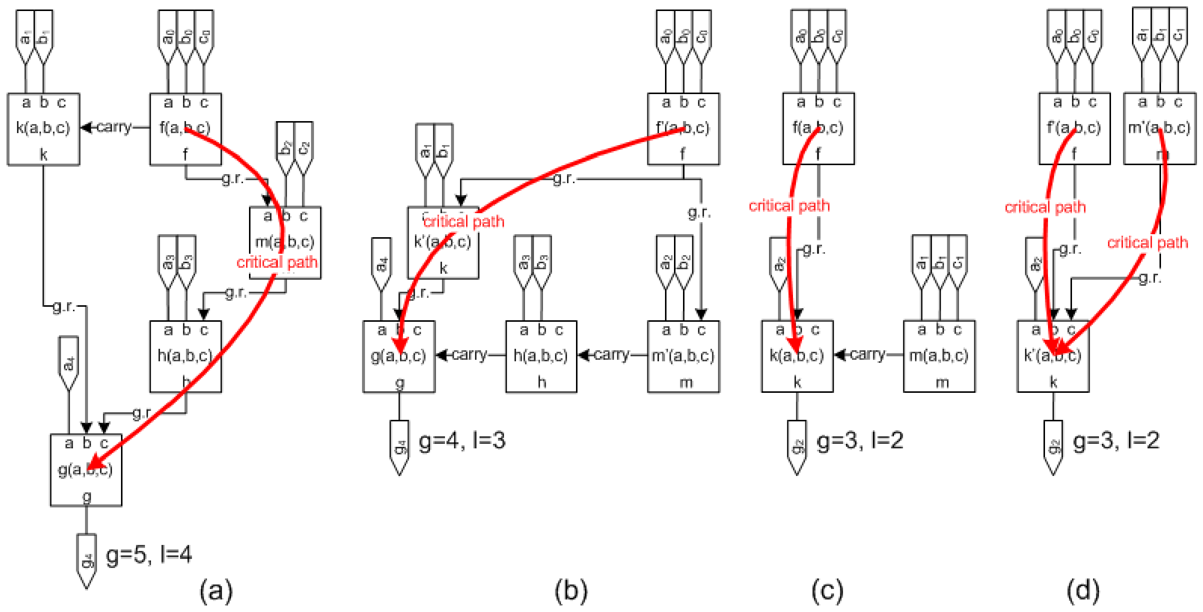


Figure 1.4 Arithmetic bias in chain allocation.

the network logic depth of $l = 4$. Assuming carry chains have $c = 0ps$ delay, the chain path has $delay = 3 \cdot g + 4 \cdot l + 2 \cdot c \approx 2.5ns$.

Likewise, arithmetic bias can lead to chains that are too aggressive. Figure 1.4(c) depicts an LE with routing depth of $g = 3$, despite making use of its chain input. In this case, it is of no advantage to restrict the output of $m(a_1, b_1, c_1)$ to using the chain net because the critical path is unaffected. Instead, the network can be implemented as in Figure 1.4(d), where both $m'(a_1, b_1, c_1)$ and $f'(a_0, b_0, c_0)$ drive general routing nets which are sunk at $k'(a_2, f', m')$. This change results in equal routing depth critical paths that afford the place and route (PNR) and clustering tools more flexibility.

The critical path can be affected negatively by bias toward arithmetic operations. This is exacerbated by the preservation of arithmetic chains through synthesis and technology mapping. In current design tools, arithmetic functions are detected at the HDL level and granted immunity throughout the entire design flow. This partitions the design into arithmetic chains and regions of support logic, as pictured in Figure 1.3. These partitions are protected from synthesis modification, inclusion in LUTs with other nodes, and subject to the requirement that the chain be clustered, placed, and routed in one contiguous chain, regardless of its actual affect on design routing depth. Allowing the design flow to choose which chains are necessary can yield better circuits.

Arithmetic and non-arithmetic chains both impact the performance of a design. Therefore, it is only fair that all arithmetic and general routing nets be given fair access to the high performance carry chain. Generic logic chains encompass both computation styles, and their formation at the technology mapping stage can alleviate the bias HDL macros create in a design. Through novel architectures, relaxation techniques, and innovative clustering solutions generic logic chains can be successfully created and utilized without HDL arithmetic macros.

1.2 Hypothesis

Given an arbitrary Boolean network and a capable architecture, generic logic chains can be created during technology mapping in polynomial time, without the preservation of high-level hardware description language macros. Each Boolean node in the mapped solution possesses optimal routing depth and, within its confines, optimal logic depth.

1.3 Approach and Contributions

A review of reconfigurable architectures and design flow techniques presented in Chapter 2 will show that, beyond using HDL macros and device primitives, there seems to be no published solution to the problem of mapping chains in reconfigurable architectures. Furthermore, architectures that support generic logic chains either use the existing carry chain at a sub-width capacity or require extra or specialized interconnection to achieve full-width chains.

Viewing the carry chain as an exploitable resource and successfully exploiting it are vastly different propositions. There are two main obstacles to successfully reusing the carry chain in reconfigurable fabrics: 1) the architecture should allow the entire K-LUT (full-width) output to traverse the carry chain and 2) the tools must be able to identify and form generic logic chains that encompass both arithmetic and non-arithmetic operations. Facilitating logic chain formation in the architecture must not degrade the performance of the arithmetic (sub-width) operations nor the performance of the traditional combinational output, and have minimal cell area impact. Likewise, identifying generic chains with CAD tools must preserve the performance of HDL solutions.

Chapter 3 presents an architecture that allows the complete K-LUT function to drive the carry chain. The major drawback of traditional carry-select architectures, which directly facilitate sub-width $(K-1)$-LUT generic logic chains, is just that–they only support sub-width chains. Alternately, the full-width K-LUT chains that are available in commercial FPGA architectures require a connection between logic elements that is separate from the existing carry chain. Here, a novel architecture is presented that provides the capacity for full-width chains trough reuse of the existing carry chain with minimal extraneous logic.

The typical FPGA design flow consists of synthesis, technology mapping, clustering, placement, and routing. Tool support for logic chains is addressed through the augmenting the technology mapping stage. The first step in developing a chain-capable tool flow is the creation of a suite of heuristic algorithms operating on a post-technology mapped design, created as a simple addition to a commercial design flow and presented in Chapter 4. These approaches are used as a case study in chain reuse, in an effort to establish its viability and guide the development of an optimal technology mapping solution for chains.

The heuristics of Chapter 4 will show that technology mapping is the ideal step at which to create logic chains. Technology mapping serves as the interface between an architecture non-specific Boolean network and a solution which is utilizes specific computing and routing resources. Chapter 5 presents a novel technology mapping algorithm, called ChainMap, that creates an optimal routing depth solution to an arbitrary Boolean network. ChainMap is inspired by the optimal logic depth FlowMap algorithm [20], but instead of defining depth as the number of logic levels of a network, depth is defined as the number of traversals of the general routing array. *Logic chains* are generalized to include both arithmetic and non-arithmetic operations. A logic chain is defined as a series of *depth increasing nodes*, i.e. those nodes which increase the logic depth of the design, while maintaining its routing depth. ChainMap identifies the optimal routing depth solution of a network in polynomial time, and in doing so, establishes optimal logic chains. Chapter 5 also presents heuristic strategies for relaxing chains that are not part of the critical path established by the optimal solution.

Chapter 6 presents methods for dealing with chains post-technology mapping. This includes novel technology map reduction and clustering techniques. While ChainMap harnesses chains for purposes of improving execution speed along the critical path, chains can also be used to reduce design area. While a chain is a single fanout connection between LEs, it is also potentially a dual fanout connection between a source LUT and two sink $(K-1)$-LUTs. ChainPack is presented as method for creating chains when they result in a reduction in the number of LEs in a design. Another contribution of Chapter 6 is HierARC, a new hierarchical clustering tool for FPGAs, adapted from bioinformatic microarray analysis.

The post place and route performance of all architectural and design flow contributions are assessed in Chapter 7 relative to traditional metrics such as circuit delay, area, and routing utilization. Designs available from OpenCores [62] are used because they contain the HDL macro chains necessary to measure the full impact of ChainMap. Chapter 8 concludes with a discussion of results, contributions, and directions for further study.

CHAPTER 2. REVIEW OF LITERATURE

2.1 FPGA Architecture

In beginning a discussion of reconfigurable fabrics and SRAM-based FPGAs, it is necessary to describe their architecture. Reconfigurable fabrics, though typified by the FPGA, come in a variety of shapes, styles, and granularities. A survey of academically proposed architectures is presented in [7], as well as a nomenclature for describing the spectrum of available technologies. The fabric can be characterized by three layers: interconnection, configuration, and processing. The interconnection layer refers to the general routing array that connects logic elements to one another. A typical FPGA routing architecture uses about 70-90% of the total transistors on the die [27]. The configuration layer refers to the memory structures that dictate the functionality of the programmable routing, look up tables, and any other component in the array that can operate in different modes. The processing layer consists of the actual structures that compute a value in the array. Most fabrics subscribe to the 90/9/1 model, meaning that 90% of the fabric's area is devoted to interconnection, 9% to configuration bits, and 1% to actual processing elements. The performance of modern reconfigurable fabrics, in terms of execution speed, area, utilization, and power consumption, is dominated by the interconnection.

Logic elements (LEs) are typically designed to implement any K-input, one output Boolean operation or $(K-1)$-input, two output arithmetic operation. The standard computation element is the SRAM look-up table, Figure 2.1(1) is, for all intents and purposes, a K-input multiplexer whose data inputs are populated by static random access memory (SRAM) cells, and whose selection inputs are the operation inputs. In this way, for $K = 3$ all $2^{2^3} = 2^8$ 3-input, 1-output functions can be implemented. Each LE typically contains carry logic to support arithmetic chains, marked (2), and a D-Flip Flop (DFF), denoted by (3), for sequential

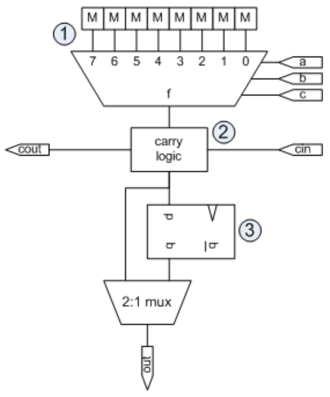


Figure 2.1 A conceptual logic element (LE).

logic. SRAM cells are the most common configuration tool, though flash and anti-fuse are also among the commercially available technologies [1][2].

The most effective LUT width for reconfigurable fabrics has traditionally been viewed as the 4-LUT. There are a few studies assessing the area/speed trade-off for LUT size in an FPGA, such as [68] which finds that 5 or 6 input LUTs are better from a execution speed perspective, and [63] which finds that 3 or 4 input LUTs are better for area efficiency. More recently, the Stratix II/III adaptive logic module (ALM) [43] has contended that a fracturable LUT offers the best potential performance. The fracturable LUT is radically different from standard FPGA cell design, and supports multiple LUT widths including dual 6-LUTs with 4 shared inputs, dual 5-LUTs with 2 shared inputs, dual independent 4-LUTs, and a single 7-LUT with a subset of all operations. The Xilinx 5 FPGAs [74] also feature 6-LUTs, indicating that, in general, FPGAs are moving toward incorporating more logic in every computing resource.

The move to wider LUTs is partially attributable to the desire to increase performance by decreasing the number of traversals through general routing for all designs. Another reason for wider LUT widths is that shrinking technology sizes have caused the size of an LUT to shrink, so that more complex LUT structures can fit in the same space that a legacy 4-LUT previously occupied. However, this size changing does not extend to the general routing array because smaller wires lead to increased resistance and delay. Simply, interconnection does not scale as well as logic [64]. This result is bore out in Table 2.1, where LUT delay shrinks with process

Table 2.1 Commercial FPGA estimated component delays in picoseconds.

FPGA	K	Routing (ps)		Logic (ps)		Process
		General	Chain	Chain	K-LUT	
Xilinx Virtex 2 Pro	4	[300,1500]	0	39	250	1.5V, 0.13 μm
Xilinx Virtex 4	4	[300,1400]	0	34	147	1.2V, 90 nm
Xilinx Virtex 5	6	[245,1200]	0	20	80	1.0V, 65 nm
Altera Stratix	4	[300,1500]	0	58	366	1.5V, 0.13 μm
Altera Stratix II	6	[300,1300]	0	35	366	1.2V, 90 nm

technology, but typical routing delay remains relatively unchanged. If the area of the routing array remains reasonably static and the processing layer shrinks, the inclination is to give the processing layer more functionality by substituting a fracturable LUT for a two 4-LUTs in the same physical area [43]. However, the additional complexity of LEs also necessitates wider interfaces to routing, with every input to a LE requiring 30 or more support routing wires [43]. As vendors increase K, they often increase the complexity of the routing array as well.

Figure 2.2 shows how LEs (1) are arranged in a programmable interconnection array such that groups of LEs, called clusters, are formed to share resources. Altera refers to clusters as logic array blocks (LABs), while Xilinx refers to them as configurable logic blocks (CLBs). The LEs in each cluster share local interconnect (2), access to the general routing array (3), common control signals, and a carry chain. Programmable routing consists of two types of interconnection structures, local and general routing. Local routing allows an LE to connect to any other LE in the same cluster, while general routing connects clusters.

In the limited example provided by Figure 2.2, row channels intersect columns every 4 clusters, however, in practice general routing arrays are far more densely packed with wires of varying length, typically 1, 2, 4, 8, 16, and 24 clusters, with switches allocated liberally. The interconnection array is flush with resources, containing adjacent LE connection structures providing low-latency connectivity and long-distance wires allowing any LE to connect to any other LE. All this connectivity comes at the expense of increased chip area or domination of

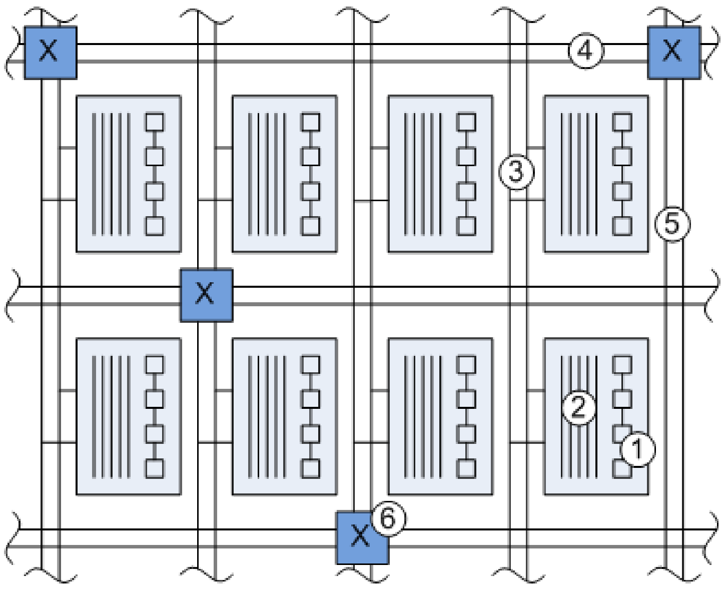


Figure 2.2 An island-style FPGA.

the area that is available. Channels consist of individual tracks, each containing the basic segment configuration of wires of varying length.

Clusters are first introduced in [50], to improve performance and density in FPGAs strangled by the amount of connectivity necessary to form large arrays of independent LEs. Each cluster is connected through their local routing to the row (4) and column (5) signal channels using connection boxes. Specific row and column channel intersection points contain configurable switches (S-boxes) (6) that enable each LE to access any other LE in the array. Row and column channels intersect at the switch boxes at regular intervals.

FPGAs are commonly classified as either island-style or hierarchical. Each possesses clusters that are surrounded on each side by general routing, allowing them access to all other clusters, as in Figure 2.2. However, a hierarchical FPGA views each cluster as a mini-FPGA containing LEs, I/O pads, and other components that connect to each other via local routing [3]. The distinction is that each cluster in a hierarchical FPGA is treated as a small, self-contained reconfigurable module, while island-style clusters consist only of LEs.

The size of the cluster has evolved over time and varies greatly depending on the size of the LE and functionality required within the cluster. When choosing a cluster size there are two considerations, the number of LEs in the cluster, N, and the number of inputs from general routing, I. Within a cluster, the LEs are typically fully connected, i.e. the output of any LE can be mapped to the input of any LE in the same cluster. However, not all inputs to each LE are are accessible from general routing. One result, found in [11], is that for $K = 4$ and clusters of $N \leq 16$ LEs, $I = 2 \cdot N + 2$ is a sufficient number of general routing inputs to maintain 98% cluster utilization. Additionally, because the functionality of cluster I/O renders each LE functionally equivalent from the point of view of the general routing array, the size of the routing array can be significantly decreased.

Clusters of size $1 \leq N \leq 8$ LEs are area efficient from the standpoint of the number of transistors needed to support LEs with local routing. This is largely dependent on the quadratic growth of the number of cluster input multiplexers necessary to provide cluster connectivity with the $I = 2 \cdot N + 2$ general routing interface for $K = 4$ [11]. More recent work in [4], accounting for a modern process technology size of $0.18\mu m$, and using a full timing model, finds $I = \frac{K}{2} \cdot (N + 1)$ for LUTs of $5 \leq K \leq 6$, while reiterating that cluster sizes $3 \leq N \leq 10$ yield 98% cluster utilization.

As FPGAs have evolved, the demands on their performance and capabilities has rapidly increased. Historically, FPGAs have been used almost exclusively for design prototyping. However, due to their flexibility and relative low cost, they have recently become a viable inclusion in production designs. Manufacturers have tried to recoup performance in FPGAs through the inclusion of common dedicated components such as carry chains, block RAM, dedicated multipliers, much more complex and specialized components like high-speed serializer/deserializer (SERDES) communication cores, and even embedded microcontrollers. Each of these specialized and dedicated components increases design performance by intertwining commonly used computational elements with the reconfigurable logic. While these components enable designs to perform better, they are mainly positioned to help DSP applications.

Throughout this work, the reconfigurable fabrics under consideration are primarily SRAM-based LUT architectures, specifically, but not limited to, FPGAs. The terms reconfigurable fabric, programmable logic, and FPGA will be used interchangeably. Netlist and Boolean network refer to a combinational circuit implemented in a reconfigurable fabric. Each reconfigurable fabric is assumed to use a island-style routing structure consisting of inter-cluster general routing and intra-cluster local routing, known collectively as programmable routing. LEs are assumed to be a variation of Figure 2.1, at minimum containing an SRAM-based LUT, carry chain logic, and DFF, but also potentially having more complex internal support logic.

2.1.1 Commercially Available Architectures

The Altera Stratix [6] is a 4-LUT architecture, whose routing structure is typical of island-style FPGAs. Each logic array block (LAB), e.g. cluster, is a set of 10 LEs featuring 30 general local routing interconnect lines which service intra-cluster routing between LEs and provide for signals to be sourced/sank to/from the general routing array. Every LE is connected to the downstream LE on the carry, register cascade, and LUT chains. The general routing array provides connectivity between clusters in column spans of 1, 4, 8, and 16 clusters, and row spans of 1, 4, 8, and 24 clusters. The Stratix carry chain is the carry select style (Figure 2.3(b)), augmented to form a 2-level chain. The carry-select scheme in Stratix directly facilitates chain reuse for $(K-1)$-input functions.

The Stratix II/III ALM [43] is considerably different from other basic LEs. As has been previously discussed, the ALM contains a fracturable LUT capable of operating with multiple LUT widths up to dual 6-LUTs with 4 shared inputs, dual 5-LUTs with 2 shared inputs, dual independent 4-LUTs, and a single 7-LUT with a subset of all operations. In addition, each half of an ALM contains a dedicated full-adder, enabling parallel ripple carry chains for 3-operand arithmetic. The Stratix II/III carry chain is incorporated directly into the full adder, and is accompanied by a shared arithmetic signal that facilitates 2-level ripple carry addition. Both the shared arithmetic signal and the carry-in are fed directly to the full adder and thus do not use carry-select arithmetic.

The Stratix II [6] interconnection array is structured similarly to the Stratix. However, due to its inclusion of a vastly different basic logic element, fewer routing options have been provided. Each LAB is a set of 8 ALMs (de facto 16 LEs) featuring 44 local routing interconnect lines which service intra-cluster routing between ALMs and provide for signals to be sourced/sank to/from the general routing array. Every ALM is connected to the downstream ALM on the carry, register cascade, and shared function chains. The general routing array provides connectivity between LABs in column spans of 4 and 16 LABs, and row spans of 1, 16, and 24 LABs. The Stratix III general routing array uses columns spanning 4 and 12 LABs, and rows spanning 1, 4, and 20 LABs.

The Xilinx Virtex II Pro and Virtex 4 [74] are the same basic architectures, maintaining the 4-LUT as the standard computation elements. Slices contain two LEs and a configurable logic block (CLB), e.g. cluster, contains 4 slices (8 LEs). The Virtex 5, while using a 6-LUT, still maintains the same basic structure of 4 slices per CLB. Published descriptions of the V4 and V5 routing structures are vague, but appear to use a diagonal routing scheme. In all Virtex series architectures, the carry chain used is carry-propagate (Figure 2.3(a)). The carry-propagate scheme used by Xilinx is not directly compliant with chain reuse.

2.1.2 Carry Chains and Dedicated Routing between LEs

One extremely common dedicated structure found in nearly all modern programmable logic devices is the arithmetic carry chain. Each adjacent LE in a cluster is connected to its predecessor and its descendant through an exclusive connection. The carry chain is a very specific, highly optimized routing structure in that it employs specialized logic and is designed to provide near $0ps$ latency interconnect between a carry source and its adjacent sink.

There are two primary methods for implementing carry logic in reconfigurable fabrics: propagate/generate and carry-select. Figure 2.3(a) shows the propagate/generate method where the propagate function serves as the selection input to a multiplexer, choosing between the carry-in and the generate. Xilinx Virtex family FPGAs implement this style of carry computation which fits the equation $c_i = p \cdot c_{i-1} + \overline{p} \cdot g$, where the propagate condition p is the result of the LUT computation (an XOR gate), c_{i-1} is the carry computed by cell $i-1$ or the chain initialization, and the generate condition g tracks input b.

The Altera Stratix uses the carry-select method, as shown in Figure 2.3(b). This method

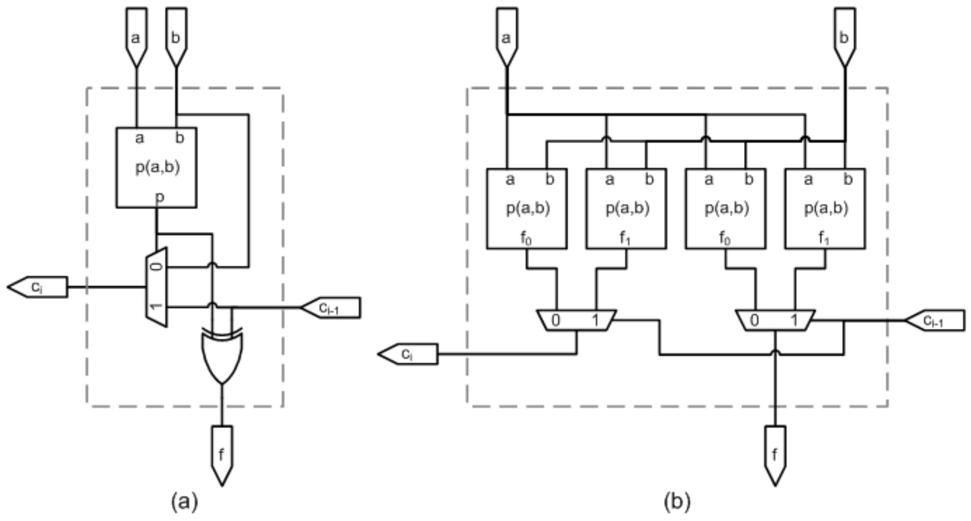


Figure 2.3 Carry computations (a) carry-propagate, (b) carry-select.

uses an LUT with one input (c_{i-1}) serving as the carry into the current LE and selects between the result of the arithmetic function computed when the carry is 0, (f_0), and when the carry is 1, (f_1), thus fitting the equation $c_i = c_{i-1} \cdot f_1 + \overline{c_{i-1}} \cdot f_0$. Likewise, the Stratix II/III ALM uses a dedicated full adder circuit in lieu of a configurable carry chain.

Commercial architectures most commonly employ a ripple carry chain because of its linear delay/area model. Its uniform architecture is a natural fit for FPGAs as each cell can be located at anywhere in the chain. More complex, higher-performance strategies such as the Brent-Kung carry-lookahead, block carry [41], and carry-skip [37] have been proposed as alternate solutions. While the carry-lookahead and block carry chains offer much higher performance in the form of non-linear speed degradation as the number of cells in the chain increases, their area increases exponentially. The simple carry-skip chain is a natural fit for reconfigurable fabrics because, although it possesses a linear delay model, it also has the advantage of a linear area model and is easily partitioned at each cell in the chain and on cluster transitions. Ripple carry, carry-skip, 2-level carry select, and 2-level Brent-Kung carry chains are shown in Figure 2.4. The logic and interconnection complexity of the ripple and skip chains are contrasted with those of the Brent-Kung and 2-level carry select. While the more complex chains may preform arithmetic operations much faster for chains of greater than 16 cells, they loose flexibility because a cell's

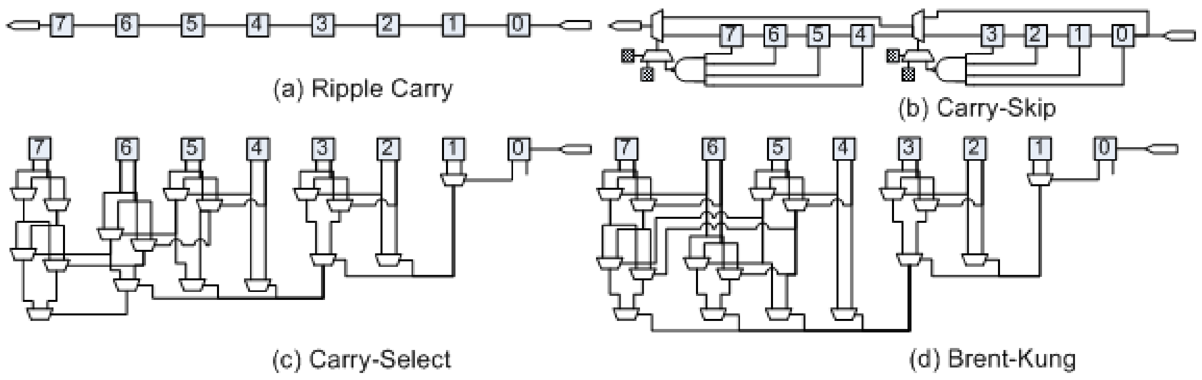


Figure 2.4 Advanced carry chains.

position in a chain is a more important factor throughout the technology mapping, clustering, placement, and routing phases of the design flow.

Special DSP-centric reconfigurable fabrics have also been proposed that reduce the complexity and area of each LE so that each cell implements one bitslice of an arithmetic operation with no extraneous logic or configuration bits [53]. However, all these specializations designed to improve arithmetic are severely degrading to non-arithmetic Boolean operations. According to [53], even purely DSP applications average about 25-30% random logic with arithmetic operations accounting for approximately 60% of logic on average, and the remaining design space devoted to multiplexing. The cumulative effect is that, while 60% of DSP logic is arithmetic, 40% is not. Recall from Table 1.1 that the average prevalence of arithmetic in a sampling of designs is 14.2%. The simple fact is that non-arithmetic LEs dominate the design space for the majority of designs, and continue to make up a significant portion of LEs in DSP applications.

Table 2.1 gives observed component delays for commercially available architectures. These values represent the parameters used to estimate design performance available in Xilinx and Altera architectures and design tools. The standard wire delay used for a carry chain is $0ps$ in all cases, while the variable routing delay typically lies in the range of $300ps$ to $1.5ns$ across all process technologies and architectures. LUT and carry chain logic latency is in all cases significantly smaller than that of the variable routing delay. Additionally, as process technology size shrinks, LUT delay follows suit, but routing delay remains reasonably static due to the trade off between speed and wire size. In most circuits, this correlates to 70% of the delay being due to expensive routing traversals, and most of the remaining 30% due to the LUTs. Almost none of the delay is due to the carry chain logic/interconnection.

Hardwired connections have been explored as a means for saving space within the general routing array by providing dedicated connections between computing components in [19], and have been commercially introduced by the Xilinx 4000 series [75]. The Altera Stratix and Xilinx Virtex II Pro chips also incorporate specific instances of hardwired connections in the form of cascaded LUTs and sum-of-product (SOP) chains, respectively.

Hardwiring connections between computing elements in the realm of FPGAs results in a dedicated connection between 2 or more LUTs. An example is cascaded 4-LUTs with each output driving a subsequent LUT's input and having the ability to be tapped for other programmable connections. The idea is explored in [19] with moderate success, although the constraints imposed by hardwired connections tend to mitigate their benefit. Mapping is performed in two steps: 1) a set of hardwired logic block (HLB) segments is generated and 2) these segments are packed to minimize the number of HLBs in the final netlist. A technology library is used to match the HLBs to a directed acyclic graph (DAG).

Experimentally, three basic HLB topologies are assumed and all LUT outputs are available to programmable routing. The three topologies considered feature LUTs of varying width, including all combinations of 2-LUTs with less than 4 levels, all 3-LUTs with 3 or fewer levels, all 4-LUTs with 3 or fewer levels and 9 or fewer LUTs, all 5/6/7-LUTs with 2 levels, and those with 3 levels and 6 or fewer LUTs. The finding is that hardwiring generally leads to area increases unless the number of hardwired connections is low. By instituting dedicated connections between LUTs, synthesis becomes a more difficult task and only produces small to moderate performance gains. Later in [10] architectural families are considered, wherein a variety of FPGAs might be offered, each tuned with a different hardwired topology. The finding is that hardwiring generally leads to area increases unless the number of connections is low. Typical family structures found to be somewhat advantageous include deep LUT chains (similar to carry chains), or wide topologies where one LUT receives a fraction or all of its inputs from parent LUTs. A family of 8 different topologies is found to outperform a family of one 12-14% in area, and 18-20% in speed. However, this does not account for the cost in maintaining and fabricating large chip families.

Xilinx provides for wide Boolean functions through high-level HDL macros and primitives. The Xilinx V2P library guide denotes how wide homogeneous Boolean expressions (e.g. 16-input AND) can be formed using the carry chain, but synthesis does not identify and implement all such components, nor are the primitives recognized by ISE v9.1. Wide functions are formed by configuring each LUT to an identical function and using them to either propagate *cin* or a programmable 0/1. It is easy to implement a simple homogeneous expression such as a wide AND/OR/NAND/NOR, however it is much more of a challenge to design anything complex. The only recourse is for the designer to implement such expressions, from scratch, using low-level LUT and carry chain primitives.

The Xilinx V2P provides for more complex SOP expressions through a dedicated sum-of-products OR gate, denoted ORCY, located in each slice. The ORCY combines the *cout* of the current slice with the ORCY output of the an adjacent slice not included in the current carry chain. In this manner, it can be combined with the wide Boolean function implemented with the carry chain to form even wider operations of up to 64 inputs. This is not to say that it can implement all $2^{2^{64}}$ possible 64-input functions, but rather can be used to create a subset. Nevertheless, the SOP functionality has been discontinued in the Virtex 4/5 architectures.

In the Altera Stratix architecture [6], a hardwired connection has been allocated that is capable of connecting LUTs residing in the same cluster in a chain. Its operation is similar to chains except that the full K-LUT drives the subsequent LE. LUT chain consumers are identified by Quartus during PNR according to undisclosed metrics. Its functionality is similar to that of the architecture proposed by this work. The differences between the approaches will be outlined in Chapter 3 during the presentation of the chain reuse architecture. However, it is important to note that the Stratix II/III architectures have discontinued the pure LUT chain functionality in lieu of shared arithmetic mode and 3-operand arithmetic.

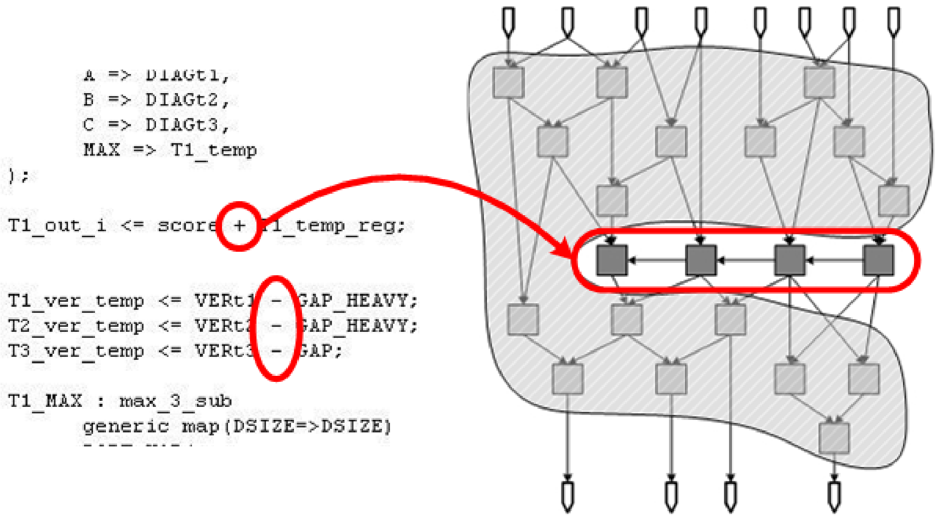
2.2 FPGA Design Flow

The typical FPGA design flow cosists of five primary steps: synthesis, technology mapping, clustering, placement, and routing. Synthesis elaborates a hardware description language (HDL) into a Boolean network. Technology mapping implements the resultant network into the specific device according to its architectural characteristics. Typically technology mapping encompasses different granularities of logic elements, such as LUTs and LEs. The process of mapping LEs into groups is commonly referred to as clustering. Place and route assigns clusters to specific locations within a fabric and configures the routing array to provide the necessary connectivity.

Each step in the design flow directly influences the performance of the resultant circuit and the effectiveness of subsequent steps. For example, if synthesis produces a poor result, technology mapping, clustering, and PNR are of no consequence because their ability to overcome poor results is limited. Changing any component of the design flow cannot be performed in a vacuum–its effects extend to all other areas. Quite often researchers address this aspect by looking ahead in an effort produce intermediate design solutions that are more palatable to consequent stages. This is exhibited in techniques like congestion aware synthesis, which facilitates easier to achieve and higher performing PNR solutions. To assess the changes to any one of these steps, some must be known about the aims and methods of each.

2.2.1 Behavioral specification and Multi-level Synthesis

Perhaps HDL elaboration and generation should be thought of as a step in and of itself. Verilog and VHDL, the two most common HDLs, were originally used to document and simulate designs created at the gate and transistor level. Eventually, the speed with which these languages allowed designs to be created in increasingly complex systems caused them to become the primary vehicle through which circuits are synthesized. However, their accuracy is highly dependent upon the skill of the designer and their ability to convey an idea, concept, or specification to a machine-consumable description. Unfortunately, this reliance on human



```
        A => DIAGt1,
        B => DIAGt2,
        C => DIAGt3,
        MAX => T1_temp
);

T1_out_i <= score + T1_temp_reg;

T1_ver_temp <= VERt1 - GAP_HEAVY;
T2_ver_temp <= VERt2 - GAP_HEAVY;
T3_ver_temp <= VERt3 - GAP;

T1_MAX : max_3_sub
        generic map(DSIZE=>DSIZE)
```

Figure 2.5 A carry-chain induced partition.

intervention and ingenuity at the nascent stage of the design flow has a decisive impact on the the performance of the application.

One of the major concerns of software engineering is to reduce programming errors in software and increase its reliability. This applies to HDL designers with respect to their ability to correctly map a behavior to a component. For the most part, the designer and HDL are responsible for identifying the higher granularity structures of the design, such as dedicated multipliers, block RAM, and arithmetic chains. HDL can describe such structures in a way that an elaboration/synthesis tool can infer them, or macros and primitives can be used to ensure their incorporation. Figure 2.5 conceptualizes how HDL arithmetic operators explicitly define an arithmetic carry chain in a Boolean network.

Arithmetic chains and other higher granularity structures essentially partition the design in the same way as sequential logic (such as FFs) and primary I/O. The corresponding effect this de facto partitioning has on a Boolean network during synthesis, technology mapping, clustering, and PNR is largely undetermined. Fewer degrees in freedom could just as easily aid a largely heuristic design flow by allowing the exploration of a larger fraction of the solution space in less time, as it could remove constraints and foster higher performance. In fact, Quartus II employs incremental compilation in which parts of an application can be highly optimized individually or intellectual property reused, and protected from modification by subsequent design flow traversals [5]. The hardwired FPGA outlined in Section 2.1.2 and presented in [10] indicates that replacing routing with hardwired connections is both good and bad. Using non-programmable, dedicated routing structures helps increase speed, yet introduces enough additional complexity that it also increases design area.

Synthesis takes the elaborated behavioral description of an application and optimizes its Boolean network. If the HDL specifies a sequential design, the associated latches and flip-flops (FFs) are implemented, causing the design to be partitioned such that all cycles in the design are disrupted by a memory structure. The result is a register transfer level (RTL) specification of Boolean gates and sequential logic structures. The goal of synthesis is to optimize the network, or each partition of the network, such that delay (maximum logic depth) and area (number of gates) are minimized.

Two-level synthesis is the process of constructing a network using product-of-sums or sum-of-products descriptions of each output and is an ideal fit for Programmable Logic Arrays (PLAs). While the two-level approach is effective for small networks with limited I/O, it quickly leads to unmanageable implementations requiring too many resources as circuit sophistication increases. Multi-level synthesis is more suited for today's complex circuits because, instead of forcing Boolean functions to be implemented as wide two-level functions, multiple levels are allowed. Multi-level synthesis typically incorporates two-level techniques, coupled with algebraic factorizations and other heuristic strategies. Current state-of-the-art multi-level synthesis tools for FPGAs are those available commercially. Traditionally, the most common academic tool has been SIS [66], which was initially released in 1992. One limiting factor of SIS is that does not support the today's common dedicated structures, such as arithmetic chains, during netlist optimization.

Recently, SIS has given way to ABC [59] as the synthesis tool of choice for academics. ABC is characterized by its integration of logic synthesis, technology mapping, and retiming [60]. ABC uses AIGs (multi-level logic networks composed of two-input ANDs and inverters) as an internal representation to simplify the data structures and allow a natural transformation to functional representations such as binary decision diagrams (BDDs) and SIS's traditional sum-of-products (SOPs). ABC primarily uses Boolean satisfiability (SAT) as its optimization technique, and extends it into technology mapping.

Progress in the synthesis/mapping realm has slowed significantly recently, with few new contributions having been made since the 1990s. The optimality of current solutions is studied in [23] using benchmarks with known optimal synthesis and technology mapped solutions. Work in [23] presents a method for producing benchmark circuits of two types, Logic synthesis Examples with Known Optimal (LEKO) solutions, and those with Known Upper bounds (LEKU). The performance of synthesis and technology mapping tools, such as SIS/DAOmap and ABC mapper is often measured relative to other techniques, not versus benchmarks with known optimal solutions. The popular MCNC suite [76] has been used pervasively throughout the CAD community in the design of algorithms. However, what has occurred is that synthesis and mapping algorithms have become very efficient at producing solutions for that specific set of circuits, resulting in the stagnation of synthesis research and few new innovative solutions.

A similar performance plateau in the performance of ASIC-specific algorithms occurred once solutions reached 1.66-2.53x optimal. Soon after the realization of a plateau, a rush of new techniques, including FastPlace [72], rapidly reduced VLSI routing to roughly 1.2x optimal. In general, small improvements relative to existing techniques get exposed as still far-from-optimal improvements. Using ABC, SIS/DAOmap, Xilinx ISE, and Altera Quartus II to solve LEKO and LEKU circuits, the best academic and industrial FPGA synthesis and technology mapping tools are shown to produce solutions $\approx 70x$ optimal area and, in some cases, $\approx 500x$ larger than known upper bounded examples. Results indicate FPGA synthesis solutions are far from optimal.

2.2.2 Technology Mapping

There are many variations on technology mapping in literature. The four common goals of technology mapping are: 1) minimize delay (polynomial [20]), 2) minimize area (NP-Hard [30]), 3) minimize power consumption (NP-Hard [31]), and 4) maximize routability (NP-Complete

[73]), with many techniques trying to achieve solutions of any combination thereof. Over the years, many algorithms have been presented that first attempt to yield a depth optimal result and then try to heuristically reduce area, power consumption, and/or routing complexity. Unfortunately, the optimal solution of multiple performance metrics simultaneously has been proven to be NP-complete [30]. Initial solutions to depth optimization in technology mapping of Boolean networks incorporate libraries and are shown to be NP-Hard [46]. In the DAGON strategy, a Boolean network can be decomposed to a set of fanout-free trees with optimal technology mapping performed for each tree independently using dynamic programming [46]. However, the 2^{2^k} function set implemented by a K-LUT proves too large to be manipulated efficiently. Nevertheless, dynamic programming is shown to be a very capable approach to the problem of technology mapping.

MIS-pga [61] combines layout synthesis with mapping. It performs mapping in two phases: 1) an infeasible network (i.e. one with nodes greater than K inputs) is made feasible by recursively splitting nodes through kernel extraction or classical Boolean function decomposition, and 2) the feasible network is optimized by collapsing pairs of nodes or collapsing clusters of nodes. MIS-pga uses different decomposition techniques, each for a different situation: cube packing, co-factoring, AND/OR decomposition, and disjoint decomposition.

Chortle-crf, presented by [34], is one of the first technology mapping algorithms for LUT-based FPGAs to use bin packing. The work of Chortle-crf is continued in [35] with the goal of reducing the delay of combinational circuits in LUT-FPGAs due to logic LEs. Chortle-d presents a bin-packing approach to finding the minimal depth of a given network, represented as a forest of DFS trees. Chortle-crf is used to map the LUTs efficiently, followed by a reapplication of Chortle-d to the paths in the network that exceed the minimal depth. LUTs with a single output are merged with downstream LUTs if they do not violate the width of the K-LUT. The Chortle-d algorithm is shown to provide a local depth-optimal solution [33].

DAG-Map [18] improves upon Chortle-d by considering the entire network and is optimal for K-LUT trees. A helper algorithm, DMIG, transforms an arbitrary network into a 2-input network with only a $O(I)$ factor increase in network depth, where I is the number of PIs to the network. DAG-map uses K-feasible cones, which consist of a node and its predecessors such that any path connecting a node in the cone to the output node lies entirely in the cone. If the number of inputs to each cone is less than or equal to K, so that any K-LUT can implement a K-feasible cone. The level of the output node is the length of the longest path from any PI node to the given node. The depth of the network is the largest node level in the network. DAG-Map formulates the problem as a covering of a given network with K-feasible cones that are not necessarily mutually exclusive.

The first step in technology mapping with DAG-Map is to decompose the Boolean network, G, and then transform it into a 2-input network (i.e. each gate has at most 2 inputs). DMIG replaces each $K > 2$ gate with a balanced binary tree of the same gate type (gate function must be associative) and yields an increase in depth by as much as $\Omega(lgn)$. The resultant depth of the DMIG transformation is $depth(G') \leq lg(2 \cdot d) \cdot depth(G) + lg(I)$, where d is the maximum degree of fanout in G. DMIG is important because it yields a network that K-bounded, i.e. all nodes possesses less than or equal to K inputs–a requirement for many technology mapping algorithms. It also increases the mapping technique's solution space through node decomposition.

The second step is to apply the DAG-Map mapping algorithm which has two phases: labeling the network to determine each node's level in the final solution and generating the logically equivalent network of K-LUTs. The general approach to DAG-Map is that for any node in a network tree, it joins logic depth p if it can, where p is the maximum level of any of its predecessors, else it must be implemented at level $p + 1$. It relies upon tree structures to ensure a monotonic increase in the number of inputs per node. The advantages of this technique are that it works on the entire network, does not have to decompose the network into fanout free trees, can implicitly replicate nodes to minimize delay, and is optimal when the initial network is a tree but not when it is an arbitrary network. DAG-Map is designed to minimize depth, not area, and accordingly, two post-processing steps are used to reduce the final design without increasing depth. Overall, DAG-Map does better than either MIS-pga or

Chortle-d with the exception of runtime and lays the groundwork for optimal depth technology mapping.

FlowMap [20], a continuation of DAG-map, is the first technique to map an arbitrary Boolean network to a $K - LUT$ architecture with optimal logic depth in polynomial time. It works by using the Max-flow Min-cut algorithm for network flows, which says that the maximum flow through a network is found at the minimum flow capacity cut in the network [32]. Using this, K-feasible cuts are found within the network to combine gates in the same K-LUT at minimum logic depth. FlowMap has the same basic approach as DAG-Map, except that it successfully solves arbitrary networks through incorporation of network flow computation. It is also shown to yield minimal area designs through the similar post-processing techniques to DAG-Map.

The ingenuity of FlowMap is that it uses the application of network flows to solve the monotone cluster constraint [52] for LUT inputs that hinders previous approaches. The number of inputs of a programmable logic block is not a monotone clustering (e.g. mapping) constraint, and therefore the solution of network H does not imply anything about network G where $G \in H$. For H that satisfies some constraint Γ, it is implied that any subnetwork G where $G \subset H$ also satisfies Γ. For K-LUT formation, this is not the case. It may occur that removing a node $v \in H$, where $|input(H)| \leq K$, creates $G = H - \{v\}$, such that $|input(G)| > K$. This occurs if $G \cup \{v\}$ causes one or more of $input(G)$ to be made internal to H. If the clustering constraint is monotone, tools as early as 1969 [52] provide an optimal polynomial time solution. However, until the presentation of FlowMap and its use of network flow, the solution of LUT mapping for arbitrary Boolean networks was sub-optimal. Various approaches could provide optimal solutions for a subset of topologies in polynomial time, but not for arbitrary Boolean networks.

FlowMap works similar to DAG-Map, in that it is a dynamic programming approach that identifies whether or not nodes are capable of joining a K-LUT without causing it to violate K-feasibility. However, instead of relying on the topology to enforce the input monotone constraint, FlowMap uses network flows to establish a minimum height K-feasible cut in an arbitrary network. In the labeling phase of FlowMap, nodes are labeled such that each node has a label that is greater than or equal to all of its predecessors, where the label signifies the optimal logic depth of the node. The variable p is the maximum label of the nodes in $input(t)$. N_t are the predecessors of t, referred to as the cone of t. FlowMap creates a DAG that collapses all of the predecessors of t with label p into t to form t', signifying $LUT(t)$, and then constructing a flow residual graph so that the Max-flow, Min-cut algorithm [32][26] can be applied to find the minimum height K-feasible cut in N_t. If a K-feasible cut can be found, t is labeled p and a K-LUT can be constructed of the nodes designated by the cut and the nodes contained in t'. If a K-feasible cut cannot be found, t is labeled $p+1$ and is implemented in a new K-LUT.

The mapping phase consists of using the labels and the K-feasible cuts found in the first phase to construct K-LUTs. Starting from the POs of the network, K-LUTs are constructed using the K-feasible cuts during labeling. Each node in the network either implements a K-LUT containing it and its predecessors included in the K-feasible cut, or is implemented as another node's predecessor. In this way, not all nodes are explicitly implemented and node duplications are carried out implicitly.

FlowMap and its derivative algorithms have provided the base solution to the technology mapping problem for FPGAs with regard to producing logic depth optimal designs in polynomial time. However, they do so by viewing the FPGA architecture sans the valuable carry-chain resource that has become industry standard. This carries with it the assumption that the nets connecting LUTs are all general routing nets. Yet, circuit delay is not just dictated by the number of logic elements required, but also by the number of routing and carry nets traversed. A static arbitrary net-delay model version of FlowMap has been presented [21], but it necessitates that the delay of each net be predetermined. Carry chains also have the special constraint of being a point-to-point connection between adjacent cells, a constraint that must be addressed if valid implementations with arbitrary net delay FlowMap are to be generated [21].

An extension or alternative to network-flow based technology mapping is cut-enumeration.

In cut-enumeration, all possible K-feasible cuts are computed for the network, and each are considered relative to all others in an attempt to find the best combination of cuts. Though cut-enumeration considers both deal and area during mapping, its drawback is that it is very high in computational complexity, requiring the consideration of $O(n^K)$ cuts. Cut enumeration strategies are described by how they perform cut generation, the process of computing cuts, cut ranking, which aims to compare the cuts of a given node according to optimization objectives, and cut pruning, which describes the process of eliminating less desirable cuts [25].

CutMap [22], one of the first techniques to employ cut enumeration, does so under the auspices of network flow computation. Using the optimal depth of a network as a bound, CutMap uses alternative cuts and heuristics to create the fewest number of LUTs possible. The first phase of the algorithm uses network slack computation to first determine a minimum height or minimum cost cut for each node. Predicted cost, using maximum fanout-free cones (MFFCs), is determined according to the likelihood of a node being implemented by an LUT during mapping. Roots of large MFFCs are likely to be contained in an LUT, with a K-feasible cut through an MFFC will result in more nodes being implemented. In the second phase, actual cost is computed optimally in $O(2Kmn^{\lfloor K/2 \rfloor+1})$, where m is the number of edges and n the number of nodes. The average CutMap solution yields 0.78x LUTs than FlowMap.

Accelerating the cut ranking procedure is done in [25] with metrics such as duplication-free mapping, whose goal is to prevent separate maximum fanout-free cones from implementing common subtrees. The ability to determine how to optimally map a multiple-output node is the primary reason why area minimization is NP-hard [30]. Cut pruning reduces run time by selecting the most desirable cuts created during generation and ranking. Pruning is made more efficient by eliminating undesired cuts before their generation by identifying non-essential sub-cuts (branch and bound). Cuts can be subsets of many other cuts, but non-essential sub-cuts belong to no cuts that possess a “best” rank, and thus can be safely eliminated from consideration. Likewise common sub cuts can be used by descendants to construct their cuts. Results for the cut enumeration techniques presented in [25] indicate it produces reduced or area neutral designs at a fraction of the runtime of other common cut enumeration approaches.

One of the most recent cut enumeration methods, presented in DAOmap [17], also stresses the number of potential node duplications during enumeration to more accurately map area-minimal designs. Again, the cuts of the target node's predecessors are used to form the target node's cut. A binate covering algorithm gives an optimal solution in exponential time and heuristics are used to make the runtime more reasonable. DAOmap progresses in three stages: 1) use potential duplicates to minimize area from a global point of view, 2) enforce timing to relax non-critical paths, and 3) perform iterative cut selection. The worst case number of cuts for any node is $O(n^K)$, making cut pruning as outlined in [25] critical, especially for $K \geq 7$. Area cost for each node v is propagated as the predicted area of each $U_c + \sum_{u \in input(C_v)}[A_u/|output(u)|]$, where U_c is the area contribution of cut C_v and A_u is the area of each $u \in input(v)$ amortized over its fanouts. In this fashion, when the fanouts of any node u re-converge, the area of the cut is their sum. Combining area cost with the constraint of optimal mapping delay using arrival time, DAOmap produces 16% area reductions over CutMap while producing a run time speedup of 24.2x.

ABC [59] integrates logic synthesis and technology mapping using AIGs to facilitate the use of cut-enumeration. The contribution of ABC is that node implementations determined each optimization pass are remembered for future use, a prospect made possible by the simplicity of the AIG representation. In this manner, the solution of multiple metrics can be achieved over multiple iterations. The number of iterations and the history of node implementations can be varied to change the number of re-mappings that are pursued. ABC also uses Boolean SAT to perform technology mapping.

An alternative approach to cut-enumeration and network flow based technology mapping is Boolean matching [9], as used in ABC. While an effective mapping technique, Boolean matching is potentially capable of identifying chains in a Boolean network. Boolean matching techniques attempt to map an n-input logic function to a m-pin hardware module through to Boolean satisfiability (SAT) matching techniques. The work in [65] outlines the basics of Boolean SAT and addresses some of the obstacles to its use. It is characterized by a 2-stage approach that first coarsens the network by assuming that every m-pin hardware module is capable of implementing a m-input function and second provides detailed acceptance/rejection of stage 1 results.

Assuming that a m-pin hardware module can implement all m-input functions is not realistic, but allows a partitioning of inputs to be formed and eliminates large subsets of solution space that are infeasible even for an m-input function. Boolean SAT is targeted toward programmable logic blocks consisting of arbitrary topologies, such as two K-LUTs sharing $K-2$ inputs (as in the Stratix II ALM). In this example, two LUTs sharing inputs can not implement all $(K+4)$-input functions, but rather just a subset. Boolean SAT techniques typically use branch and bound approaches to identify infeasible solutions, as well as those that are closely related, and eliminate them from consideration. The second, detailed phase eliminates solutions from phase one that not SAT. The whole process is accelerated by eliminating as many solutions as possible and creating valid pin partitions in phase one. Experiments using the proposed technique demonstrate a 340% run time improvement and 27% additional mappings over previous Boolean SAT methods. A chain can be viewed as a $(K-1) \cdot L+1$ pin hardware block, thus Boolean SAT techniques offer one possible avenue for their mapping. However, even for modest chains of length $L = 16$ and $K = 4$, the solution of a 49-input function is required–a far cry from the 11-input functions to which Boolean SAT is currently applied. Chains have the potential to quickly overwhelm Boolean SAT techniques.

Similar to synthesis, since the 1990s few new notable technology mapping algorithms have been presented in literature. Tools and algorithms worth mentioning, such as IMAP [56], Hermes [70], DAOmap [17], and ABC mapper [59] attempt to further reduce the area impact of logic depth optimal designs. Optimality of synthesis and technology mapping [23][54] studies indicate that currently, given optimal synthesis solutions, technology mapping tools are capable of producing results ranging from Quartus' 1.03x to DAOmap's 1.22x of optimal. However, the technology map is usually hindered by the inability of synthesis to produce a good result on which mapping can be performed. Commonly, techniques that blur the line between synthesis and technology mapping yield better results, as in Quartus II and ABC.

Overall, approaches to technology mapping present useful techniques in minimizing logic depth in FPGAs, but often use an incorrect notion of where the delay in the combinational circuit is encountered–the bulk of the delay is not in LUT depth, but rather interconnection delay between those LUTs. This assumption is based on the idea that there are no quick connections between LUTs, and thus minimizing the LUT depth minimizes the delay through the circuit. Re-timing and placement-aware techniques have begun to address this fact, but do not provide a direct avenue for assigning logic chains optimally. No tools currently address chains directly, instead deferring to HDL macros.

2.2.3 Logic Clustering

As FPGAs have evolved, it has been shown that grouping similar LEs to share resources can lead to increased circuit efficiency. These groups of LEs, known as clusters, share control signals and general routing access, as well as connect to each other using local routing, as in Section 2.1. Clustering reduces global overall design complexity, and thus simplifies the problems of placement and routing [12]. The island-style FPGA, presented in [50], has become commonplace among commercial architectures.

At the time of the publication of [11], not much work had been done on island-style FPGAs. It boasts itself as the first work to investigate the use of logic clusters within a 2-level hierarchy in a flat FPGA architecture. Key cluster characteristics include the number of LEs per cluster N, and the total number of distinct inputs to each cluster I. The cluster packing tool it presents, Vpack, has been augmented to account for timing information in its current incarnation of T-Vpack.

T-Vpack [12] works by selecting an LE with the most used inputs (as cluster inputs are the hardest resource to come by), and then greedily selects the LE with which it has the most in common until all cluster I/O or LE capacity is exhausted. In its simplest form (VPack), T-Vpack chooses LEs to join a cluster based on the *attraction* of LE v to the cluster C, i.e. the size of the intersection of the set of inputs of the LE and the set of unique inputs of the cluster, as per Equation 2.1. The greater the size of the intersection, the more signals the LE and the cluster have in common. This correlates to greater sharing of signals which utilize the interface between local and general routing. It also implies that connections formed amongst LEs within the same cluster are desirable, but not expressly encouraged. In the Vpack approach all LEs are tested and the LE with maximum attraction is chosen to join the cluster.

$$attraction(v) = \frac{|nets(v) \cap nets(C)|}{I + N + M} \tag{2.1}$$

Gain is a useful metric in associating the best LEs for inclusion in the same cluster. However, architectural constraints can lead to the disqualification of the highest gain LE. An LE is allowed to join a cluster if it subscribes to each of three constraints:

1. Its inclusion does not exceed the maximum number of external cluster inputs, I.
2. Its inclusion does not exceed the maximum number of external cluster clocks, M.
3. The total number of LEs has to be less than the cluster size, i.e. $|v \cup C| \leq N$.

T-Vpack also incorporates a gain function that combines attraction with timing estimation. Timing criticality is defined using the notion of slack. The slack between two nodes, u and v in a timing graph is computed using Equation 2.4. It measures the delay from PIs to a node u with $T_{arrival}(u)$, and the delay from POs to u, given the max arrival time (i.e. critical path), in $T_{required}(u)$. Given arrival and required time at each node, the slack between any pair of timing graph nodes can be computed, and indicate the base criticality of any given path through the network using Equation 2.5.

$$T_{arrival}(v) = max_{\forall u \in input(v)}\{T_{arrival}(u) + delay(u,v)\} \tag{2.2}$$

$$T_{required}(v) = min_{\forall u \in output(v)}\{T_{required}(u) - delay(v,u)\} \tag{2.3}$$

$$slack(v,u) = T_{required}(u) - T_{arrival}(v) - delay(v,u) \tag{2.4}$$

$$baseCrit(v) = \forall_{u \in input(v)} 1 - \frac{slack(v,u)}{MaxSlack} \tag{2.5}$$

$$\tag{2.6}$$

Base criticality (Equation 2.5), the dominant component of criticality, reflects the maximum critical input into LE v and is normalized to the most critical net in the network.

However, the base criticality is not unique in the network, and in most cases is quite common. Therefore, a two-level tie breaker system is incorporated and scaled with ϵ and combined with base criticality to form Equation 2.8. The first tie breaker measures the number of critical paths on which v resides. Equation 2.7 is multiplied by a nominal value of $\epsilon \approx 0.01$ to add a bias toward nodes with more critical paths in common with cluster C. The second tie breaker, $D_{PI}(v)$, accounts for node v's location within a network path, and is scaled by $\epsilon^2 \approx 0.0001$. $D_{PI}(v)$ measures the maximum distance, in LEs, that node v is from circuit PIs. Alternately, distance from POs is equally valid, the goal being that a path be clustered from terminal to terminal and not beginning in the middle. If a path's intermediate nodes are clustered before its ends, the path will likely require a greater number of clusters to implement, and more general routing traversals, ultimately increasing the cost of the clustered solution.

In T-Vpack, the gain of adding LE v to cluster C is given by Equation 2.9. It combines attraction and criticality with a scaling factor, γ, that is experimentally determined as 0.75 [12]. If $\gamma = 0$, then the clustering is the original Vpack, conversely, if $\gamma = 1$ clustering is purely connection driven. The normal operation of T-Vpack fills a cluster until resources are expended, however, in some cases, clusters with fewer than N LEs but I inputs can accommodate more LEs. If a cluster reaches a point where no other LEs can be accommodated because I has been reached, but $|v \cup C| < N$, T-Vpack invokes a hill-climbing mode. LEs are added to a cluster even if it becomes infeasible regarding I, because it may occur that subsequent LEs can return the cluster to viability. If a net's source terminal is added to a cluster containing at least one of it's sink terminals, an input is saved. In some cases adding LEs will cause a net to become completely internal to the cluster, also saving an input pin. Hill-climbing mode has been shown to improve the clustering solution's logic utilization by 1-2%.

$$critPaths(v) = inCritPaths(v) + outCritPaths(v) \tag{2.7}$$

$$criticality(v) = baseCrit(v) + \epsilon \cdot critPaths(v) + \epsilon^2 \cdot D_{PI}(v) \tag{2.8}$$

$$gain(v, C) = \gamma \cdot criticality(v) + (1 - \gamma) \cdot \frac{|nets(v) \cap nets(C)|}{I + N + M} \tag{2.9}$$

Based on a LE using a 4-LUT and considering early FPGA architectures, a cluster containing N LEs needs $K \cdot N$ inputs for complete global connectivity (i.e. every LUT input can reach the general routing array). Experiments using T-Vpack indicate that the full connectivity found in Xilinx and Altera FPGAs is likely over-aggressive, and for $K = 4$ only $I = 2 \cdot N + 2$ inputs are required for 98% cluster utilization. Additionally, cluster sizes of 4 are about 5-10% more area-efficient than no clusters. Other works find a cluster of size 4 LEs to be the most area efficient, while 5 and 6 inputs are found to offer the best performance. A revisit of N and I for more recent FPGA architectures in [4] finds that $I = \frac{K}{2} \cdot (N + 1)$ achieves 98% cluster utilization and $N = 8$ is the most efficient clsuter size.

To more successfully address routing complexity during clustering, R-pack [14] institutes a routability-based scoring function. It operates on the precept that by addressing the factors that affect routability, the cumulative routing cost can be reduced, as expressed by Equation 2.10. Total routing cost, given by Equation 2.10, is designed to increase as the number of pins of a net increases, but the rate of increase diminishes as nets become larger. The more pins a net contains, the more difficult to route, and the higher $\alpha(x)$. However, once a net becomes sufficiently large the increased difficulty in routability caused by adding one more pin becomes less and less [16].

$$R_{cost} = \sum_{x \in Nets} N_x \cdot \alpha(x) \tag{2.10}$$

$$\alpha(x) = 2 - \frac{1}{pins(x)} \tag{2.11}$$

R-Pack is built using the T-Vpack clustering engine, and uses a slightly different method of computing gain. The scoring system assesses the impact of LE v joining cluster C, where each net incident on v gets 1 point for every edge consumed, 1 point for input-pins saved, and 1 point for output pins saved. On the other hand, new nets to the cluster result in a 1 point deduction for a new input pin, and no points for a new output pin. The score of an LE joining the active cluster is the sum of the scores of its nets. The R-Pack scoring criteria addresses the routing cost of Equation 2.10 only indirectly. Place and route experiments indicate R-Pack is able to reduce channel width by 16.5% over Vpack, on average.

iRAC [67], aims to address routing cost through bias toward low fanout nets and clustering with the underlying FPGA architecture in mind. Rent's rule is used to govern the population of clusters such that each cluster's connectivity mirrors that of the FPGA routing architecture. To achieve this, cluster utilization is limited through restricting the number of inputs used per cluster explicitly. In this manner, congestion is limited because the clustering solution more closely matches the connectivity of the architecture. Place and route experiments indicate iRAC is able to reduce channel width over R-Pack and T-Vpack by 35%, on average.

Work in [71] also clusters with the architecture in mind, but instead limits the number of LEs per cluster. It uses the T-Vpack and iRAC approaches in conjunction with limiting cluster utilization to address routing channel width constrained FPGAs. Through localized depopulation of clusters, an unroutable design can become routable. By clustering with the underlying architecture in mind, the perception of the clustering solution more closely matches architectural realities.

The cluster-seed approach has been adopted by many publicly available tools. It works by populating an active cluster until no additional LEs can be accommodated. An empty cluster is first seeded with an LE that influences which subsequent LEs are added to the cluster. Thus, the seed LE becomes of particular importance, as it ultimately dictates how each cluster is populated. iRAC, T-VPack, [71], and RT-Pack each use the cluster-seed approach. T-VPack, chooses the cluster seed as the LE with most external inputs, attraction, or with the highest timing-based criticality [12]. RT-Pack mixes routing cost impact into the equation [15].

Other clustering approaches deviate from the cluster-seed model through the use of more complex approaches. The tool presented in [47] aims to consider intra-cluster and inter-cluster resources separately, and then combine these factors to in an overall cost function. The clustering solution is generated using simulated annealing and requires on the order of $500 \times (Total\ LEs)$ iterations to arrive at a solution. Place and route experiments indicate an average 19% reduction in channel width, 13.5% reduction in chip area, and a 9.3% reduction in critical path delay over T-Rpack.

A *top-down* approach presented in [58] uses hMETIS-Kway [45] in a 2-step algorithm which first produces k-partitions of the LE network, and in the second phase applies resource constraints to create a feasible cluster network through relocating LEs. Place and route experiments indicate an average 15% reduction in routing channel width over T-Vpack and T-Rpack with nearly identical critical path latency.

Clustering solutions abound in literature, but none deal with chains explicitly. Chains are assumed to be mapped from head to tail, according to HDL, regardless of routability. The primary reason for this is that chains effectively dictate the clustering solution. Little flexibility us afforded by chains because all of their members must be clustered contiguously. Chains can be segmented such that whatever inter-LE gain they do possesses can be taken advantage of.

2.2.4 Place and Route

Commercial PNR tools are highly proprietary in nature, largely because they are commonly tailored to a specific architecture. The most prevalent open source academic tool, VPR features automatic architecture generation using designer-specified foundry parameters, simulated annealing based placement, and routability or timing driven routing. VPR incorporates a routing resource graph which describes switches as edges, and clusters, IO pads, ports, and routing channels as nodes. VPR is presented in its entirety in [12] and commonly used in conjunction with T-Vpack.

The placement procedure uses simulated annealing to place clusters according to three different cost functions. The effectiveness of simulated annealing depends on the initial temperature, the number of moves per temperature, the variation of temperature throughout annealing, and the termination of the process, which are known collectively as the schedule. The VPR schedule is drawn from several previous works, including [42], [51], and [69]. They have been combined to form an adaptive schedule that tailors itself to each unique problem. VPR's contribution is to adaptively tailor the schedule so that more time is spent when a significant fraction of moves are being accepted, and less time spent at temperature extremes where all or no moves are being accepted. Placement cost is determined through a combination of timing, bounding box, and routing congestion information.

In general, routing algorithms come in two styles. The first style is referred to as global-detailed, and consists of one step during which the complete routing path of a net is determined, including specific wires. The second style is a two-step algorithm that first performs global routing to the pins and channels that will be used, and in the second step determines the exact wires. Generally, performing global and detailed routing in two separate steps is infeasible because the solution of the first step places too many constraints on the second, potentially resulting in an unroutable design or one that is too computationally expensive to achieve in real time. Global-detailed routing is usually capable of creating feasible solutions in reasonable time.

Routing is performed through the use of a routing resource graph, shown in Figure 2.6 which has been partially excerpted from [12]. Figure 2.6(a) depicts a typical pair of clusters in a FPGA, while (b) is its corresponding resource graph. Each node in the graph is given capacity corresponding to the number of tracks though which connections can be routed on the particular resource. Sink nodes, which represent LEs, are given a capacity of K while channels (x, y) are given capacity equivalent to the number of wires they incorporate. Cluster outputs serve as source nodes, and have a capacity of one. All other structures, such as I/O pads and cluster ports $(i1, i2, o1)$ are modeled with a capacity of one. In this manner, the connectivity of the FPGA routing architecture can be modeled and the nets of a placed design routed.

To perform timing analysis and incorporate it in PNR, the ability to accurately model the delays of the architecture is paramount. VPR incorporates a timing graph that models each of the wire and component delays present in the FPGA defined by the user. Figure 2.7 depicts a simple circuit consisting of LEs, input pins, and the nets that connect them. In the graph, nodes have no delay, while edges impose it. Therefore, an LUT would be modeled by $K + 1$ nodes; K-input nodes, and 1 output node, with the delay from any input to the output modeled by the edges between the two.

In a programmable routing architecture, the delay of each net varies and is dependent on the switch boxes and wire lengths realized by the final routing solution. To use timing information during placement, net delay is estimated by routing a single net between two

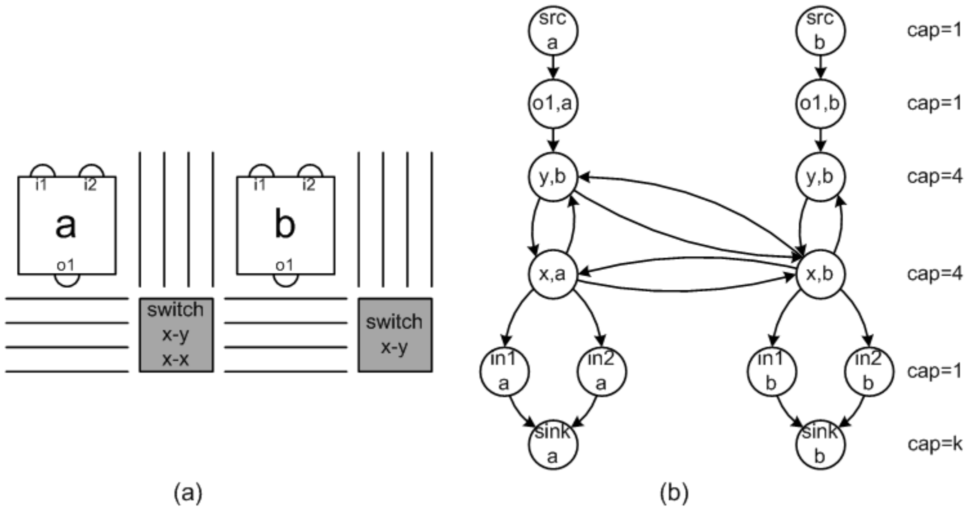


Figure 2.6 Typical routing channel (a), and (b) its corresponding routing resource graph.

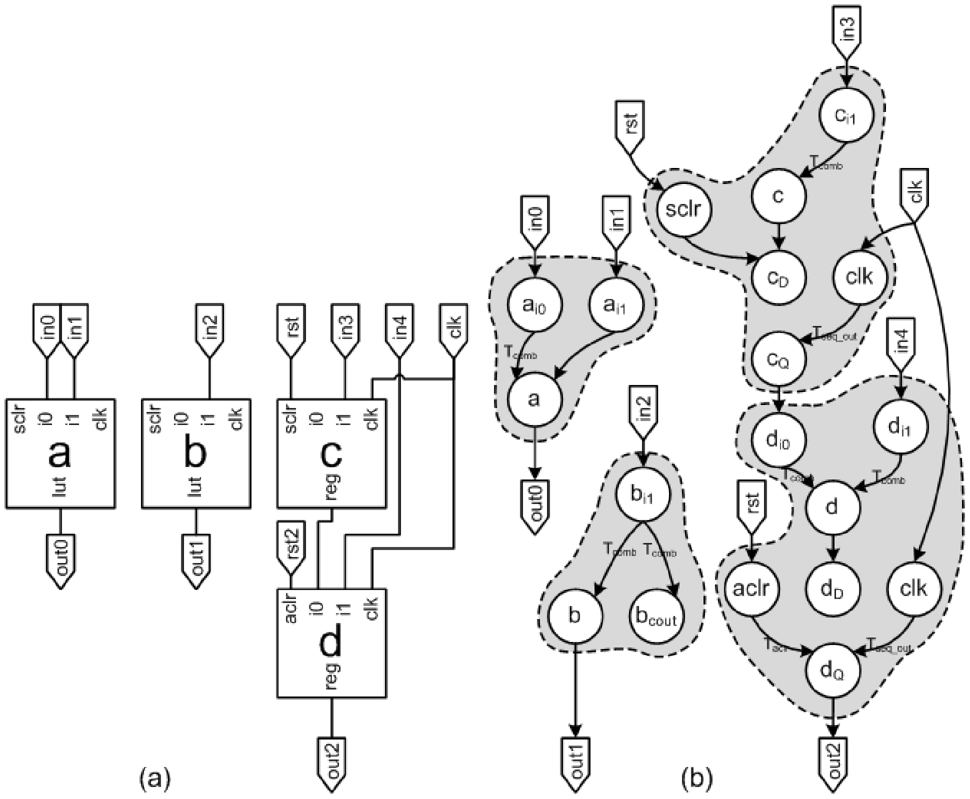


Figure 2.7 Timing graph (a) circuit, and (b) its timing graph realization.

random clusters in the array, measuring the delay, ripping the net up, an then repeating the procedure. An average routing delay can ascertained for clusters of varying distances from each other, allowing path and net timing driven placement to occur. In contrast, the component delay of each LE operating in a specific mode is static. Intra-cluster connectivity and LEs are VPR modeled with the following timing parameters:

- T_{comb} - The delay from an LE input pin, through the LUT and output multiplexer, to the output pin.
- T_{seq_in} - The delay from an LE input pin, through the LUT to the FF input, and including the setup time (T_{su}) of the FF.
- T_{seq_out} - The delay from the LE FF output, through the output multiplexer and to the output pin, including the clock-to-Q time (T_{CO}) of the FF.
- T_{sopin_sipin} - The delay from a LE output pin to an LE input (local routing).
- T_{cipin_sipin} - The delay from a cluster input pin to an LE input pin.
- T_{sopin_copin} - The delay from a LE output pin to a cluster output pin.

The routability algorithm VPR uses is an iterated maze router based on Pathfinder negotiated congestion routing [29]. The cost function incorporates the base cost, historical congestion, and present congestion elements from the Pathfinder algorithm and a unique element referred to as bend cost. The base cost quantifies the impact of using a particular resource, and the historical and present congestion cost give the route a measure of how used a particular resource is currently and in previous routing iterations. The bend cost is designed to encourage general routing to use as few orthogonal (x, y) connections as possible because they tend to prevent the use of long lines and complicate detailed routing. Consequently, the bend cost makes it easier to generate a detailed routing solution. Because global-detailed computes the routing solution in one step, it's use of the bend parameter is superfluous, and is considered 0.

VPR's timing-driven routing algorithm is also based on the Pathfinder algorithm, but improves upon it by incorporating an Elmore delay model and by dynamically changing routing resource costs. The advantage of the Elmore delay model over a linear delay model is that it more accurately predicts routing connections that use chains of pass transistor instead of buffers, or pass transistor based multi-fanout nets. Because pass transistors are commonly used to implement multiplexer, subset, Wilton, and universal channel switch boxes, the Elmore model is an ideal fit for FPGAs.

Using one of the two available routing algorithms, VPR generates an interconnection topology to fit the given design. The basic interconnection of each topology remains the same, but the number of tracks available in each channel (channel width) is computed via a binary search until the minimum width that can accommodate the design is determined. Minimum channel width is a useful performance metric for the routability of a design, but only when binary search routing is employed. Another binary-search dependent metric is total transistor area. It serves as a measure of the transistor area implemented by the switches and connection boxes in the routing array and changes as the channel width increases. A metric not explicitly dependent on binary search routing is total wire length, which measures the total length of wire used for routing a design. Designs requiring a higher channel width, total wire length, or area typically have more nets, higher connectivity, or both.

In dealing with chains during place and route, the only deviations from normal behavior are that all clusters of a chain must be placed adjacent to each other, and the chain nets must be implemented by the special chain resources. Routing between LEs residing in the same chain is trivial, because there is only one valid possible route to choose from. The process of placement is a bit more complex because chain LEs and independent LEs must be considered concurrently. VPR placement works using a simulated annealing technique, where clusters are swapped, the improvement of the move assessed, and the move correspondingly accepted or rejected. The technique for single clusters can be adapted to a chain of clusters, as mandated by the presence of chains, by swapping with group of clusters of equal size. The technique outlined in [8] indicates how chains can be handled in the VPR environment. The process by which chains are placed is as follows:

1. Clusters in the same chain are placed consecutively from column bottom to top.

2. A cluster is selected at random, and if it is a member of a chain, it must be swapped along with the other members of its chain.
3. A move will be determined legal if it does not violate any physical constraints of the chip, or sever existing chains.
4. An illegal move will not be considered as a rejected configuration during simulated annealing. It is simply discarded and another random cluster pair is chosen.
5. An accepted swap results in the movement of the entire contiguous chain.
6. If a swap of L clusters is accepted/rejected, it counts as a gain/loss of L.

The swap source and destination clusters dictate the range of clusters to be exchanged. Let there be two randomly selected clusters $A_i \in A$ and $B_j \in B$, where A and B are chains with lengths L_A and L_B, respectively, and $0 \leq i \leq L_A - 1$ and $0 \leq j \leq L_B - 1$. The (x, y) coordinates of A_i and B_j are swapped directly, requiring that the remainder of the ranges of A and B be established by the maximum sizes of their terminals. The number of cells at the tail of each chain to be swapped is $max\{L_A - i, L_B - j\}$, while the number of cells at the head of each chain is $max\{i, j\}$.

Converting relative chain coordinates to the (x, y) coordinates of an array of clusters yields Equation 2.13 and Equation 2.13. Figure 2.8 depicts an example swap of chains A and B and their surrounding regions. The head swap consists of clusters $(2, 1)$ through $(2, 3)$ and $(6, 1)$ through $(6, 3)$ because the maximum length is given by $L_{head} = max\{4 - 1, 4 - 4\} = 3$. Likewise, the tail swap consists of clusters $(2, 4)$ through $(2, 6)$ and $(6, 4)$ through $(6, 6)$ because the maximum length is given by $L_{tail} = max\{5 - 4, 6 - 4\} = 2$. This corresponds to a total swap length of $L_{chain} = 5$ clusters.

$$L_{head} = max\{y_{A_i} - y_{A_0}, y_{B_j} - y_{B_0}\} \tag{2.12}$$

$$L_{tail} = max\{y_{A_{L_A - 1}} - y_{A_i}, y_{B_{L_B - 1}} - y_{B_j}\} \tag{2.13}$$

$$L_{chain} = L_{head} + L_{tail} \tag{2.14}$$

The clusters incorporated in the swap must contain chains A and B, but can also include

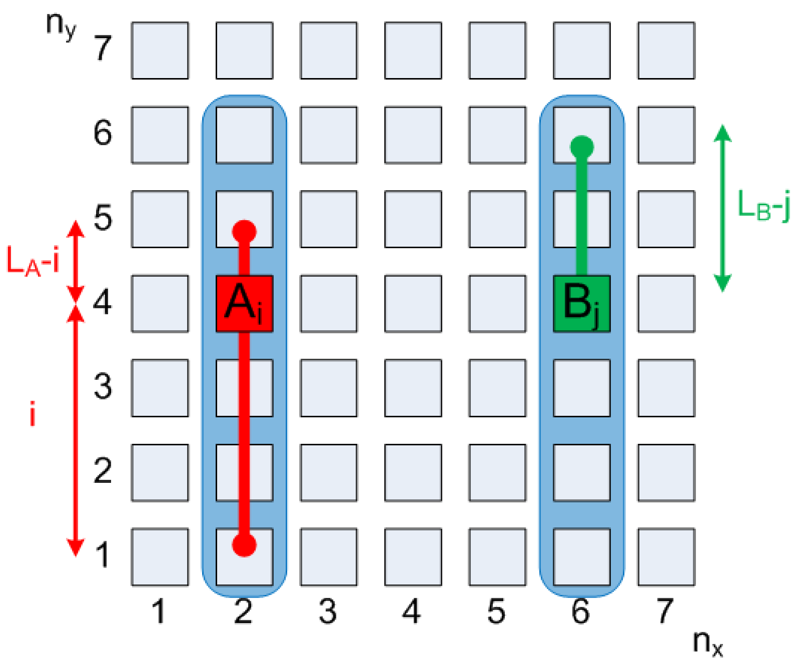


Figure 2.8 Source and destination swap regions using relative chain position.

independent clusters, unrelated whole chains, and empty clusters. The total number of clusters (empty or populated) swapped is given by $max\{i,j\} + max\{L_A - i, L_B - j\}$. The only constraints are that all cluster ranges must reside within the bounds of the array, the regions cannot intersect, and no chain can be severed. Any region that violates any constraint is deemed invalid, but not a rejected swap. An invalid swap is simply discarded and new randomly selected swap clusters are chosen; it is not considered a simulated annealing accept or reject. Any accepted swap results in a cumulative gain of $\frac{max\{i,j\}+max\{L_A-i,L_B-j\}}{2}$ relocated clusters, according to whatever metric is employed. The routing of chains is trivial, as the only available resource for routing is the chain net between clusters.

CHAPTER 3. ENABLING THE ARCHITECTURE

FPGAs typically use ripple-carry schemes, or variations thereof, for area efficient arithmetic. The Altera Stratix and Cyclone architectures [6] use a carry-select chain, characterized in Figure 3.1(a). An LE operating in $(K-1)$ mode contains two $(K-1)$-LUTs, one driving a chain net through the *cout* port, and the other driving the general routing array trough *gr*. Chains using $(K-1)$-LUTs are also referred to as *sub-width*. These LEs facilitate chains as in Figure 3.2(a). The Stratix also incorporates an LUT chain, characterized in Figure 3.1(b), wherein one K-LUT simultaneously drives the same logic function to the chain net and general routing. The Stratix LUT chain uses an auxiliary connection between LEs, separate from the carry chain, to achieve K-LUT mode and form heterogeneous chains as in Figure 3.2(b). Chains using K-LUTs are also referred to *full-width*.

3.1 Carry Chain Reuse Logic Element

To realize the full potential of the carry chain without extraneous interconnection, a novel architecture is necessary. A modified carry-select architecture presented in [36] operates in

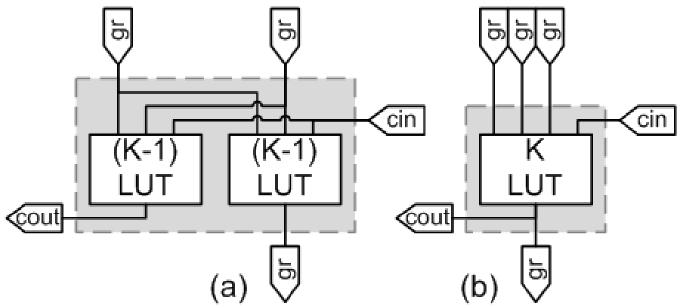


Figure 3.1 (a) *(K-1)*-LUT mode, (b) K-LUT mode

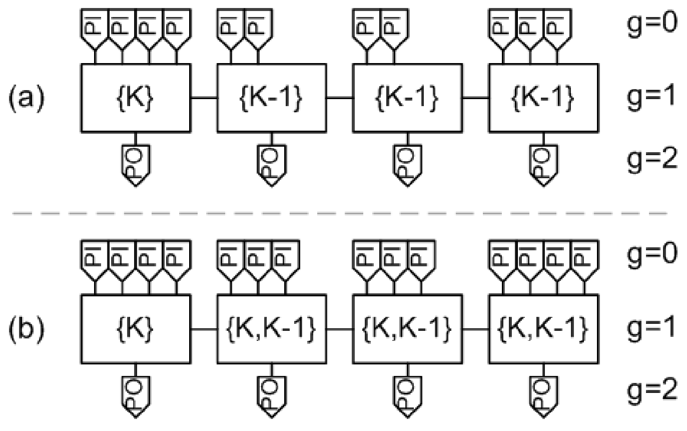


Figure 3.2 (a) *(K-1)* carry-select chain, (b) $\{K-1, K\}$ heterogeneous logic chain

either mode depicted in Figure 3.1, and forms heterogeneous chains as in Figure 3.2(b). Heterogeneous chains are capable of using a combination of full and sub-width LEs. The reuse cell supports heterogeneous chains without the additional interconnection required by the Stratix LUT chain, instead reusing the existing carry chain. Generic logic chains require an LE that can operates in either mode in Figure 3.1, i.e. the Stratix or the reuse LE presented here[36], and are not currently suitable for Xilinx devices or the Stratix II/III.

The basic operation of a carry select adder bitslice, implemented by the *traditional cell* in Figure 3.3, is to pre-compute the carry and sum for both possible carry in values, before it arrives. If the *cin* is 0, the results of the $c_0 = f_0(dataa', datab)$ and $s_0 = f_2(dataa', datab)$ LUTs are passed to their respective outputs. Likewise, if *cin* is 1, the results of the $c_1 = f_1(dataa', datab)$ and $s_1 = f_3(dataa', datab)$ LUTs are passed. However, upon further inspection, the expressions $cout = \overline{cin} \cdot c_0 + cin \cdot c_1$ and $sum = \overline{cin} \cdot s_0 + cin \cdot s_1$ are identical to that of a 3-input LUTs with inputs $\{cin, dataa', datab\}$. This is an efficient way to compute arithmetic functions, but limits the outputs *cout* and *sum* to 3-input functions. For generic logic chains it is desirable that the full computing capacity of the native LUT, in this case 4-inputs, be capable of traversing the *combout* and *cout* outputs simultaneously.

Allowing the full K-LUT value to be output from a logic cell has a few constraints. First, there is a limited number of mask bits, and if all of them are used to compute either the

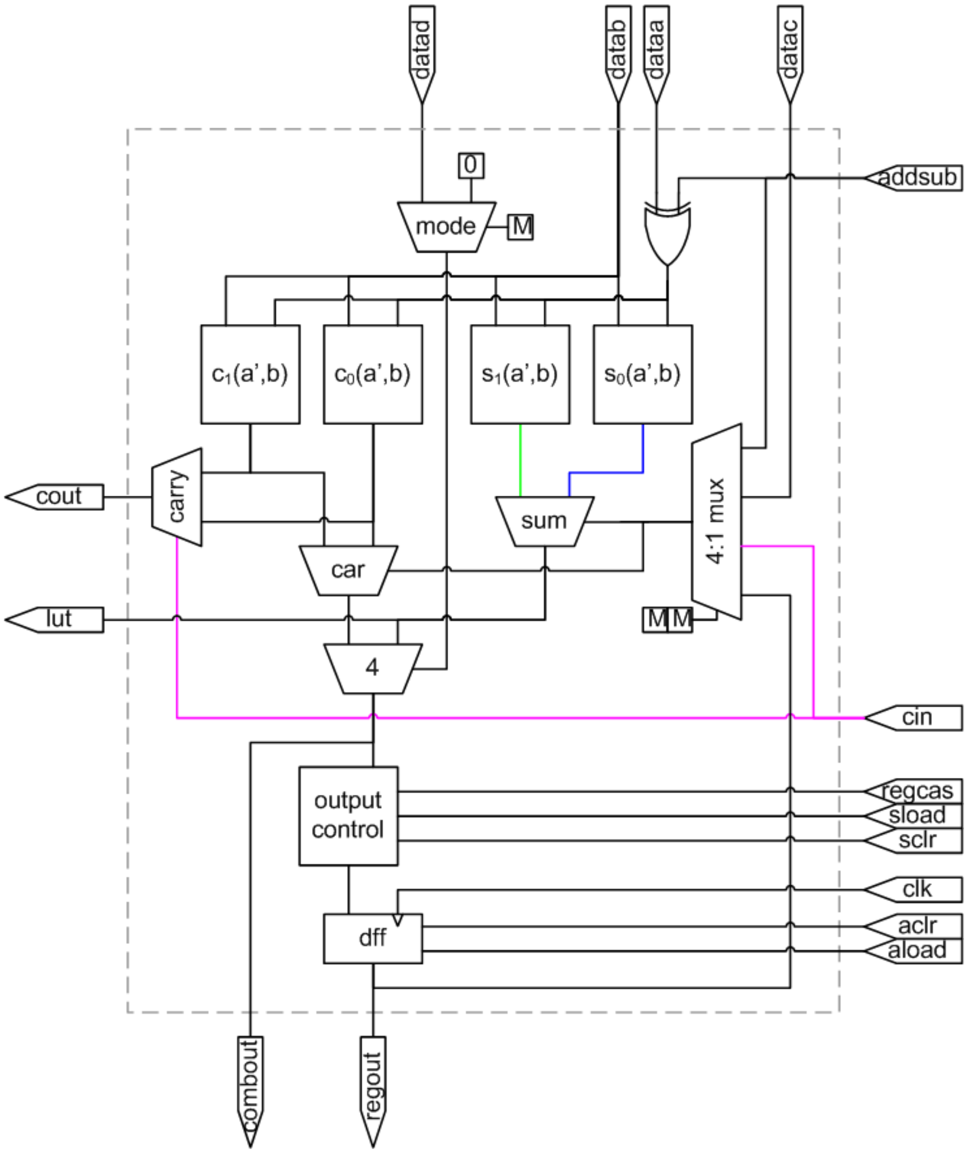


Figure 3.3 Traditional carry-select architecture.

cout or *combout*, both ports must output the same result. Such generic chains are referred to as *full-width logic chains*. This is a deviation from traditional arithmetic chain logic, which computes separate $K-1$ values for each port. Second, for the sake of legacy tool flows and design flexibility, regular arithmetic carry-select capability should be preserved. Third, the impact of simultaneously supporting full and sub-width logic chains should be minimal. This is to say that the architecture modifications and extra logic should not significantly impact logic cell delay and area, relative to that of a traditional cell.

Figure 3.4 depicts a carry-select reuse cell inspired partially by the Stratix logic cell characterized by Figure 3.3. Its main contribution is to enable the full K-LUT function computed by an LE to drive both the routing and chain net outputs simultaneously with the same value. It performs this without extraneous interconnection between adjacent LEs. The reuse cell allows the entire function computed by the 4-LUT structure of the LE to traverse the carry chain. It achieves this through two additional 2:1 multiplexers and a modified LUT mask relative to the traditional cell. The traditional cell in Figure 3.3 will be used as the point of comparison for the architecture discussion. Several modifications have been made to the traditional cell that preserve its functionality and facilitate chain reuse. In the following description, $dataa'$ will be used to refer to the output of the XOR gate with inputs *dataa* and *addsub*, while x will be used to denote an ambiguous component or signal.

The *mode* multiplexer, while depicted as a simple 2:1 multiplexer in Figure 3.4, actually has a dual output capability, as depicted in Figure 3.5. When in arithmetic mode (memory bit is set to 0), the multiplexer passes a static logic value on both outputs. The output destined for multiplexers car_0 and car_1 is pulled to ground by a pulldown resistor, and the output destined for multiplexers sum_0 and sum_1 is pulled high by a pullup resistor. When in normal mode (memory bit is set to 1), the static outputs are overridden, and *datac* is passed to all multiplexers. It has been represented as a 2:1 multiplexer in Figure 3.4 because it is transistor-neutral relative to a traditional 2:1 pass transistor multiplexer and provides similar functionality, but is difficult to symbolize in the schematic. The function of the *mode*

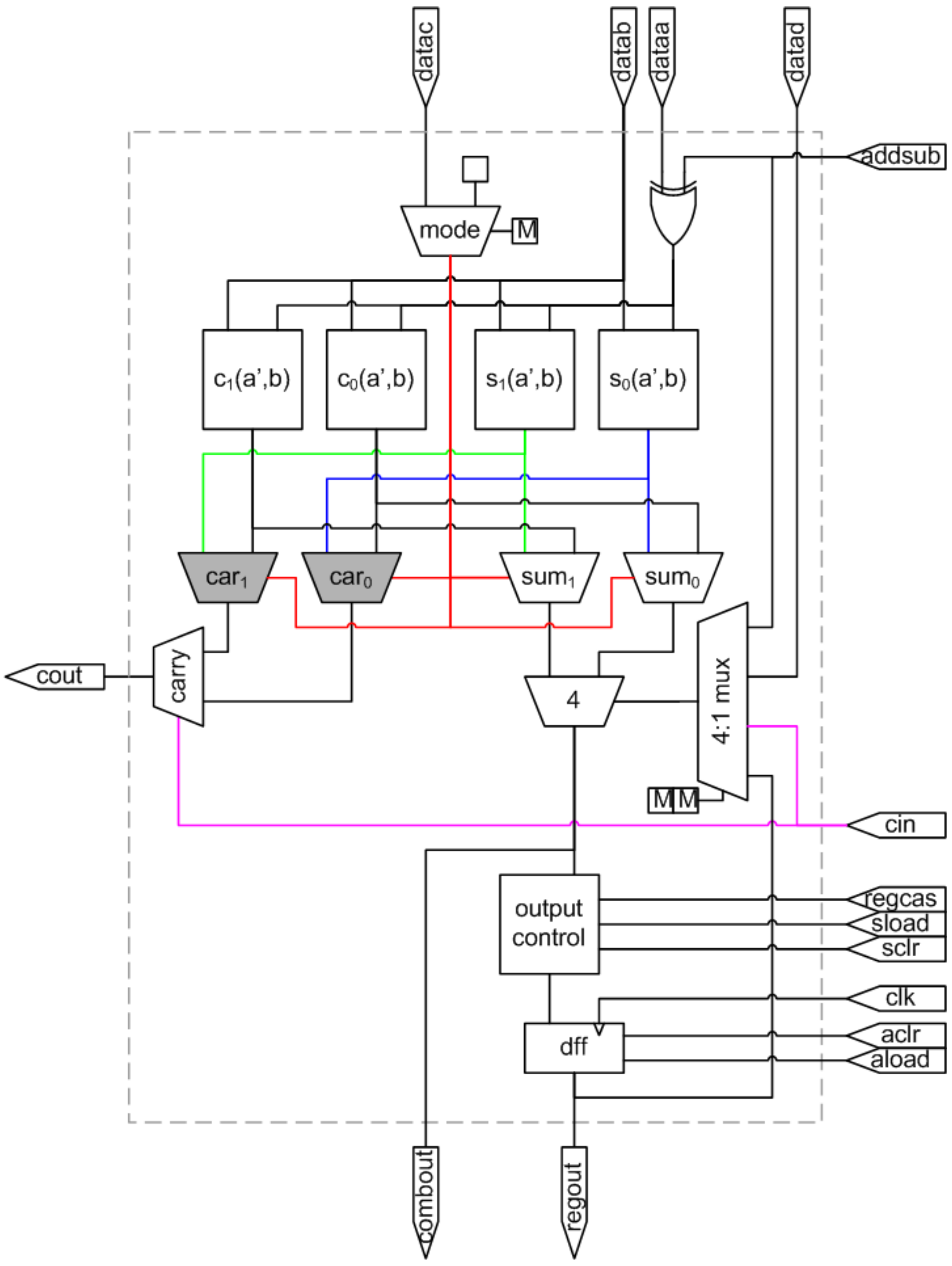


Figure 3.4 Chain reuse carry-select architecture.

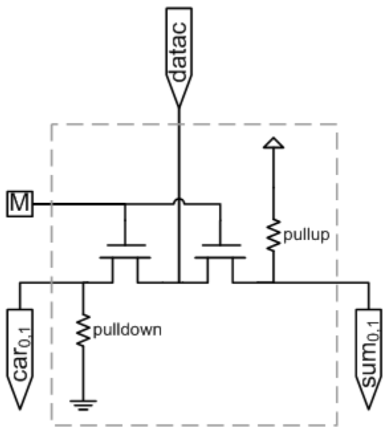


Figure 3.5 Implementation of the *mode* multiplexer.

multiplexer functionality will be discussed shortly, but is necessary for proper normal and arithmetic operation.

In arithmetic mode, the traditional cell computes the sum of $dataa'$ and $datab$ using LUTs s_1 and s_0 and the *sum* multiplexer, whose result is passed through multiplexer 4. The reuse LE, on the other hand, passes s_1 and s_0 through a level of multiplexers and instead relies upon multiplexer 4 to compute the sum. While this is a difference in implementation, it is functionally equivalent when the *mode* multiplexer is set in arithmetic mode. Similarly, the carry computation is performed after allowing the LUT results to pass through multiplexers car_1 and car_0. The carry computation is completed once the carry into the cell is available, as is true in the traditional design.

The additional level of multiplexing in the reuse design institutes a delay on the carry computation once the general inputs, $dataa'$ and $datab$, become available. This, for the most part, only affects the first cell in a carry chain because it is the only one dependent on the arrival of general routing inputs. It is assumed that in an synchronous design all general routing inputs become available roughly at the same time, thus making the delay through the carry chain the critical path of the circuit. All cells in the chain, with the exception of the first, have already computed the carry for both *cin* conditions and propagated those values to the *carry* multiplexer to await the arrival of the *cin*. In short, the sum computation of the reuse cell should expect identical latency to the traditional, while the carry latency should only

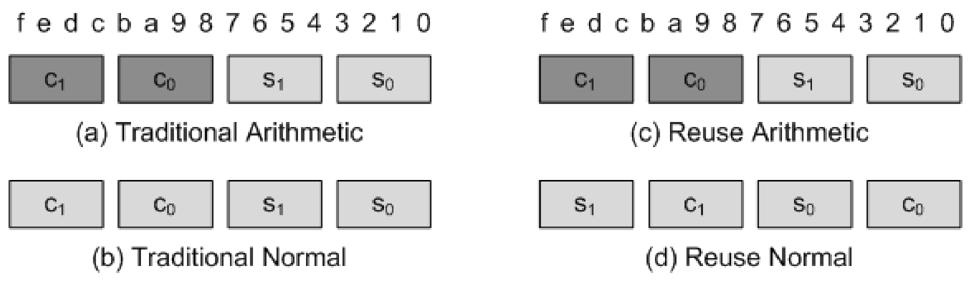


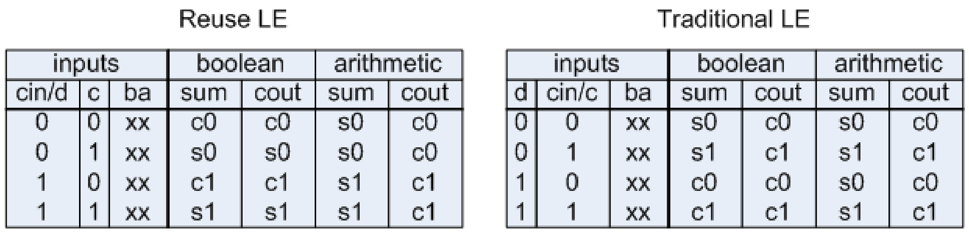
Figure 3.6 Mask modes: (a) traditional arithmetic, (b) normal, and (c) reuse arithmetic, and (d) reuse normal.

differ for the first cell in a chain. The carry function $cout = f(dataa', datab, cin)$ is preserved in arithmetic mode.

The normal (Boolean) operating mode of both cells is appreciably different. The *mode* multiplexer is set to pass a general routing input *datad* to multiplexer 4 in the case of the traditional cell. However, in the reuse cell the *mode* multiplexer passes *datac* to the $car_{0,1}$ and $sum_{0,1}$ multiplexers–a slight difference that allows the carry out of a cell to be $cout = f(dataa', datab, datac, cin)$ for Boolean non-arithmetic chains. The *combout* function is computed similarly in both cell designs, the difference being that the LUT mask is rearranged in the reuse cell. Because of the need to pass the carry and sum LUT outputs through the $car_{0,1}$ and $sum_{0,1}$ multiplexers, respectively, in arithmetic mode without performing a computation the middle nibbles of the 4-LUT mask are interchanged, as shown in Figure 3.6(d). This is a small and innocuous change that is easily dealt through input reordering. Figure 3.7 gives the truth table representations of all LE operating modes.

In summary, no extra delays are introduced in the reuse cell for Boolean operations, but the *cout* can accommodate a $f(dataa', datab, datac, cin)$, while the *combout* can be the same function as the traditional cell of three general inputs and flexible fourth input from the 4 : 1 multiplexer. The function of the 4 : 1 multiplexer is to allow different inputs to the LUT structure including an external data input, *datax*, *cin*, the register feedback of the cell, *regout*, and *addsub*. This functionality is preserved with the only difference between the two cells being the value of *datax* (*datad* for reuse, *datac* for traditional).

Circuit layout and simulation results in Table 3.1 indicate that the *cout* for the initial LE in



Reuse LE

inputs			boolean		arithmetic	
cin/d	c	ba	sum	cout	sum	cout
0	0	xx	c0	c0	s0	c0
0	1	xx	s0	s0	s0	c0
1	0	xx	c1	c1	s1	c1
1	1	xx	s1	s1	s1	c1

Traditional LE

inputs			boolean		arithmetic	
d	cin/c	ba	sum	cout	sum	cout
0	0	xx	s0	c0	s0	c0
0	1	xx	s1	c1	s1	c1
1	0	xx	c0	c0	s0	c0
1	1	xx	c1	c1	s1	c1

Figure 3.7 Mask truth tables for the reuse and normal LEs.

Table 3.1 Layout Summary

Mode	$\mu m x \mu m$	$cout_0$ (ns)	$cout_{n-1,1}$ (ns)	$combout$ (ns)
Re	97.95 x 82.35	6.04	3.50	5.94
Trad	97.95 x 79.95	6.01	3.49	5.90
R:T	1.03	1.01	1.00	1.01

a chain, and the *combout* for all LEs both suffer a 1.01x delay at 3.3v 0.6μm process technology. The assumption of such a recalibration in timing results is that the same layout techniques, when applied to each LE, will be an indication of the expected impact on a commercial LE layout.

Obviously, FPGA vendors use highly optimized LE designs to which the research community is not privy to, and such a design penalty estimation is necessary to fairly compare performance results. A caveat of this work is that the layout results obtained are not necessarily representative of commercial LE implementations. The assumptions are that the traditional and reuse LE designs have been treated equally during layout. In the overall LE layout, area is dominated by the 20 SRAM configuration bits, D Flip-flop, pass transistor 4-LUT structure, and output control. However, these components are common to each LE design. In this context, the two additional 2:1 pass transistor multiplexers of the reuse LE yield a 1.03x area increase. The 1.03x increase would be further amortized over the entire LE area with the inclusion of additional configuration bits and additional LUT infrastructure resulting from an increase in LUT size.

3.2 Summary

This chapter presents a novel LE design for carry chain reuse. The reuse LE is shown to allow a full K-LUT operation to traverse the existing carry chain as well as support traditional $K-1$ operation. These functions are respectively known as full (K) and sub-width $(K-1)$ operation modes. The reuse LE has little impact on delay and area, a property that is further reinforced when increasing LUT sizes are considered. Increasing K-LUT size only serves to increase the number of configuration bits and the size of the LUT computation structure. Increasing the area of all other LE constituents allows the area penalty imposed by the reuse cell, two pass transistor multiplexers, to be amortized over a larger area. Thus, the reuse cell becomes an even more attractive design choice as LE capability increases and a valuable architectural modification for generic logic chains.

CHAPTER 4. CASE STUDY: POST-TECHNOLOGY MAP HEURISTICS

Carry chains in reconfigurable fabrics serve a very important, yet very specific purpose: to facilitate the efficient implementation of arithmetic functions. Carry chains allow arithmetic functions to bypass the performance-costly general routing array. However, if a carry chain isn't used for an arithmetic function, it becomes a superfluous adjacent cell interconnection resource. There are several challenges to carry chain reuse for non-arithmetic chains, some architectural, while others are based on tool support.

The architectural obstacles have resolved by the novel reuse architecture presented in Chapter 3, through the availability of other architectures supporting logic chains, and the ability to use sub-with $(K-1)$ chains available from carry-select arithmetic. Yet, these logic chain structures are useless unless a CAD tool can efficiently implement them. Current software packages identify arithmetic carry chains through high-level HDL macros and primitives. The LUT chain is mapped by Quartus II during PNR according to undisclosed metrics. The only recourse for a designer wanting logic chains is to create them with low level primitives or hand modify the design. The most common academic synthesis tools, SIS [66] and ABC [59] do not support arithmetic chains in their internal representation.

This chapter presents an initial foray into logic chain formation, post-technology map experiments on the formation of chains, and demonstrates how they can benefit an architecture [36]. To do this, a simple probability model of chain formation is created that works without modifying HDL-created chains or a technology mapped design. The only changes made to the netlist are the replacement of eligible general routing nets with high-speed chain connections. The results and discussion will justify the generalization of arithmetic chains to generic logic chains, and guide their definition and mapping in Chapter 5 [38]. As a caveat, the design characteristics in Chapter 4 will differ from Chapter 5 as different design flow tools are used, i.e. Quartus II vs. SIS/T-VPack/VPR.

4.1 Post-Technology Map Experiments

Altera's Quartus II design tool integrates all aspects of the reconfigurable fabric design flow. It performs synthesis, technology mapping, clustering, and PNR using its own set of tools, but through the Quartus University Interface Program [55] allows academic tools to be substituted for any part of the flow. Academic tools can be tested using commercially available hardware and mesh with a commercial CAD flow.

Quartus produces a post-technology mapped design file in Verilog Quartus Module (VQM) format, which can be modified and used as input to the PNR engine. This allows the Stratix architecture to be used to estimate the impact of reuse in "black-box" style, i.e. the LEs are assumed to use the reuse cell presented in Chapter 3. A VQM parser has been developed to input a technology mapped design, identify opportunities for reuse, and make the appropriate modifications to the design. The appropriate modifications are, in accordance to a particular algorithm, to remap a net from the *combout* to the *cout* of the source cell, and to remap the general *datax* input of the sink cell to the *cin*. No other connectivity alterations are incorporated. The result is a design whose logical interconnection and logic cell utilization have not been altered in an effort to ascertain the effect reuse has on application speed and routing resource utilization. The modified VQM file is admittedly not a functioning design, but will serve for experimental purposes. It is not fully functional because the internal cell design is that of the traditional cell, not the reuse cell. Use of a commercial architecture and tools, while having the advantage of providing a reasonable estimate of real-world performance, also carries with it the specialized architectural features and tool optimization techniques that conflict with the experimental cell design. In the case of chain reuse, the following considerations must be made:

1. The cell design is assumed to be the modified cell design presented in Chapter 3.

2. Timing analysis must incorporate the delay introduced by the modified cell design, according to Table 3.1.
3. Stratix LEs feature an LUT chain providing similar connectivity to the reusable carry chain.
4. PNR technology map optimization is disabled by setting the TRUE_WYSIWYG_FLOW option to “ON”.

The first two considerations have been dealt with in Chapter 3 through assessing penalties pursuant to the differences observed between cell layouts. Because the difference in area cannot be accounted for reliably, commercial power estimation of the reuse designs is infeasible. The third item indicates that the LUT chain structure can implement some of the same connections as a non-arithmetic carry chain, and can’t be disabled in the PNR engine. It will be shown that this architectural feature can be omitted when reuse cells are used. Finally, the PNR engine performs additional technology map optimization such as the trimming of inputs unused by the LUT mask. These optimizations have been disabled in both traditional and reuse designs to ensure that the cells are implemented exactly as desired.

Using black-box estimation, reuse opportunities are discovered and exploited. Four separate algorithms will be applied to reuse. A valid parent is a cell whose *cout* port is free, and whose *combout* is in an exclusive relationship with the data input of a child cell, i.e. the child is the only sink of the parent. A valid child is defined as a cell who possesses at least one data input that is in an exclusive relationship with a parent’s *combout* port and whose *cin* port is free. The following definitions are used:

- L - The length of the chain in terms of number of cells.
- S_i - Child cell i where $1 \leq i \leq L-1$
- P_j - Parent cell j where $1 \leq j \leq L-1$
- R - Correlation, i.e. the probability that cell pair S_i and P_j, where $j = i+1$, would be placed adjacent to each other by the PNR engine.
- R_{th} - The minimum threshold correlation that cells S_i and P_j must have to form a chain.

- x - An exponent used to compute R_{th} in Equation 4.5.
- N - The set of nets in the netlist.
- C - The set of cells in the netlist.

A valid chain can be formed between S_i and P_j if $i = j$ and $R \geq R_{th}$. To form chains that aren't exceedingly naive, correlation (R) is established as the likelihood that two cells would be clustered by PNR regardless of carry chain constraints. A net is incident on a cell if one of C_k's I/O ports is connected to N_a. Probability of incidence is defined in Equation 4.1 as the degree of net N_a, i.e. the number of cells it is incident on, divided by the total nets, $|N|$.

$$Pr(N_a\ incident\ on\ C_k) = \frac{deg(N_a)}{|N|} \tag{4.1}$$

The probability that two cells, C_i and C_j, would be clustered together without the influence of the carry chain is assumed to be dependent on the intersection of their nets $N_{C_i} \cap N_{C_j}$, referred to as attraction in [12]. For each cell the probability that each of their sets of nets would occur in the netlist at random are given by Equations 4.3 and 4.4. Thus, the probability that these two cells would be placed together is the product of their probabilities of occurrence, $Pr(N_{C_i})$ and $Pr(N_{C_j})$. This yields the correlation, Equation 4.4, which is the probability that all the nets incident on the pair of adjacent cells, C_i and C_j, would occur in the netlist at random.

$$Pr(N_{C_i}) = \prod_{y=1}^{|N_{C_i}|} Pr(N_y\ incident\ on\ C_i) \tag{4.2}$$

$$Pr(N_{C_j}) = \prod_{z=1}^{|N_{C_j}|} Pr(N_z\ incident\ on\ C_j) \tag{4.3}$$

$$R_{ij} = Pr(N_{C_i}) \cdot Pr(N_{C_j}) \tag{4.4}$$

R_{th} is defined in Equation 4.5 as a small net estimator. Here, s is a constant that is much less than the sum of the net degrees, $s \ll \sum_{a=1}^{|N|} deg(N_a)$ divided by the number of cells $|C|$. This limits the number of cells that have many small degree nets. The degree of a net $deg(N_a)$ divided by $|C|$ yields a density function for net N_a. However, the goal of correlating cells i

and j with R_{ij} is to cluster cells with a high probability that the PNR engine will cluster them. Cells with many small degree nets incident on them, e.g. *datax* inputs, represent a high unlikelihood that this will occur, while nets with higher degree, such as clocks and register enables, will give a greater likelihood. By taking the density function to the power x, e.g. the number of cells in the cluster, the number of small nets incident on a cluster of cells is limited when R_{th} is used as a minimum bound on R_{ij}. Because cells C_i and C_j have a total of 8 potentially small fanout nets incident on them (the total number of *datax* inputs), x is on the range $7 \leq x \leq 10$.

$$R_{th} = \left(\frac{s}{|C|}\right)^x \tag{4.5}$$

Four algorithms have been designed to test chain reuse as a component of the design flow. Each iteratively selects cells that are valid children but not valid parents and extends the chain from output to input until a stop condition is met. Figure 4.1 shows all possible chains formed by the LONG, SHORT, and SHORTm algorithms. The chains attributed to the THRESHx algorithms are just some of the possible chains formed. The algorithms perform as follows:

- LONG - Stops once a child is found not to have a parent. If S_i has multiple parents, the one with the best correlation to S_i is chosen, and extension continues.
- SHORT - Stops once a child cell is found not to have a parent or has multiple parents.
- SHORTm - It may happen in SHORT that S_1 has multiple parents, thus fulfilling the stop condition and yielding no chain ($L = 1$). In this situation SHORTm chooses the parent with the highest correlation and then stops, yielding a minimum chain ($L = 2$).
- THRESHx - A depth first search tree is formed rooted at S_1. All child/parent pairs are given a correlation score R. Only pairs whose $R \geq R_{th}$ form chains.

Each design has had its VQM technology map generated by Quartus II and had reuse applied using an algorithm. For the THRESHx algorithm, $7 \leq x \leq 10$ pursuant to Equation 4.5. The resultant designs are input to the Quartus PNR engine and timing analyzer. Timing has been recalibrated to account for the delay introduced by the modified cell design (Table 3.1).

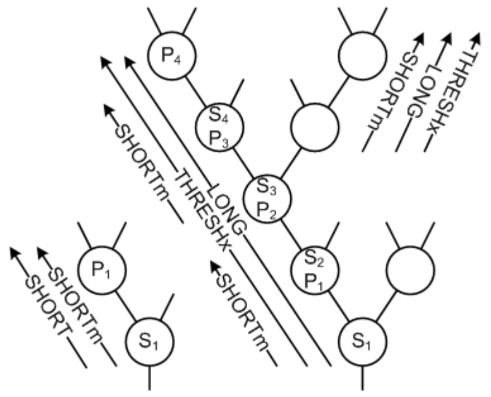


Figure 4.1 DFS tree from output to input.

Table 4.1 Reuse Summary

	Balanced Speed-up and Utilization Ratio Results using Quartus II										
Design	LEs	Nets	Algo	Chains	$\sum_i L_i$	Ave L	% A	% R	% A+R	SU	U
aes128_fast	9823	10197	THRESH9	2632	5917	2.24	0.0	60.2	60.2	1.14	0.96
cfft	3044	6389	SHORT	20	42	2.1	57.6	1.4	59.0	1.03	0.95
des3	16400	16645	LONG	1921	4130	2.15	0.1	25.2	25.3	1.47	0.91
dlx	16268	21306	SHORT	2722	7245	2.66	9.2	44.5	53.7	1.34	0.99
eth_top	375	498	THRESH7	8	16	2.0	9.1	4.3	13.4	1.07	0.92
jpeg_encoder	4726	11569	THRESH7	4	8	2.0	36.3	0.17	36.4	1.04	0.99
RSACypher	1073	1662	THRESH7	5	10	2.00	25.0	0.9	25.9	1.02	0.89
sha512	4777	8219	THRESH9	684	1464	2.14	24.4	30.6	55.0	1.13	0.92
usb	3234	3659	SHORT	377	868	2.30	8.8	26.8	35.6	1.11	0.96

Reuse effectiveness is judged using the maximum clock frequency and routing utilization. The ratio of the reuse design to the traditional design is presented, where values greater than 1 are desirable for speedup (SU) and values less than 1 for routing utilization (U). Table 4.1 shows the algorithm achieving the best simultaneous delay and utilization improvement. It presents the total LEs and nets, best performing algorithm, total chains, total cells in chains ($\sum_i L_i$), average chain length, percent of arithmetic cells (%A), percent of reuse cells (%R), the percent of cells in chains (%A+%R), and the SU and U relative to normal design flow. For almost all cases, at least one algorithm improves both SU and U, and in all cases an algorithm improves either SU or U, although sometimes at the cost of the other.

Figure 4.2 presents the speedup achieved by reuse for all designs and algorithms. Figure 4.2

and Table 4.1 indicate that speedup is highly dependent on the algorithm and application combination, with performance between 1.02-1.47x for at least one algorithm per design. LONG tends to create longer chains which harm smaller or more heavily arithmetic-dominated designs. However, it exhibits the highest speedup for any algorithm/design combination at 1.47x for DES3 due to prevalence of opportunity and ability to create longer chains. The SHORT and SHORTm algorithms produce mixed results, but also tend to create a large number of chains, thus performing better on bigger designs. Curiously, the average chain length is always $2 < \overline{L} < 3$. The reason is that the best performing algorithm is the one that uses the most chains on the critical path, but also the one that allows the PNR engine the most options. Allowing the PNR to choose where it would like cells to be placed ultimately reduces the routing complexity and the average delay of a routing connection.

The THRESHx algorithms selectively choose pairs based upon the likelihood that such cells would be clustered together by the PNR engine. For this reason, the smaller designs tend to benefit from higher threshold values ($x = \{7, 8\}$), mitigating the number of chains created and consequently the number of constraints on PNR. This general trend is seen throughout the results–for larger, more random-logic designs (DES3, DLX), the ability to increase performance is highly dependent on the ability to replace routing on the critical path with carry chains, or to change the critical path. Conversely, for smaller or more arithmetic designs (RSA, CFFT), the ability to increase performance is highly contingent on decreasing the overall routing utilization of the design (to be discussed in Figure 4.4).

To ascertain the effect carry chain reuse has on overall interconnection array utilization, the resources of the Stratix architecture must be described. Each LAB is a set of 10 LEs featuring 30 general local routing interconnect lines which service intra-LAB routing between LEs and provide for signals to be sourced/sank to/from the global routing array. Every LE is connected to the downstream LE on the carry chain, the register cascade, and LUT chains (the register cascade is shifting structure, not subject to the discussion of routing utilization). The global routing array provides connectivity between LABs in column spans of 1, 4, 8, and 16 LABs, and row spans of 1, 4, 8, and 24 LABs.

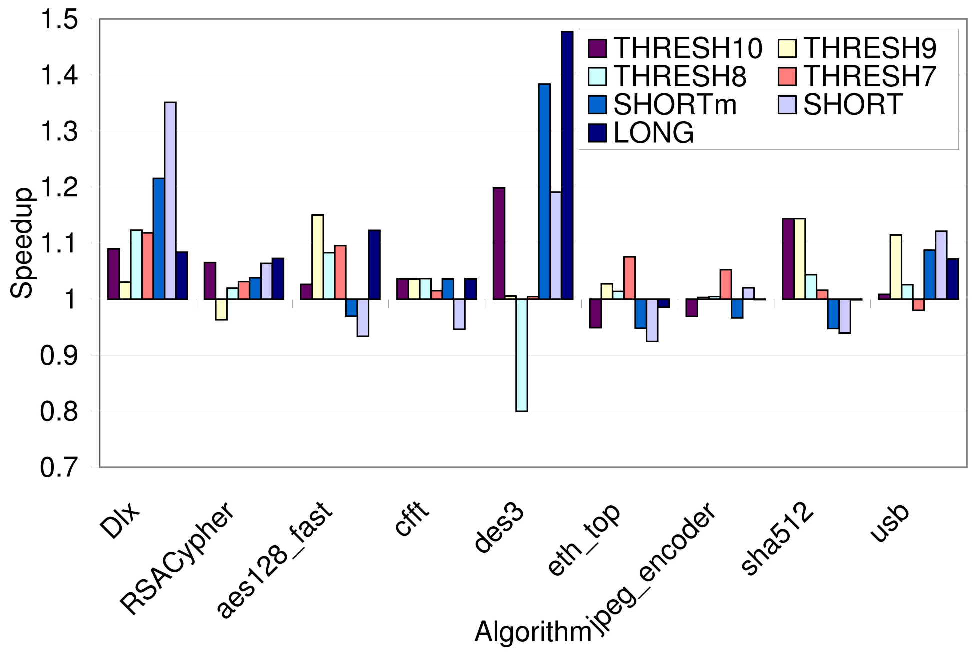


Figure 4.2 Speed-up of Reuse over Unmodified Flow vs. Algorithm

Figure 4.3 shows how reuse effects each individual routing resource (in order: c16, c4, c8, direct, lut chain, local, r24, r4, r8), where a ratio greater than 1 signifies a reuse design consuming more resources than its corresponding normal design. Both local routing resources (intra-LAB local routing and LUT chains) witness substantial average decreases on the order of 30-60%. For LUT chains, this means that they are potentially superflous to the architecture, as their functionality is essentially replaced by the carry chain. Additionally, local intra-LAB routing provisioning could possibly be decreased from 30 general purpose wires to 20-25 due to connection migration to the carry chain. However, the effective clustering of cells along the carry chain has come with the effect of increasing some of the other routing compoments. For the LONG and SHORT algorithms, global routing resources are increased quite often. However, when THRESHx is applied routing resource decreases can be seen for at least one threshold/design pair given proper selection of x because chained cells are in higher accordance with PNR decisions.

A simple cost metric has been developed to measure the cumulative effect on interconnection utilization. The columnar and local routing structures run parallel to the carry chain

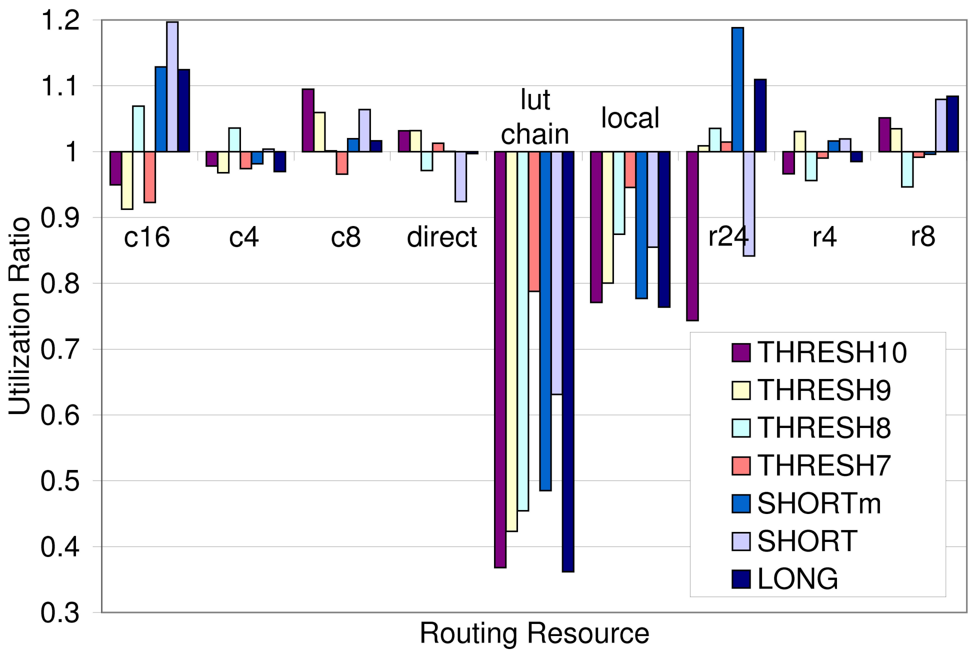


Figure 4.3 Average Ratio of Reuse to Unmodified Utilization vs. Interconnect Resource

and row structures run orthogonal to the chain as well as to the layout of LEs. Equation 4.9 gives the total routing cost as the summation of the weighted constituent routing structures. The length of an LE is used as 1 unit of wire length, and accordingly each column structure spanning 4 rows (C_4) is assessed a cost of 40 units. However, using specifications and Quartus tools, each row routing structure is deemed to be approximately half the size of its column counterpart because it does not have to span the entire length of a LAB. Thus, each 8-column row structure (R_8) spans 40 units, and so forth. LUT chains, because they are adjacent LE structures, are given a weight of half a unit. Local intra-LAB wires are given a cost of 10 units, as are direct connections between LABs (as they still must cross LABs and other wires in both vertical and horizontal directions). The goal of this metric is not to express delay, but to characterize the interconnection utilization of a given design.

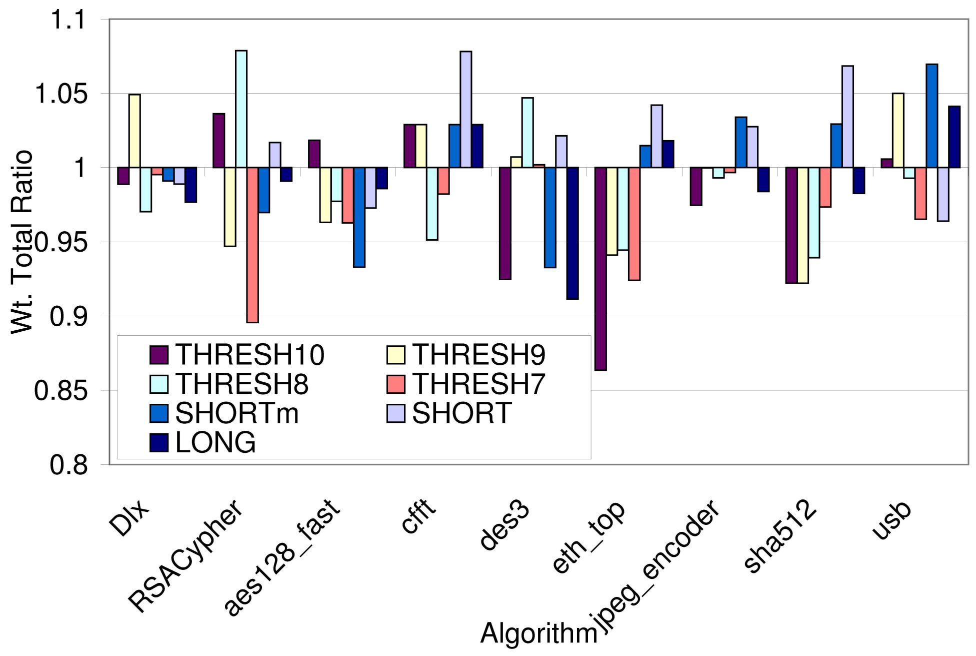


Figure 4.4 Ratio of Reuse Utilization to Unmodified Utilization vs. Algorithm

$$U_{col} = 160 \cdot C_{16} + 80 \cdot C_8 + 40 \cdot C_4 + 10 \cdot C_1 \tag{4.6}$$

$$U_{row} = 80 \cdot R_{24} + 40 \cdot R_8 + 20 \cdot R_4 + 10 \cdot R_1 \tag{4.7}$$

$$U_{lcl} = 0.5 \cdot LUT_{chain} + 10 \cdot Local \tag{4.8}$$

$$U_{tot} = U_{col} + U_{row} + U_{lcl} \tag{4.9}$$

Figure 4.4 shows the weighted interconnection utilization ratios per algorithm. As expected, the the overall change in utilization is dictated by global routing changes. In smaller designs, more constraints on the PNR engine lead to higher routing utilization. The THRESHx suite of algorithms limit the number of reuse chains that are formed by only allowing only those chains that are likely to be agreed upon by the router. Routing savings of up to 13% are observed by selecting the proper threshold with each design exhibiting utilization savings for at least one algorithm.

4.2 Summary

The post-technology map experiments with naive algorithms show that arithmetic carry chain reuse in FPGAs can offer benefits to non-arithmetic operations. Speedup and routing utilization results indicate that each design can benefit from the application of at least one algorithm. A design/algorithm pair can be found that can potentially increase maximum clock frequency, with an observed maximum of 1.47x, and decrease routing consumption, with an observed minimum of 0.87x, for the presented designs.

While some algorithms often simultaneously increase performance and decrease routing utilization, others may do quite the opposite. The selection of the appropriate algorithm depends on the characteristics exhibited by the design, such as number of arithmetic chain cells, number of cells in the design, the average fanout of each net, as well as the desired performance of the designer. One of the most important findings of the experiment is that forming every possible chain is not necessarily a good policy. Larger, less arithmetic designs (DES3) may be able to accommodate a large increase in the number of chains, but smaller, more arithmetic designs (RSA) have already reached "saturation."

One reason for performance decrease is that long chains restrict the PNR engine. When the average chain length is small, $2 < \overline{L} < 3$, performance increases most because it allows the PNR engine the most options. Allowing the PNR to choose where it would like cells to be placed ultimately reduces the routing complexity and the average delay of a routing connection. The post-technology map experiments with naive algorithms show chain reuse has great potential, however valuable information is lost during technology mapping that could lead to even more efficiency. The observations of this case study will guide the creation of the optimal mapping algorithm of Chapter 5.

CHAPTER 5. OPTIMAL LOGIC CHAIN TECHNOLOGY MAPPING

As Chapter 4 discovered, non-arithmetic chains have the potential to greatly affect the performance of any given design. However, their assignment must be done with care, because it has also been shown that overuse of chains can decrease performance. In short, not every chain is a good chain. Additionally, opportunities for chain reuse are limited in designs already possessing high numbers of arithmetic chains. Of the suite of techniques outlined in Chapter 4, none consistently yields favorable solutions, an indication that a more exact solution is needed to fully harness the potential of chains. It is especially important chains be addressed during technology mapping, instead of as an afterthought, otherwise valuable opportunities are obfuscated by the packing of Boolean nodes into K-LUTs.

Finding logic chains in a design is contingent on a definition of a chain that encompasses both arithmetic and non-arithmetic logic. A generic logic chain can be defined as a set of consecutive nodes, such that each increases the logic depth of the design without increasing its general routing depth. The delay of a chain net will be considered $0ps$, due to the emphasis placed on chain efficiency during architecture design and layout. Chains have the constraint that a LE chain output is limited to a single LE fanout (although it is a dual fanout between LUTs), and that each LE is limited to a single chain input. With the advent of the architecture described in Chapter 3, no distinction will be made between arithmetic and non-arithmetic chains henceforth. Optimal logic chain creation occurs when the minimum general routing depth of a design is achieved. Consequently, logic depth is traded for routing depth, and HDL macros are disregarded during mapping.

5.1 Problem Formulation and Definitions

The optimal routing depth technology map solution described by ChainMap is partially based on the optimal logic depth FlowMap [20], and is formulated similarly for ease of comparison. SIS [66] nomenclature is used to describe an arbitrary Boolean network. Such a network can be represented as a directed acyclic graph (DAG) $N = (V, E)$ with vertices V and edges E, where $n = |V|$ and $m = |E|$. Each Boolean gate in the network is represented as a node, and $edge(u, v)$ connects nodes $u, v \in V$ if there exists a net from the output of gate u to an input of gate v. Notation is abused such that $u \in N$ implies that $u \in V$ and $edge(u, v) \in N$ implies $edge(u, v) \in E$ for $N = (V, E)$. A predecessor is defined as a node u such that there exists a directed path from u to v for $u, v \in N$. Likewise, a descendant is a node v such that there exists a directed path from u to v for $u, v \in N$. PIs have no incoming edges and POs have none outgoing. The following definitions will be used in the description of ChainMap:

- u, v, w, x are general nodes in a graph
- $PI(N)$ and $PO(N)$ refer to the set of primary inputs or outputs of N, respectively
- i, j are scalar indices used with nodes
- s is an auxiliary global source node, s.t. $\forall v \in PI(N)$, $edge(s, v)$ is added
- t denotes a sink node, and N_t is a subgraph of N containing node t and its predecessor nodes and edges
- s denotes a source node, and N_s is a subgraph of N containing node s and its descendant nodes and edges
- d is a depth increasing node
- $g(v)$ is the routing label and $l(v)$ the logic label for v
- p is a scalar s.t. $p = max\{g(u) : u \in N\}$
- q is a scalar s.t. $q = max\{l(u) : u \in N\}$
- $P \subseteq N_t$ s.t. $v \in P$ if $g(v) = p, \forall v \in N_t$
- $P_d \subseteq P$ consisting of d and its predecessors in P
- N_t' is a DAG with a valid depth increasing node

- N''_t is derived from N'_t to apply Max-flow Min-cut
- $d' \in N'_t$ is formed by collapsing the nodes in P_d into d
- $t' \in N'_t$ is formed by collapsing the nodes in $\overline{P_d}$ into t
- $(X, \overline{X}), (Y, \overline{Y}), (Z, \overline{Z})$ denote node cuts in a network, e.g. nodes are partitioned so that $s \in X$ and $t \in \overline{X}$
- $input(H)$ for a set $H \subseteq N$, is the set of $\{u : \forall u \notin H, v \in H, \exists edge(u, v)\}$, and is also abused for nodes
- $output(H)$ for a set $H \subseteq N$, is the set of $\{u : \forall u \in H, v \notin H, \exists edge(u, v)\}$, and is also abused for nodes
- $cap(u, v)$ denotes the flow capacity of $edge(u, v)$
- $LUT(t)$ is the set of nodes in the K-LUT of t

Through abuse of notation, a node or set denoted as "prime" indicates to which network it belongs. For example, $(X', \overline{X'})$ is a cut belonging to network N'_t. A *K-feasible cone* N_v is a subgraph of N containing v and each of its predecessors such that $input(N_v) \leq K$. The goal is to cover *K-bounded* N, where $\forall_{v \in V} |input(v)| \leq K$, with K-feasible cones for implementation in a K-LUT FPGA.

The *level* of t is the longest path from any PI predecessor of $\{u : u \in PI(N_t), u \neq t\}$ to t, with PIs possessing a level of 0. The distinction that ChainMap makes from FlowMap is that level is in terms of the maximum number of routing connections traversed from $PI(N_t)$ to t. Chain connections do not count as a routing level increase, therefore, if the longest path between a PI and node t traverses g general routing connections and c chain connections, $level(t) = g$. The *depth* of the network is the maximum level of all its vertices.

As in FlowMap, the concept of a network cut, $(X, \overline{X})$, is pivotal. The *node cut size*, given by Equation 5.1, quantifies the size of $input(\overline{X})$, i.e. the number of nodes that have a forward edge crossing the cut. To find the K-feasible node cut, the *edge cut size* will be employed, according to Equation 5.2. For the remainder of the algorithm discussion a unit delay model is incorporated, meaning that $cap(u, v) = 1, \forall u, v \in V$. The *logic height* of the cut is the maximum node label in X, as in Equation 5.3. The *routing height* of the cut is the maximum node label in X, as in Equation 5.4.

$$n(X, \overline{X}) = |\{u : edge(u, v) \in N, u \in X\ v \in \overline{X}\}| \tag{5.1}$$

$$e(X, \overline{X}) = \sum_{u \in X, v \in \overline{X}} cap(u, v) \tag{5.2}$$

$$h_L(X, \overline{X}) = max\{l(u) : u \in X\} \tag{5.3}$$

$$h_G(X, \overline{X}) = max\{g(u) : u \in X\} \tag{5.4}$$

The primary objective is to minimize the network routing delay by minimizing $h_G(X, \overline{X})$ for all nodes. Using a binary depth model, each routing net increases routing depth by 1, but it is not increased by any chain net. The secondary objective is to minimize the logic delay of the network by minimizing $h_L(X, \overline{X})$ for all nodes such that $h_G(X, \overline{X})$ is minimum, because network delay is also defined by the delay through its K-LUTs. A third objective is to minimize the area of the design in terms of the number of K-LUTs required by the solution. A solution is optimal if the network routing depth is minimum and the logic depth, within the confines of minimum routing depth, is also minimum.

ChainMap consists of three phases: labeling, mapping, and duplication, with an optional fourth, relaxation. In the labeling phase, ChainMap identifies whether or not a DAG can be constructed that consists of a given node t and its predecessors, and contains a depth increasing node d. If such a DAG is possible, two subsequent graph transformations are applied that isolate d in N'_t and convert the network to N''_t, one to which Max-flow Min-cut can be applied. If a K-feasible cut can be found, then t does not increase the routing depth of the design. If $t = d$, this is akin to the minimum height logic cut identified by FlowMap, and contains all other possible cuts. The second phase of ChainMap is identical to that of FlowMap, wherein the K-feasible cuts computed during labeling are used to form K-LUTs. The third phase

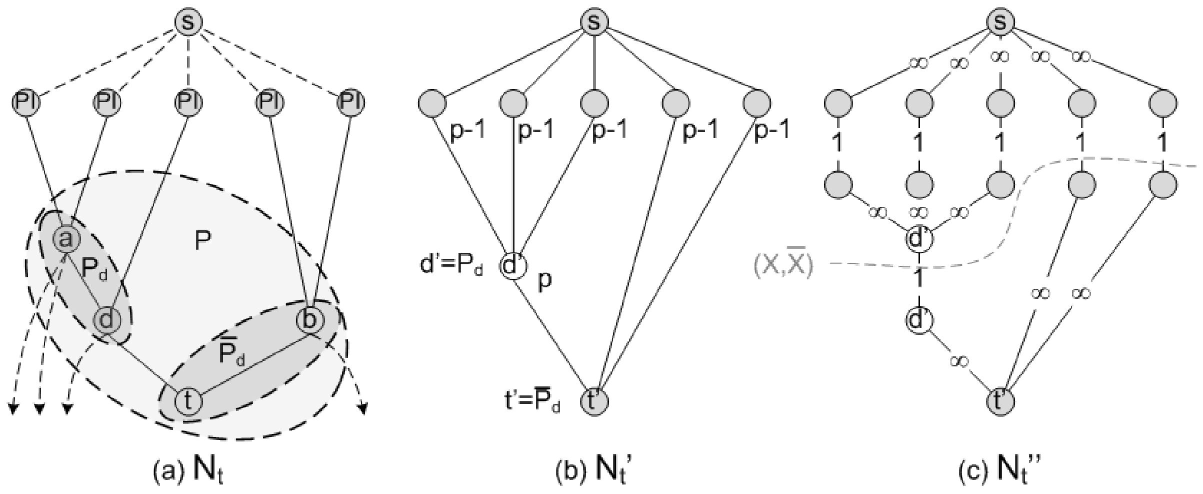


Figure 5.1 Transformation from Boolean network N_t to DAGs N_t' and N_t'' for chain cut.

of ChainMap is to duplicate nodes that source multiple chain nets to adhere to the special constraints imposed by chains. An optional relaxation phase can be applied to restrict the number of duplications required.

5.2 ChainMap Labeling

ChainMap correlates $g(v)$ to the *general routing depth* of node v. This is a subtle change in definition from FlowMap, which uses $l(v)$ to indicate both logic and routing depth because it considers all nets to be routing connections. The introduction of the logic chain provides for a net with properties different from general routing. A chain net allows any $u \in input(v)$ to cause $l(v) = l(u) + 1$ while allowing for the possibility that $g(v) = g(u)$.

The labeling phase is performed on a topological ordering of the nodes in N, ensuring that node $u \in input(v)$ is processed before v. N is K-bounded, meaning $input(u) \leq K, \forall u \in N$. Each $u \in PI(N)$ has $g(u) = l(u) = 0$. Figure 5.1(a) shows an example N_t where all edges traversing to $u \notin N_t$ have been pared away, and the auxiliary source s added.

If $LUT(t)$ denotes the set of nodes in the K-LUT which implements t, then $\overline{X} = LUT(t)$ and $X = N_t - LUT(t)$. Given X and $\overline{X}$, a K-feasible cut $(X, \overline{X})$ is formed such that $s \in X$

and $t \in \overline{X}$ and $n(X, \overline{X}) \leq K$. A *depth increasing node* is one which is solely responsible for increasing the routing depth of $LUT(t)$.

Definition 5.2.1. *Let node $d \in input(\overline{X})$ be a node with maximum label $g(d) = p$. If $g(d) > g(v), \forall v \in input(\overline{X}), v \neq d$, then d is depth increasing.*

Let $u \in X$ be a node with $p = g(u)$ and d be a depth increasing node, then the routing label of t is $g(t) = p$ if $d \in X$ and $g(t) = g(u) + 1$ otherwise. Equation 5.4 indicates that to minimize the $h_G(X, \overline{X})$ of $LUT(t)$, the minimum height K-feasible cut $(X, \overline{X})$ must be found in N_t.

Lemma 5.2.2. *The minimum routing depth solution of N_t is given by:*

$$g(t) = \min_{K-feasible\ (X,\overline{X})} h_G(X, \overline{X}) + \begin{cases} 0 & if\ d \in X \\ 1 & otherwise \end{cases}$$

Let $v \in X$ be the maximum logic label $q = l(v)$, then $l(t) = l(v) + 1$. The logic label of t is dependent on the K-feasible minimum height routing cut $(X, \overline{X})$. Because the nodes in X and $\overline{X}$ represent nodes in different LUTs, logic depth simply increases at each routing cut.

Lemma 5.2.3. *The logic depth of N_t is given by:*

$$l(t) = h_L(X, \overline{X}) + 1$$

Furthermore, for any t, $g(t) \geq g(u)$ and $l(t) \geq l(u)$, $\forall u \in input(t)$. This is important because the value $g(t)$ has two possibilities: if a minimum height cut can be found at $h_G(X, \overline{X}) = p - 1$ or $h_G(X, \overline{X}) = p, d \in X$ then $g(t) = p$, otherwise $g(t) = p + 1$. Likewise, the logic label of t follows a similar derivation and its proof is identical to that presented by Lemma 2 in FlowMap [20]. For purposes of discussion, this proof is excerpted as Lemma 5.2.5. Lemmas 5.2.4 and 5.2.5 ensure that the routing and logic labels of each node are greater than or equal to any of their predecessors.

Lemma 5.2.4. *If p is the maximum routing label of the nodes in $input(t)$, then $g(t) = p$ or $g(t) = p + 1$.*

Proof. If $u \in input(t)$, then any cut $(X, \overline{X}) \in N_t$ results in either $u \in X$ or $u \in \overline{X}$.

When $u \in X$, Equation 5.4 requires that $h_G(X, \overline{X}) \geq g(u)$ and by Lemma 5.2.2 $g(t) \geq h_G(X, \overline{X})$, therefore, $g(t) \geq g(u)$.

When $u \in \overline{X}$, the K-feasible cut $(X, \overline{X})$ defines a K-feasible cut $(Y, \overline{Y})$ in N_u, where $Y = X \cap N_u$ and $\overline{Y} = \overline{X} \cap N_u$. Let $(Z, \overline{Z})$ be the minimum height K-feasible cut computed for N_u. Since $(Z, \overline{Z})$ is the minimum height cut, then $h_G(Y, \overline{Y}) \geq h_G(Z, \overline{Z})$ because $Z \subseteq Y$. Likewise, since $Y \subseteq X$, $h_G(X, \overline{X}) \geq h_G(Y, \overline{Y})$, therefore, $h_G(X, \overline{X}) \geq h_G(Z, \overline{Z})$. There are two possible values for both $g(t)$ and $g(u)$ according to Lemma 5.2.2, resulting in four possible cases. Figure 5.2(a) applies to i and ii, while (b) applies to iii and iv.

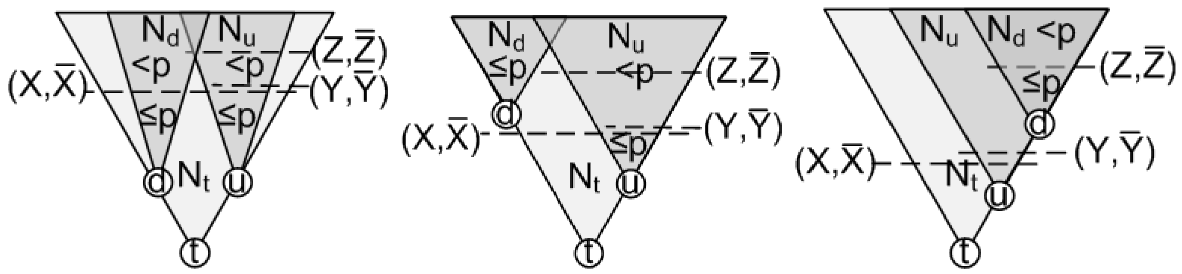
(i) If $g(t) = h_G(X, \overline{X}) + 1$, $g(u) = h_G(Z, \overline{Z})$, then $g(t) > h_G(X, \overline{X}) \geq h_G(Z, \overline{Z}) = g(u)$, thus $g(t) > g(u)$.

(ii) If $g(t) = h_G(X, \overline{X}) + 1$, $g(u) = h_G(Z, \overline{Z}) + 1$, then $g(t) - 1 = h_G(X, \overline{X}) \geq h_G(Z, \overline{Z}) = g(u) - 1$, thus $g(t) \geq g(u)$.

(iii) If $g(t) = h_G(X, \overline{X})$, $g(u) = h_G(Z, \overline{Z})$,then $g(t) = h_G(X, \overline{X}) \geq h_G(Z, \overline{Z}) = g(u)$, thus $g(t) \geq g(u)$.

(iv) If $g(t) = h_G(X, \overline{X})$, $g(u) = h_G(Z, \overline{Z}) + 1$, then $d \in X$. By Definition 5.2.1, $g(d) > g(v), \forall v \in input(\overline{X}), v \neq d$. If $d \notin Y$ then all of Y is less than $g(d)$, and $g(t) = h_G(X, \overline{X}) = g(d) > h_G(Y, \overline{Y}) \geq h_G(Z, \overline{Z}) = g(u) - 1$, thus $g(t) \geq g(u)$. If $d \in Y$, Figure 5.2(c), then $g(t) = h_G(X, \overline{X}) = h_G(Y, \overline{Y}) = g(d)$. Because d is a depth increasing node of t, and $input(\overline{Y}) \subseteq input(\overline{X})$ then d is also a depth increasing node of u, but it is known that $g(u) = h_G(Z, \overline{Z}) + 1$, which by Lemma 5.2.2 indicates $d \notin Z$, implying $d \in \overline{Z}$. Since $d \in \overline{Z}$, then $h_G(Z, \overline{Z}) = g(d) - 1$. Therefore, $g(t) = g(d) = h_G(Z, \overline{Z}) + 1 = g(u)$, thus $g(t) = g(u)$.

A valid alternative K-feasible cut is when $(N_t - \{t\}, \{t\})$ because N is K-bounded. In this situation, any node $u \in N_t - \{t\}$ is either $u \in input(t)$ or a predecessor of those nodes, such that $u \in N_t - input(t) - \{t\}$. Therefore, the maximum routing label, $g(u) = p$, where $u \in N_t - \{t\}$, and $h_G(N_t - \{t\}, \{t\}) = p$, resulting in $g(t) \leq p + 1$. Items i-iv prove $g(t) \geq g(u), \forall u \in input(t)$, thus $p \leq g(t) \leq p + 1$. □



(a) $d \notin N_u, g(t) = h_G(X, \overline{X}) + 1$ (b) $d \notin N_u, g(t) = h_G(X, \overline{X})$ (c) $d \in N_u, g(t) = h_G(X, \overline{X})$

Figure 5.2 Conceptual network cuts.

Lemma 5.2.5. *If q is the maximum logic label of the nodes in $input(t)$, then $l(t) = q$ or $l(t) = q + 1$.*

Proof. If $u \in input(t)$, then any cut $(X, \overline{X}) \in N_t$ results in either $u \in X$ or $u \in \overline{X}$.

When $u \in X$, Equation 5.3 requires that $h_L(X, \overline{X}) \geq l(u)$ and by Lemma 5.2.3 $l(t) \geq h_L(X, \overline{X})$, therefore, $l(t) \geq l(u)$.

When $u \in \overline{X}$, $(X, \overline{X})$ defines a cut $(Y, \overline{Y})$ in N_u, where $Y = X \cap N_u$ and $\overline{Y} = \overline{X} \cap N_u$. Therefore, $h_L(X, \overline{X}) \geq h_L(Y, \overline{Y})$ because $Y \in X$ indicating that $l(u) \leq h_L(Y, \overline{Y}) \leq h_L(X, \overline{X}) \leq l(t)$. Therefore all predecessors of $u \in N_t - \{t\}$ are $l(u) \leq l(t)$. This implies that $l(u) \leq l(t), \forall u \in input(t)$, resulting in $l(t) \geq q$.

A valid alternative K-feasible cut is $(N_t - \{t\}, \{t\})$ because N is K-bounded. In this situation, any $u \in N_t - \{t\}$ is either $u \in input(t)$ or a predecessor of those nodes, such that $u \in N_t - input(t) - \{t\}$. Therefore, the maximum logic label, $l(u) = q$, where $u \in N_t - \{t\}$, and $h_L(N_t - \{t\}, \{t\}) = q$, resulting in $l(t) \leq q + 1$. Therefore, $q \leq l(t) \leq q + 1$. □

Lemma 5.2.4 dictates minimum routing depth is achieved if $g(t) = p$, either by a depth increasing node d, or by $g(u) = p - 1, \forall u \in N_t - LUT(t)$. Each $v \in N_t$ for which $g(v) = p$ or $v = t$ belongs to set P and is an eligible depth increasing node. To see if any $d \in P$ is depth increasing, P must be partitioned into P_d and $\overline{P_d}$, as in Figure 5.1(a). For any $d \in P$, a depth first search (DFS), toward PIs rooted at d and in P, yields P_d and $\overline{P_d} = P - P_d$. Figure 5.1(a) shows $P_d = \{d, a\}$, which constitutes a logic chain at level p, and $\overline{P_d} = \{t, b\}$, constituting $LUT(t)$. If $\overline{P_d} \neq \emptyset$, $t \in \overline{P_d}$ and consists of nodes potentially included in $LUT(t)$, and its contents collapsed into t to form t'. If $d = t$, $\overline{P_d} = \emptyset$ indicating that $LUT(t)$ includes all of the nodes in P (as $P = P_d$), and the contents of P are collapsed into t to form t'.

Lemma 5.2.6. *Let set P contain $\{v : v \in N_t, g(v) = p\} \cup \{t\}$. For $d \in P$, let P_d be the DFS tree rooted at d and in P, and $\overline{P_d} = P - P_d$. N'_t contains a depth increasing node d if there exists no $edge(u, v)$, where $u \in P_d - \{d\}$ and $v \in \overline{P_d}$.*

Proof. If $d = t$, then $\overline{P_d} = \emptyset$ and t' is formed by collapsing P. Here, because t is not a predecessor of any node yet labeled in N it is assumed to be the depth increasing node of its unknown descendant until proven otherwise.

When $d \neq t$, t' is created by collapsing the nodes in $\overline{P_d}$. The lack of an edge connecting any node in $P_d - \{d\}$ to any in $\overline{P_d}$ indicates that $g(u) < p, \forall u \in input(t'), u \neq d$. Using proof by contradiction, assume d is a valid depth increasing node and that there exists $edge(u, v)$, where $u \in P_d - \{d\}$ and $v \in \overline{P_d}$. It is known $g(d) = p$ and $d \neq u$, implying $g(u) \geq p$. Therefore, $(N_t - \overline{P_d}, \overline{P_d})$ defines a cut where $u, d \in input(\overline{P_d})$ and $g(u) = g(d) = p$. By Definition 5.2.1, d is not a valid depth increasing node because $\exists edge(u, v) \in N_t$ where $d \neq u$, which is a contradiction. □

The presence of a valid $d \in N_t$ can be ensured, however, it does not guarantee that it can be identified correctly. N'_t does not guarantee that a K-feasible cut, if it exists, will not divide P_d and result in an invalid routing cut $(X, \overline{X})$ s.t. $g(u) = g(v), \forall u, v \in input(\overline{X}), u \neq v, d \in \overline{X}$. The solution is to collapse all of the nodes of P_d into d', as in Figure 5.1(b), thereby creating N'_t with d' as the lone predecessor node of t' with $g(d') = p$ when $d \neq t$, and $d' = t'$ when $d = t$. As there may be more than one valid depth increasing node, all $d \in P$ must be tested as a valid depth increasing node and for K-feasible cut. Using Lemma 5.2.5, the logic label can be used to select the d that produces minimum $h_L(X, \overline{X})$.

Any N_t that does not contain a d is deemed invalid and is eliminated from consideration. The case when $d = t$ implies that $g(t) = p$ and t is regarded as the first cell in a chain. If a valid N'_t is formed, and a K-feasible cut is found in it, a corresponding K-feasible cut can be found in N_t.

Lemma 5.2.7. *Given a valid N'_t with d', N_t has a $p-1$ height K-feasible routing cut when $d \in \overline{X}$ and p when $d \in X$ if and only if N'_t has a K-feasible routing cut.*

Proof. Let T denote the set of nodes in N_t that are collapsed into t' and D denote the set of nodes in N_t that are collapsed into d'.

If $d' \in \overline{X'}$ or $d' = t'$, then $\overline{X} = (\overline{X'} - \{d', t'\}) \cup D \cup T$ and $X = X'$. Accordingly, $(X, \overline{X})$ is a K-feasible cut of N_t because $input(\{d', t'\}) = input(D \cup T)$. Consequently, $h_G(X, \overline{X}) \leq p - 1$ because $X' = X$ does not contain any node with routing label p or higher, as all such nodes are located in $(D \cup T) \subseteq \overline{X}$. According to Lemma 5.2.4, $g(t) \geq p$ implies that $h_G(X, \overline{X}) \geq p - 1$. Since $p - 1 \leq h_G(X, \overline{X}) \leq p - 1$, then $h_G(X, \overline{X}) = p - 1$.

If $d' \in X'$, then $\overline{X} = (\overline{X'} - \{t'\}) \cup T$ and $X = (X' - \{d'\}) \cup D$. Accordingly, $(X, \overline{X})$ is a K-feasible cut of N_t because $input(t') = input(T)$. Lemma 5.2.6 yields $h_G(X, \overline{X}) = p$ because $g(d) = p$ and $d \in X$. Furthermore, Lemma 5.2.6 indicates $g(u) < p, \forall u \in input(\overline{X}), u \neq d$. □

Using a valid N'_t with d', the flow residual graph N''_t is constructed. The node cut-size problem is transformed to an edge cut-size problem by splitting each node, allowing the use of the Max-flow Min-cut algorithm. For $\{v : v \in N''_t, v \neq s, v \neq t'\}$, replace $\{v\}$ with $\{v_1, v_2\}$ connected by bridging $edge(v_1, v_2)$ with $cap(v_1, v_2) = 1$, $input(v_1) = input(v)$, and $output(v_2) = output(v)$. Give all non-bridging edges infinite capacity. The result is flow residual graph N''_t, to which the Max-flow Min-cut algorithm can be applied to determine if there is a K-feasible cut, and therefore a corresponding cut in N'_t [32]. This technique is exactly the same as that used in Lemma 4 of FlowMap [20] and is summarized in Lemma 5.2.8.

Lemma 5.2.8. *N'_t has a K-feasible routing cut if and only if N''_t has a K-feasible routing cut.*

Proof. Using the Max-flow Min-cut Theorem [32], N''_t has a cut with $e(X'', \overline{X''}) \leq K$ if and only if the maximum flow between s and t' is no more than K. Each bridging edge in flow residual graph N''_t has capacity of 1, thus the augmenting path algorithm can be used to find maximum flow. If $K + 1$ augmenting paths are found, N''_t cannot possess a K-feasible edge cut. If K or fewer augmenting paths are found, $e(X'', \overline{X''}) \leq K$, resulting in a disconnection of the N''_t before finding the $(K + 1)^{th}$ path. The K-feasible node cut $(X'', \overline{X''})$ can be identified by performing a DFS rooted at s on the nodes in N_t'' that are reachable in the residual graph. N_t'' induces a node cut $(X', \overline{X'})$ in N_t' by creating $u \in input(\overline{X'})$ corresponding to $u_1 \in input(\overline{X''})$. □

The ability of the depth increasing node to be any $\{d : d \in N_t, g(d) = p\}$ creates multiple valid $LUT(t)$ sets, each with equal routing depth but potentially different logic depth. For each N_t with a K-feasible node cut as found in N_t'', the optimal overall depth cut can be found by choosing the minimum $h_L(X_t, \overline{X_t})$ according to Equation 5.3. It must also be noted that the minimum depth routing and logic solution is not necessarily unique, and there may be alternative cuts that produce depth-equivalent solutions. This characteristic alludes to cut-enumeration techniques and can be used to encourage LE formation through choosing the cut with the minimum node cut size, $n_{min}(X_t, \overline{X_t})$.

Lemma 5.2.9. *If $h_L(X_t, \overline{X_t}) > h_L(X, \overline{X})$, the minimum routing and logic depth solution of N_t is $(X_t, \overline{X_t}) = (X, \overline{X})$.*

Let m be the number of edges in N_t. Given the preceding discussion, a minimal depth solution uses a $O(n)$ search for d, a $O(m + n)$ DFS search for its predecessors, and $O(K \cdot m)$ to identify the minimum depth routing cut for each d.

Theorem 5.2.10. *A minimum height routing cut with minimum logic depth in N_t can be found in $O(n^2 + K \cdot m \cdot n)$.*

Applying Theorem 5.2.10 in topological order yields a labeling of N_t such that the routing depth of t is minimum and, within its confines, the logic depth is also minimum. This yields a complete labeling solution for each node in N.

Corollary 5.2.11. *A minimum depth solution of N can be found in $O(n^3 + K \cdot m \cdot n^2)$.*

5.3 ChainMap Mapping

The mapping phase of the ChainMap algorithm is identical to that of FlowMap and its proof is reproduced here for the sake of completeness. It consists of creating a set T that initially contains all the POs. For each $t \in T$, a minimum height cut $(X_t, \overline{X_t})$ was computed during labeling. Using this cut, t' is created from the nodes in $\overline{X_t}$ and is the K-LUT implementing all nodes in $\overline{X_t}$. T is updated as $(T - \{t\}) \cup \mathit{input}(t')$, and the process is repeated until all of the nodes in T are PIs. It remains valid for ChainMap as long as node labeling is performed as prescribed in Section 5.2.

Theorem 5.3.1. *For any K-bounded Boolean network N, ChainMap produces a K-LUT mapping solution with minimum depth in $O(m + n)$ time.*

Proof. By induction, for any node $t \in N$, if a K-LUT t' is generated for t during the mapping phase, then the level of t' in the mapped solution is no more than $g(t)$ and $l(t)$, the depth of the optimal mapping solution for N_t. Since any solution for N induces a solution for N_t, $g(t)$ and $l(t)$ are also the minimum depths for the K-LUT generated for t in a mapping solution of N. Therefore, the mapped solution of N is optimal and requires $O(n + m)$ time [20]. □

Corollary 5.3.2. *Labeling requires $O(n^3 + K \cdot m \cdot n^2)$, and mapping requires $O(n + m)$. Hence, the first two stages of ChainMap are polynomial in $O(n^3 + K \cdot m \cdot n^2) + O(n + m) = O(n^3 + K \cdot m \cdot n^2)$. In practice, $m = O(K \cdot n)$ and $K = \{4, 5, 6\}$, making their runtime $O(n^3)$.*

A *logic chain* is defined as a series of depth increasing nodes, such that the logic depth of each consecutive chain node increases, while the routing depth remains constant.

Definition 5.3.3. *A logic chain is a subnetwork $L \subseteq N$ such that $g(u_j) = g(u_i), l(u_j) = l(u_i) + 1, \forall u_i, u_j \in L$.*

Figure 5.3 is a 2-bit full adder represented as a DAG. Figure 5.3(a) shows iteration 1 of ChainMap when $K = 3$, where nodes $\{a, b, e, f\}$ have routing and logic labels of 1 because they receive inputs directly from PIs. Figure 5.3(b) shows iteration 2, where nodes $\{c_0, c, d\}$ all receive logic and routing labels of 1 based on the same cut, occurring when each are depth increasing nodes for their respective sub-graph cones. Nodes $\{g, h, i\}$, all use a similar cut in the network which sees node c_0 as a depth increasing node, given by Figure 5.3(c). No K-feasible cut exists for which $\{g, h, i\}$ can be the depth increasing node and yield a minimum

Algorithm 1 The ChainMap Algorithm

```
procedure ChainMap($N$)
    for $v \in N$ do                                                    ▷ Phase 1:Labeling
        $l(v) = g(v) = 0$
    end for
    $T = N - PI(N)$ in topological order                                ▷ O(n+m)
    while $|T| > 0$ do
        $T = T - \{t\}$; $N_t = DFS(N, t)$; add global source $s$
        let $p = max\{g(u) : u \in input(t)\}$;
        let $q = max\{l(u) : u \in input(t)\}$
        $\overline{X_t} = \emptyset$;
        let $P = \{u : u \in N_t, g(u) = p\}$ in topological order
        for $\{d : d \in P\}$ do                                        ▷ Test all $g(d) = p$ cuts
            let $P_d = DFS(P, d)$; $\overline{P_d} = P - P_d$               ▷ Predecessors of $d$ with $g(v) = p$
            if $\exists edge(u, v), \forall u \in P_d - \{d\}, v \in \overline{P_d}$ then
                $N_t$ is invalid for $d$, skip rest of for loop
            end if
            form $d'$ by collapsing $u \in P_d$ into $d$
            if $\overline{P_d} = \emptyset$ then $t' = d'$
            else
                form $t'$ by collapsing $u \in \overline{P_d}$ into $t$
            end if
            create $N_t'$ with $t'$ and $d'$
            split $\{v : v \in N_t' : v \neq s, v \neq t\}$ into $\{v_1, v_2\}$
            assign $cap(v_1, v_2) = 1$ to bridge edges, $\infty$ to all others
            MaxFlowMinCut($N_t''$)                                     ▷ Compute max-flow, min-cut $O(Kmn)$
            if $\{\exists(X'', \overline{X''}) : e(X'', \overline{X''}) \leq K\}$ then
                induce $(X', \overline{X'})$ in $N_t'$ from $(X'', \overline{X''})$ in $N_t''$
                induce $(X, \overline{X})$ in $N_t$ from $(X', \overline{X'})$ in $N_t'$
                if $h_L(X, \overline{X}) < h_L(X_t, \overline{X_t})$ then
                    $\overline{X_t} = \overline{X}$; $X_t = X$
                end if
            end if
        end for
        if $\overline{X_t} \neq \emptyset$ then                                  ▷ If found a valid cut
            $g(t) = p$; $l(t) = h_L(X_t, \overline{X_t}) + 1$
        else
            $g(t) = p + 1$; $l(t) = h_L(X_t, \overline{X_t}) + 1$
        end if
    end while
    $T = PO(N)$                                                         ▷ Phase 2:Mapping
    while $\{t : t \in T, t \notin PI(N)\}$ do
        form LUT $t'$ by collapsing $v \in \overline{X_t}$ into $t$
        $T = (T - \{t\}) \cup input(t')$
    end while
    $T = N - PI(N)$ in reverse topological                              ▷ Phase 3:Duplication
    while $T \neq \emptyset$ do
        $T = T - \{t\}$;
        $L = \{u, v : u, v \in output(t), g(t) = g(u) = g(v)\}$
        for $u, v \in L$ do
            if $\{u, v\}$ is a valid LE and $L - \{u, v\} \neq \emptyset$ then
                Create $t'$ as a duplicate of node $t$
                $output(t) = output(t) - \{u, v\}$; $output(t') = \{u, v\}$
                $L = L - \{u, v\}$
            end if
        end for
        while $|L| > 1$ do
            $L = L - \{u\}$
            $output(t) = output(t) - \{u\}$; $output(t') = \{u\}$
        end while
    end while
end procedure
```

routing depth solution. However, c_0 can serve as a depth increasing node for each, indicating that $edge(c_0, h)$ and $edge(c_0, g)$ are chain nets. Thus nodes $\{g, h, i\}$ each have a routing label of 1 and a logic label of 2. This corresponds accurately to the $cout_0$ of bitslice 0 of a full-adder driving the sum_1 and $cout_1$ of bitslice 1.

Figure 5.3(d) shows how a mapped solution is generated from minimum-depth cuts. Initially, $T = \{c_1, s_1, s_0\}$, corresponding to all POs of the network. When c_1 is removed its inputs are added, yielding $T = \{i, s_1, s_0\}$. The cut generated when labeling node i generates an $LUT(i) = \{i, h, f, e\}$, resulting in $T = \{a_1, b_1, c_0, s_1, s_0\}$. The process is repeated, yielding $LUT(g) = \{g, f, e\}$, $LUT(c_0) = \{a, b, c, c_0\}$, and $LUT(d) = \{d, a, b\}$. Note that nodes $\{e, f\}$ and $\{a, b\}$ are implicitly duplicated by $LUT(i)$ and $LUT(g)$, and $LUT(c_0)$ and $LUT(d)$, respectively. This example demonstrates that ChainMap can successfully identify logic chains resulting from arithmetic operations.

5.4 ChainMap Duplication

The *exclusivity constraint* of chains is defined as the requirement that a chain net be a single-source, single-sink relationship between adjacent LEs. When the network is viewed as a set of LUTs, as in SIS internal representation, it means that a node t can have at most two chain outputs u and v. However, there are constraints on which LUTs can be part of the same LE, assuming that an architecture allows a full K-LUT function on the chain. Note that a discussion of N now assumes that the mapping phase has been applied, thus references to t indicate the actual K-LUT formed by collapsing the nodes in $LUT(t)$ to t.

Lemma 5.4.1. *For each $t \in N$, if $\{u, v : u, v \in output(t), v \neq u, g(t) = g(u) = g(v)\}$ satisfy the following constraints, $\{u, v\}$ can populate the same LE. If any u cannot be paired with any v, u is implemented in an LE by itself.*

(i) If $input(u) = input(v)$ and $|input(u)| = |input(v)| = K$, then u and v must compute the same function.

(ii) If $|input(u) \cup input(v)| < K$, then u and v can compute separate functions.

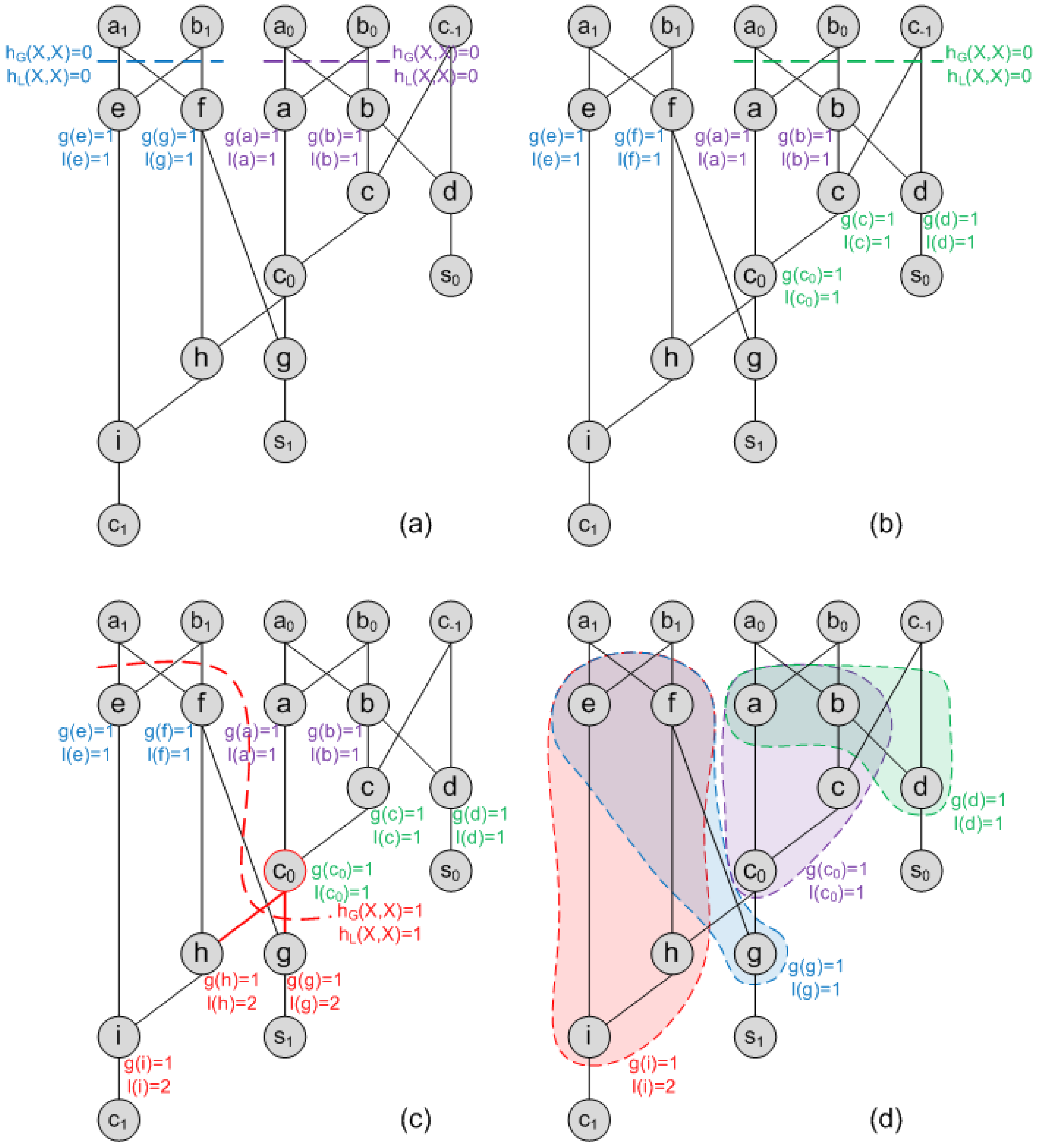


Figure 5.3 2-bit full adder for $K = 3$

(iii) *For a pair* $u, v \in output(t)$, $g(w) > g(u), \forall w \in output(u)$ *and* $g(x) = g(v), \forall x \in output(v)$.

(iv) $u \notin input(v)$ *and* $v \notin input(u)$.

In Lemma 5.4.1(i), the number of distinct inputs for nodes $\{u, v\}$ meeting $|input(u) \cup input(v)| \leq K$ does not necessarily ensure that the computation resources are available in an LE. If either $|input(u)| = K$ or $|input(v)| = K$, then $\{u, v\}$ cannot reside in the same LE because there can only be one K-input function computed by the LE, as in Figure 3.1(b). However, if both $|input(u)| < K$ and $|input(v)| < K$, the LE has enough LUT resources to accommodate both sub-width functions but is still limited to $K - 1$ distinct inputs, reflected in Lemma 5.4.1(ii), and in Figure 3.1(a). Exclusivity also requires that outputs of u and v are heterogeneous. That is, u must only source a routing net, while v must only source a chain net, or vice versa, as in Lemma 5.4.1(iii). This constraint indicates that an LE has only one available *cout* port and one *sum* port. It should be noted that the use of the terms *cout* and *sum* refer only to the type of net a node drives, chain or routing, respectively. It does not indicate the Boolean function computed by either node, it is merely borrowed nomenclature from carry-select addition. If nodes u and v are to be contained in the same LE, one must exclusively use the *cout* port, and one must exclusively use the *sum* port. Finally, Lemma 5.4.1(iv) indicates u and v cannot be dependent on each other because there is not internal LE connection between the *sum* and *cout* LUTs.

If a node has more than one chain net output, it must be duplicated if its descendants cannot meet the aforementioned constraints. Figure 5.4(a) shows a logic chain tree formed by ChainMap. In it, all routing nets are omitted, and all nodes are in logic chains. Original internal nodes are white, leaf nodes are black, and duplicate nodes are gray. Using $output(b) = \{t_1, t_3, c\}$ as an example, assume no LEs can be formed of any pair. This precipitates two duplications of b, which causes $output(a) = \{b, t_4\}$ to change to $output(a) = \{t_4, b, b, b\}$. Assuming no LEs can be formed of any pair in $\{t_4, b, b, b\}$, a is duplicated three times, which causes s to be duplicated at least three times. This pattern continues for all nodes in Figure 5.4(a), resulting in Figure 5.4(b).

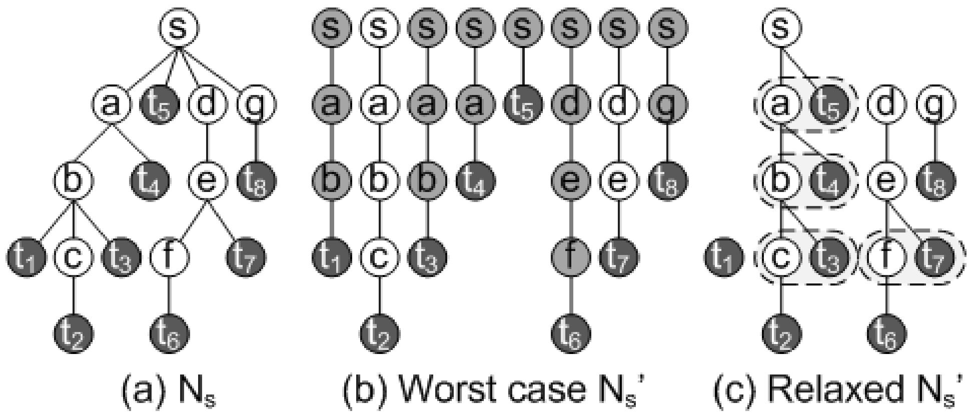


Figure 5.4 Chain tree (a) before, (b) worst case duplication, (c) average case with relaxation.

Lemma 5.4.2. *The number of node duplications required in N to satisfy exclusivity is $O(n^2)$.*

Proof. Let N_s be a subgraph consisting of edges and nodes discovered in a depth first search rooted at $s \in N$, such that for $u \in N_s, v \in output(s)$, v is visited only if $g(u) = g(v)$. By Definition 5.2.1, there can only exist one $edge(u, v) \in N_s, \forall u, v \in N_s$. Therefore, N_s is a logic chain tree with leaf nodes denoted $t_i, 1 \leq i \leq |V(N_s)|$, as in Figure 5.4(a). Additionally, there exists a logic chain $L_j, 1 \leq j \leq |V(N_s)|$ from s to t_i, pursuant to Definition 5.3.3.

The worst case area expansion occurs when u is duplicated $\forall edge(u, v), \forall u, v \in N_s, v \in output(u)$. This implies the duplication network N_s' consists of each path from s to t_i duplicated in its entirety. Figure 5.4(b) demonstrates that N_s' consists of a logic chain for each t_i, because $1 \leq i, j \leq |V(N_s)|$, $|V(N_s')| = O(|V(N_s)|) \cdot O(|V(N_s)|)$. Therefore, for N with n nodes, the number of duplications is $O(n^2)$. □

Theorem 5.4.3. *For any K-bounded Boolean network N, a $O(n^2)$ expansion is performed for n nodes in N, and ChainMap produces a depth optimal solution valid within the exclusivity constraint in $O(n^3)$ time.*

Corollary 5.4.4. *The labeling phase of ChainMap requires $O(n^3 + K \cdot m \cdot n^2)$, the mapping phase requires $O(n + m)$, and duplication requires $O(n^3)$. This makes the entire ChainMap algorithm polynomial in $O(n^3 + K \cdot m \cdot n^2) + O(n + m) + O(n^3) = O(n^3 + K \cdot m \cdot n^2)$. In practice, $m = O(K \cdot n)$ and $K = \{4, 5, 6\}$, making the complete runtime $O(n^3)$.*

The ChainMap algorithm is presented in Algorithm 5.3 and includes all three stages. ChainMap maintains a polynomial $O(n^3)$ runtime with mapped solution area bound by $O(n^2)$ of the original network. Area is a big concern because ChainMap assumes its routing delay is equivalent to that encountered in a traditional mapping solution. If the worse case is encountered, the increased wire length usurps any performance gains. Duplication is combated by relaxing chain nets to allow more nodes to comply with Lemma 5.4.1.

5.5 ChainMap Relaxation

The classic trade-off between area and speed is extremely evident in ChainMap solutions. Results indicate full duplication yields highly prohibitive area increases. For example, the number of 5-LUTs in traditional mapping versus a ChainMap solution increases from 4,752 to 9,835 for $cfft$ ($K = 5$, $before$, 2.07x). Relaxation of routing depth can be used as a means for reducing area. In return for adding a level of routing to some paths, a chain net and its duplication are eliminated. Because ChainMap makes all paths of roughly uniform routing depth, the delay of the network is dependent on the variance in logic depth. The goal is to relax paths with minimum logic depth and mask the additional routing delay with paths of high logic depth.

5.5.1 Shallowest Logic Branch Trimming

Figure 5.4(a) shows a DFS chain tree rooted at node s. Assuming Lemma 5.4.1 is fulfilled, $output(s) = \{a, t_5, d, g\}$ can form an LE of $\{a, t_5\}$. Consequently, assuming $\{d, g\}$ fulfill (i), (ii), and (iv), they still cannot form an LE because they violate (iii). Duplications occur en masse under this circumstance, along the longest network paths. Instead, if $edge(s, d)$ and $edge(s, g)$ are relaxed from chain to routing nets, the tree is disconnected at d and g, and at least 2 duplications of s are saved. Figure 5.4(c) assumes that all nodes satisfy Lemma 5.4.1, except for nodes $\{d, g\}$, which violate item (iii), and t_1 because $\{c, t_3\}$ form a valid LE. All are relaxed because they are not along the longest logic branch of their respective sub-trees. Figure 5.4(b) shows the worst case for area, while Figure 5.4(c) shows the average case ChainMap solution, with LE pairs circled in dotted lines.

A simple relaxation technique, presented in [38], masks added routing depth along the longest logic paths in a network. For all $s \in N$ and $u, v \in output(s)$, the longest DFS chain tree branch v and its valid LE mate u are preserved, while $output(s) - \{u, v\}$ are relaxed. Longer logic chains are preserved, ultimately masking the delay of the relaxed $edge(s, v)$. This heuristic method specifically targets arithmetic designs typically containing chain tree nodes with long and short logic branches. Shallowest logic branch trimming refers to the relaxation of all of the shortest logic branches rooted at each node, while the longest logic branches are preserved.

5.5.2 Least Critical Branch Trimming

The shallowest logic technique is easily applied, however, judging delay based solely on depth and not accounting for the routing depth imposed by other relaxations does not translate well to a final placed and routed design. To more accurately identify the critical connections in a design, and conversely the least critical ones, it is necessary to incorporate timing information into relaxation. To perform relaxation with criticality, the delay of a directed edge from u to $v \in output(u)$ is denoted $delay(u, v)$ and a binary delay model is used such that chain nets have delay $delay(u, v) = c$ and general routing nets have $delay(u, v) = g$.

Instead of judging critical paths solely on logic depth, the connection criticality metric employed by T-Vpack, outlined in Section 2.2.3 by Equations 2.2 to 2.8, can be employed to identify relaxations with a higher degree of accuracy. Criticality is an indication of the amount of slack available in a connection, with tie-breakers to positively bias the connection that affects the greatest number of critical paths ($critPaths(v)$, Equation 2.7), and is the least distance to PIs or POs ($D_{PI}(v)$). Rather than relaxing the shortest logic paths in $output(s)$, it is more exact to relax the connections that are least critical.

Criticality, as expressed by Equation 2.8 can not be conveyed directly to relaxation. It requires modification to more effectively address the problem of duplication in chains. The base criticality and critical path impact remain as expressed by Equation 2.5 and Equation 2.7, respectively. However, worst case network expansion occurs when the tail of a chain is duplicated, creating a ripple-effect that causes entire length of a chain to be duplicated. Therefore, the path length criticality tie-breaker $D_{PI}(v)$ is altered to stress the length of chain head, and is given precedence over critical paths. $D_{head}(v)$ measures the distance v is from the start of a chain, and creates a bias toward nodes that, if duplicated, would cause duplication across the entire chain length. Nodes that can potentially cause a large number of duplications are regarded as less critical. Another factor that expresses the likelihood of duplication is the number of chain outputs of a node, N_{cout}. As N_{cout} increases, the number of duplications increases, and criticality decreases. $D_{head}(v)$ and N_{cout} are given precedence over $critPaths(v)$, as conveyed by Equation 5.5 through reassignment of ϵ.

$$criticality(v) = baseCrit(v) + \epsilon \cdot \left(1 - \frac{D_{head}(v)}{MaxD_{head}}\right) + \epsilon^2 \cdot \left(1 - \frac{N_{cout}(v)}{MaxN_{cout}}\right) + \epsilon^3 \cdot critPaths(v) \tag{5.5}$$

For relaxation via branch trimming, the network is parsed in reverse topological order (PO to PI), and each node s possessing chain descendants, $D = \{v : v \in output(s), delay(s,v) = c\}$, is subjected to relaxation. If $|D| < 2$, s requires no duplication. If $|D| > 1$ or more chain descendants, the best LE formed by $\{u, v \in D\}$ consisting of v where $criticality(s,v) > criticality(s,u), \forall u \in D$ and u where $criticality(s,u) > criticality(s,w), \forall w \in D - \{v\}$ and u is a valid LE mate to v. That is, the LE contains v, the chain with highest criticality, and u, the chain with the second highest criticality that forms a valid LE pair with v. Note that if there is not a node u that forms a valid LE with v, $u = \emptyset$ is a valid condition. Once the highest value LE is chosen, $delay(s,w) = g, \forall w \in D - \{v,u\}$, i.e all other chain outputs are relaxed.

Least critical branch trimming, Algorithm 5.5.2, is called by ChainMap instead of duplication (Section 5.4). The network is parsed in reverse topological order to assure that each node's descendants have been successfully relaxed before exclusivity and relaxation are applied to the current node. The procedure call $ComputeNetworkCriticality\{N\}$ refers to the process outlined in [12] used to compute network slack using Equation 5.5. An important characteristic of least critical branch trimming is that it allows no duplications. For every node possessing chain outputs, its chain descendants either constitute the "best LE", or are relaxed. Because of the application of least critical branch trimming, no node duplications are required by ChainMap's duplication phase.

Algorithm 2 Least Critical Branch Trimming

```
1: procedure TrimBranches($N$)
2:     $S = N - PI(N) - PO(N)$ in reverse topological order
3:     while $S \neq \emptyset$ do
4:         ComputeNetworkCriticality($N$)          ▷ Re-compute network criticality
5:         $S = S - \{s\}$
6:         $D = \{v : v \in output(s), delay(s, v) = c\}$
7:         Define $v$ s.t. $criticality(s, v) > criticality(s, w), \forall w \in D$
8:         Define $u$ s.t. $criticality(s, u) > criticality(s, w), \forall w \in D - \{v\}$ and $u, v$ are a valid LE
9:         for $w \in D - \{u, v\}$ do
10:            $delay(s, w) = g$
11:        end for
12:    end while
13: end procedure
```

5.5.3 Global Least Critical Relaxation

Disallowing duplications posits the question: can they be beneficial or are they universally undesirable? An alternative to branch trimming is global least critical relaxation. It incorporates the same notion of criticality used in branch trimming (Equation 5.5), but instead of traversing the network in reverse topological order and requiring all chain descendants not part of the "best LE" be relaxed, the least critical chain net that can accommodate relaxation in the entire network is relaxed each iteration. The processes continues until no further relaxation can be tolerated, at which point the original duplication phase of ChainMap (Section 5.4) is invoked. Global least critical relaxation deviates from branch trimming in the following ways:

- Relaxation is performed after the mapping phase and before duplication.
- Duplications are allowed.
- The critical path of the network is bounded by a user-defined value.
- Relaxation is only performed on connections that can accommodate it.

The relaxation phase occurs after mapping and before duplication of the network. Global relaxation has the goal of finding all connections that are capable of accommodating a relaxation, ranked in order from least to most critical, and recomputed after each relaxation. Once all connections have been accommodated and relaxed, the network is duplicated to comply with the exclusivity constraint. Any duplications that are necessary in the relaxed network are performed, regardless of area impact.

Estimated network delay is calculated according to Equation 2.2 as the maximum arrival time of all POs. The designer has the option of granting the network global slack, or limit it to that which is natively available in the technology map. Instituting global network slack, Δ increases the slack available in all paths by setting $T_{required}(v) = max\{T_{arrival}(u)\} + \Delta, \forall u, v \in PO(N)$. This directly effects Equations 2.3 and 2.4, and allows an increased number of relaxations to occur. In fact, global network slack is the primary way through which duplications are eliminated using global least critical relaxation. The procedure call $ComputeNetworkCriticality\{N, T_{max}\}$ refers to the process outlined in [12] used to compute network slack using Equation 5.5, with the exception that the maximum required time is incremented by Δ to provide global slack.

Relaxations are only allowed on connections that can accommodate it. This means that the amount of slack available in a chain connection has to meet or exceed the cost of a general routing connection. Any connection not able to accommodate a relaxation is left alone, regardless of criticality value. Varying the $G : L$ ratio of the network serves as the secondary method for eliminating duplications. As $G : L$ increases, either the global network slack or number of duplications must increase. Conversely, for values of $G : L \approx 1$, the number of relaxations will increase as will the potential delay through the network. Experiments have determined that the amount of global slack applied is about three general routing connections, with $G : L = 3$.

Algorithm 5.5.3 presents global least critical relaxation. It contains a loop that executes as long as a connection that can accommodate a relaxation exists in the network. Each iteration of the loop, the network slack must be updated, after which the accommodating connection with the lowest criticality is relaxed. After each relaxation takes place, the slack of each path containing it must be updated. This has the effect of changing the criticalities and slack of most, if not all, of the connections in the network. The least critical relaxation occurs before the ChainMap duplication phase.

Algorithm 3 Global Least Critical Relaxation

```
1: procedure RelaxLeastCritical($N$)
2:     ComputeNetworkCriticality($N$)                ▷ Initialize network criticality
3:     $T_{max} = T_{arrival}(v)$ s.t. $T_{arrival}(v) > T_{arrival}(u), \forall u, v \in PO(N)$
4:     $T_{max} = T_{max} + \Delta$                  ▷ Adjust max arrival time with global network slack
5:     ComputeNetworkCriticality($N$, $T_{max}$)     ▷ Re-compute network criticality with max arrival time
6:     while $\exists u, v \in N$ s.t. $delay(u, v) \leq slack(u, v)$ do
7:         if $\exists u, v : delay(u, v) \leq slack(u, v) \wedge criticality(u, v) < criticality(w, x), u, v, w, x \in N$ then
8:             $delay(u, v) = g$                     ▷ Relax the least critical accommodating connection
9:         end if
10:        ComputeNetworkCriticality($N$, $T_{max}$) ▷ Re-compute affected paths with max arrival time
11:    end while
12: end procedure
```

The branch trimming approaches also use global least critical relaxation after the duplication phase as a means for providing subsequent CAD flow stages greater flexibility. Global least critical relaxation applied after branch trimming essentially finds all relaxations that can be had for free, i.e. chains that don't influence the critical path. Fewer chains in the network means greater solution space for clustering, placement, and routing. The only modification to the basic global least critical algorithm is that LEs which source or sink chain connections are disallowed to relax. This prevents the dissolution of LE relationships that are important for area reasons. Chains can be used to reduce the area of a design, a characteristic that will be discussed in Section 6.1, with the presentation of ChainPack.

5.6 Post-Technology Map Results

To accurately assess the effectiveness of the ChainMap algorithm, it is necessary to test designs with HDL defined arithmetic carry chains. For this purpose OpenCores [62] DSP, security, and controller benchmarks have been selected with a range of arithmetic penetrance. Figure 5.5 depicts the design flows, each inserting arithmetic at different points:

- **Forget** - Arithmetic chains are optimized by synthesis and technology mapped by ChainMap without HDL.

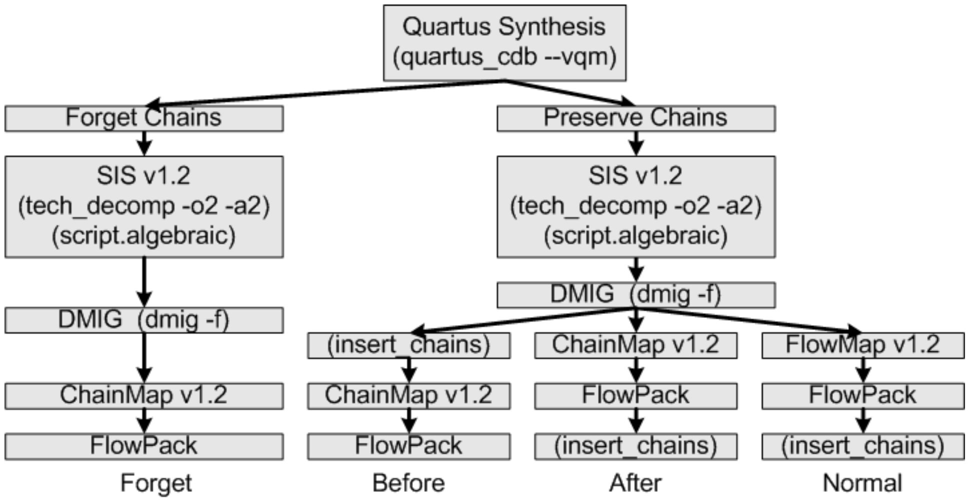


Figure 5.5 Experimental Design Flows

- **Before** - Arithmetic chains are preserved through synthesis, and reinserted before ChainMap technology mapping.
- **After** - Arithmetic chains are preserved through synthesis and ChainMap technology mapping, and reinserted before clustering, placement, and routing.
- **Normal** - Arithmetic chains are preserved through synthesis and FlowMap technology mapping, and reinserted before clustering, placement, and routing.

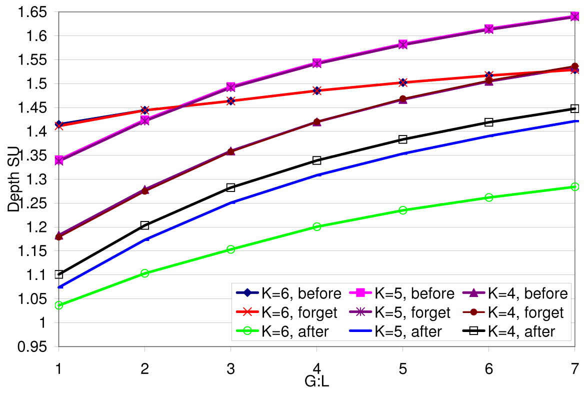
Quartus II has an open netlist format, VQM, and an open design flow where academic tools can be tested [55]. Because SIS lacks HDL elaboration, a parser has been created to implement a VQM netlist in SIS internal representation. An option has been included to preserve arithmetic carry chains or implement them as bit-sliced *cout* and *sum* operations. The drawback to using Quartus II for HDL interpretation is that optimization and K-LUT mapping on the netlist has been performed before importing to SIS. To mitigate this, the logic network is decomposed into 2-input AND and OR gates and resynthesized with SIS using *script.algebraic*. The speedup and area (i.e. number of LUTs) results produced by the three ChainMap flows are normalized to the *normal* flow. Speedup values greater than 1 represent a decrease in delay. An LUT ratio of less than 1 indicates area savings.

Figure 5.6(b) shows optimal and shallowest logic branch trimming speedup for full-width chains averaged across all benchmarks under all three flows for $K = \{4, 5, 6\}$. Likewise Figure 5.6(b) shows optimal and shallowest logic branch trimming speedup for sub-width chains. The independent axis is the ratio of average routing delay to LUT delay ($G : L$). Since routing delay is variable, Figure 5.6(b) shows how speedup is affected by changes in average routing delay relative to static LUT delay. Changing $G : L$ shows how the effectiveness of the heuristic relaxation technique changes as average routing delay increases. Common $G : L$ lies within the range of $[2, 4]$, which for Stratix is akin to an LUT delay of 366 ps and routing delay of $[732ps, 1464ps]$.

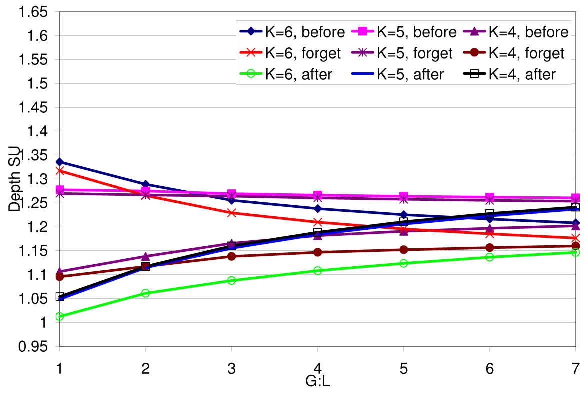
The timing-based relaxation techniques in Sections 5.5.2 and 5.5.3 are not presented because post technology map estimation using $G : L$ does not model circuit timing with enough accuracy to provide a reasonable assessment of performance. Optimal and shallowest logic results give an indication of potential performance without the place and route experiments necessary to fully judge timing-based approaches. Simple estimation allows a narrowing of the solution space without comprehensive experiments.

Tables 5.1, 5.2, and 5.3 show results for all benchmarks using full-width chains. They present the optimal and relaxed routing (G_o, G) and logic (L_o, L) of the path with maximum routing depth and maximum logic depth, the speedup when $G : L = 3$, the relaxed number of LUTs used, and ratio of ChainMap LUTs to *normal* (λ). They indicate that in all cases the optimal ChainMap solution is faster than HDL dictated chains. However, the relaxed solutions represent a mixed record of taking advantage of this potential speedup, but do consistently reduce the overall LUT utilization of a benchmark.

Benchmark results indicate optimal ChainMap performance varies with flow and LUT size, but are equal to or better than *normal*, as expected. Varying the value of K produces results that mirror the expected result of incorporating more logic into each LUT; as LUT size increases, speedup increases and area decreases. Across all LUTs, the *before* and *forget* flows closely mirror each other, with an average difference of approximately 5%. This is a very important result, as it means that arithmetic chains can be discovered and mapped without

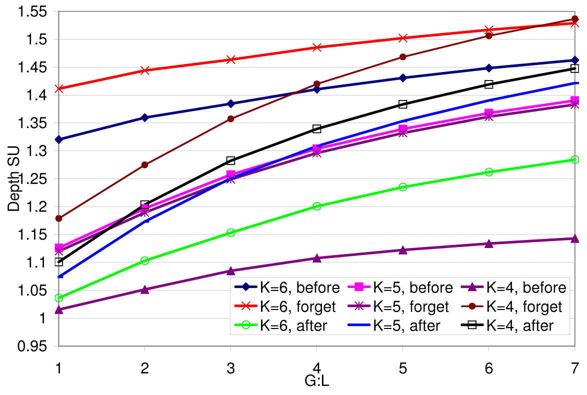


(a) Optimal

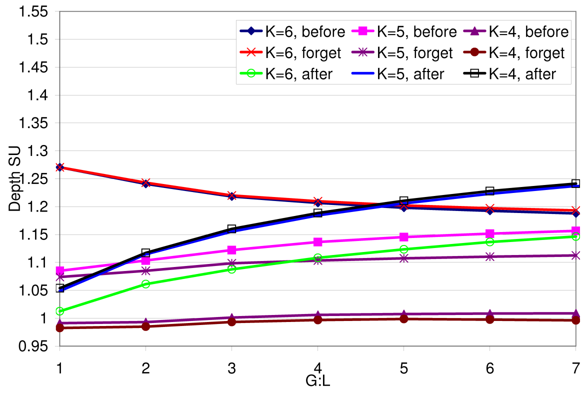


(b) Shallowest logic branch trimming

Figure 5.6 Full-width speedup of ChainMap flows relative to normal flow vs. average routing to LUT delay ratio.



(a) Optimal



(b) Shallowest logic branch trimming

Figure 5.7 Sub-width speedup of ChainMap flows relative to normal flow vs. average routing to LUT delay ratio.

relying on HDL macros. Although ignoring HDL macros and using ChainMap with relaxation produces solutions typically between 0.95x and 1.4x the speed of the *normal* case, the optimal results indicate that there are still potential performance increases to be realized.

An interesting phenomena occurs in Figure 5.6(b) where, as $G : L$ increases, the effectiveness of relaxed *before* and *forget* increases for $K = 4$, holds steady for $K = 5$, and decreases for $K = 6$, while *after* increases for all K. This is due to the disparate affect that the relaxation technique has on *before* and *forget* versus *after*, and the decrease in nets as K increases. Because relaxation is applied to shorter logic paths to mask its effect with longer logic paths, as $G : L$ increases, the ability to mask relaxed nets decreases for larger K. This does not affect the *after* flow, because very few chain nets can be identified by ChainMap when HDL macros are excluded during mapping, thus relaxation is rarely applied. $K = 4$ maintains relatively deep logic depth due to many LUTs and nets, while logic depth is reduced for $K = \{5, 6\}$, revealing their lack of ability to mask relaxations as effectively.

The most heavily arithmetic design, the radix-4 FFT, yields a relaxed solution that is 1.00x speedup of *normal*, and an optimal solution of 1.11x ($cfft$, $K = 5$, $before$). This indicates that ChainMap, coupled with the relaxation procedure in Section 5.5, produces chains at least as well as HDL macros, but that there may exist other less aggressive LUT reduction relaxation techniques. The LUT results reflect this, with the ChainMap solution 0.71x that of *normal*, indicating optimal performance can potentially be recouped through different relaxation techniques, or relying on the smaller design to yield shorter wires during PNR.

The phenomena of area reduction applies to nearly all designs tested and can potentially increase speedup values universally. It stems from two sources, the first being that the chain cut is a naturally more area aggressive. If a node fails to join a logic level q $(d = t)$ because of a cut size of greater than K, ChainMap searches out an alternate K-feasible cut $(d \neq t)$. This cut is an alternative to implementing the node on a new logic level and thus each chain cut tends to incorporate more nodes. The second, more prevalent, reason is that preserved arithmetic chains are typically 3-input gates that are not merged with others and are ultimately

Table 5.1 Performance Summary for OpenCores Benchmarks, K=4

	Normal			Forget								Before								After							
Design	G	L	LUT	G_o	L_o	SU_o	G	L	SU	LUT	λ	G_o	L_o	SU_o	G	L	SU	LUT	λ	G_o	L_o	SU_o	G	L	SU	LUT	λ
cfft	5	4	4639	3	38	1.06	5	37	0.96	3958	0.85	3	38	1.06	5	37	0.96	3764	0.81	4	6	1.00	4	6	1.00	4640	1.00
mlt3x3	2	34	901	2	33	1.03	3	32	0.98	997	1.11	2	33	1.03	3	16	1.03	850	0.94	2	34	1.00	2	34	1.00	901	1.00
reedsol	9	8	1411	7	9	1.17	10	9	0.90	1401	0.99	7	9	1.17	10	9	0.90	1401	0.99	7	9	1.17	10	9	0.90	1407	1.00
jpeg	6	15	5890	4	8	1.32	7	8	1.14	6546	1.11	4	7	1.32	6	6	1.18	5493	0.93	5	16	1.06	6	16	0.97	5916	1.00
dct	4	8	4767	2	19	1.12	3	19	1.00	5597	1.17	2	19	1.12	3	19	1.00	4572	0.96	3	8	1.00	4	8	1.00	4767	1.00
eth	7	6	301	4	6	2.00	6	5	1.50	307	1.02	4	6	2.00	6	5	1.50	302	1.00	5	9	1.13	6	5	1.03	325	1.08
usb	8	7	3587	5	8	1.35	6	8	1.19	3609	1.01	5	8	1.35	6	8	1.19	3569	0.99	5	8	1.35	6	8	1.19	3738	1.04
xtea	6	36	982	4	36	1.13	7	34	0.98	1163	1.18	4	36	1.13	5	36	1.06	1034	1.05	5	31	1.10	5	31	1.10	990	1.01
des3	7	6	946	6	7	1.08	6	7	1.08	1056	1.12	6	7	1.08	6	7	1.08	1056	1.12	6	7	1.08	6	7	1.08	1064	1.12
rsa	7	39	1227	4	36	1.25	7	35	1.07	1234	1.01	4	36	1.25	7	35	1.07	1002	0.82	6	38	1.07	6	39	1.05	1194	0.97
md5	18	78	2600	13	76	1.15	24	74	0.90	3033	1.17	14	75	1.13	22	71	0.96	2872	1.10	15	38	1.08	18	68	1.03	2838	1.09
sha512	8	70	5908	7	72	1.01	12	70	0.89	6702	1.13	7	69	1.04	11	68	0.93	5855	0.99	8	70	1.00	8	70	1.00	5780	0.98
twofish	55	54	2748	20	64	1.77	26	55	1.65	3696	1.34	20	64	1.77	26	55	1.65	3696	1.34	20	64	1.77	26	55	1.65	3696	1.34
ava	30	34	13670	8	26	2.48	19	29	1.44	14543	1.06	8	26	2.48	19	29	1.44	14894	1.09	8	26	2.48	19	34	1.36	14772	1.08
aes128	15	14	13286	9	15	1.37	12	16	1.13	15311	1.15	9	15	1.37	12	16	1.13	15311	1.15	9	15	1.37	12	16	1.13	15311	1.15
Total	187	413	62863	98	453	–	153	438	–	69153	–	99	448	–	147	417	–	65671	–	108	379	–	138	406	–	67339	–
Ratio	1.00	1.00	1.00	0.52	1.10	1.36	0.82	1.06	1.14	1.10	1.10	0.53	1.08	1.36	0.79	1.01	1.17	1.04	1.04	0.58	0.92	1.28	0.74	0.98	1.16	1.07	1.07

Table 5.2 Performance Summary for OpenCores Benchmarks, K=5

	Normal			Forget								Before								After							
Design	G	L	LUT	G_o	L_o	SU_o	G	L	SU	LUT	λ	G_o	L_o	SU_o	G	L	SU	LUT	λ	G_o	L_o	SU_o	G	L	SU	LUT	λ
cfft	4	6	4749	3	36	1.11	5	34	1.02	3840	0.81	3	36	1.11	5	35	1.00	3357	0.71	3	41	1.00	5	5	1.00	4639	0.98
mlt3x3	2	34	901	2	17	1.74	3	17	1.54	760	0.84	2	17	1.74	3	17	1.54	754	0.84	2	34	1.00	2	34	1.00	901	1.00
reedsol	7	6	1231	5	11	1.04	7	9	0.90	1217	0.99	5	11	1.04	7	9	0.90	1217	0.99	5	11	1.04	7	9	0.90	1225	1.00
jpeg	6	13	5875	4	6	1.72	6	6	1.29	5059	0.86	4	6	1.72	6	6	1.29	4916	0.84	5	16	1.00	6	16	0.91	5887	1.00
dct	4	8	4767	2	11	1.65	3	10	1.47	4123	0.86	2	11	1.65	3	10	1.47	4059	0.85	3	8	1.00	4	8	1.00	4767	1.00
eth	6	5	258	4	9	1.62	6	6	1.42	242	0.94	4	9	1.62	6	6	1.42	242	0.94	4	20	1.06	6	8	1.00	267	1.03
usb	8	7	3111	5	8	1.35	6	8	1.19	3211	1.03	5	8	1.35	6	8	1.19	3186	1.02	5	8	1.35	6	8	1.19	3387	1.09
xtea	6	36	1009	4	30	1.29	6	29	1.15	900	0.89	4	30	1.29	6	29	1.15	910	0.90	5	32	1.13	5	32	1.13	974	0.97
des3	7	6	824	5	8	1.17	6	8	1.04	993	1.21	5	8	1.17	6	8	1.04	993	1.21	5	8	1.17	6	8	1.04	1002	1.22
rsa	6	38	1132	4	21	1.70	7	20	1.37	928	0.82	4	21	1.70	7	19	1.40	912	0.81	5	39	1.04	6	38	1.00	1135	1.00
md5	18	58	2569	12	52	1.41	21	26	1.11	2498	0.97	12	51	1.43	21	41	1.12	2465	0.96	15	51	1.06	16	75	1.01	2517	0.98
sha512	8	70	5518	6	68	1.09	10	65	0.99	4854	0.88	6	68	1.09	9	66	1.01	4828	0.87	8	70	1.00	8	70	1.00	5358	0.97
twofish	50	49	2602	13	60	2.01	23	56	1.59	3100	1.19	13	60	2.01	23	56	1.59	3100	1.19	13	60	2.01	23	56	1.59	3100	1.19
ava	22	26	13415	8	24	1.92	11	19	1.77	11807	0.88	8	24	1.92	11	19	1.77	11989	0.89	8	24	1.92	11	19	1.77	12501	0.93
aes128	13	12	11939	7	16	1.38	11	13	1.11	12703	1.06	7	16	1.38	11	13	1.11	12703	1.06	7	16	1.38	11	13	1.11	12703	1.06
Total	167	374	59900	84	377	–	131	326	–	56235	–	84	376	–	130	342	–	55631	–	93	438	–	122	399	–	60363	–
Ratio	1.00	1.00	1.00	0.50	1.01	1.49	0.78	0.87	1.26	0.94	0.94	0.50	1.01	1.49	0.78	0.91	1.27	0.93	0.93	0.56	1.17	1.25	0.73	1.07	1.16	1.01	1.01

implemented as lone, underpopulated LUTs. ChainMap allows these underpopulated LUTs to be packed together.

5.7 Summary

ChainMap provides a polynomial time solution to the problem of identifying generic logic chains in a Boolean network. By looking at the problem of circuit depth from the perspective of minimizing routing depth, it has been shown that considerable performance gains are potentially available. The important contributions of ChainMap are as follows:

Table 5.3 Performance Summary for OpenCores Benchmarks, K=6

	Normal			Forget								Before								After							
Design	G	L	LUT	G_o	L_o	SU_o	G	L	SU	LUT	λ	G_o	L_o	SU_o	G	L	SU	LUT	λ	G_o	L_o	SU_o	G	L	SU	LUT	λ
cfft	4	6	4740	3	19	1.79	5	18	1.52	3162	0.67	3	20	1.72	5	19	1.47	3060	0.65	3	41	1.00	4	3	1.00	4620	0.97
mlt3x3	2	34	901	2	17	1.74	3	16	1.60	817	0.91	2	17	1.74	3	16	1.60	748	0.83	2	34	1.00	2	34	1.00	901	1.00
reedsol	6	5	1212	5	6	1.10	5	6	1.10	1130	0.93	5	6	1.10	5	6	1.10	1130	0.93	5	6	1.10	5	6	1.10	1138	0.94
jpeg	5	14	5875	4	5	1.71	5	5	1.45	5248	0.89	4	5	1.71	5	5	1.45	4789	0.82	4	14	1.00	5	14	1.00	5877	1.00
dct	3	7	4766	2	10	1.75	3	10	1.47	4441	0.93	2	10	1.75	3	10	1.47	3993	0.84	3	7	1.00	3	7	1.00	4766	1.00
eth	6	5	255	4	7	1.79	5	6	1.62	221	0.87	4	7	1.79	5	6	1.62	218	0.85	4	20	1.06	5	8	1.06	245	0.96
usb	6	5	2815	4	7	1.21	6	6	0.96	2685	0.95	4	7	1.21	6	6	0.96	2662	0.95	4	7	1.05	6	6	0.96	2876	1.02
xtea	5	35	915	4	28	1.25	6	28	1.09	876	0.96	4	28	1.25	6	28	1.09	746	0.82	5	31	1.06	5	31	1.06	912	1.00
des3	4	3	347	4	3	1.00	4	3	1.00	338	0.97	4	3	1.00	4	3	1.00	338	0.97	4	3	1.00	4	3	1.00	347	1.00
rsa	6	38	1120	4	19	1.81	7	19	1.40	954	0.85	4	19	1.81	7	19	1.40	814	0.73	5	38	1.06	6	38	1.00	1127	1.01
md5	15	52	1730	11	44	1.47	20	44	1.09	2041	1.18	11	43	1.49	18	43	1.16	1945	1.12	13	73	1.01	15	72	0.97	2129	1.23
sha512	8	70	5362	6	68	1.09	11	15	0.96	4741	0.88	6	68	1.09	8	66	1.04	4492	0.84	8	70	1.00	8	70	1.00	5118	0.95
twofish	40	39	2559	13	57	1.66	23	45	1.36	2797	1.09	13	57	1.66	23	45	1.36	2797	1.09	13	57	1.66	23	45	1.36	2797	1.09
ava	17	21	10394	8	19	1.67	10	18	1.50	9960	0.96	8	19	1.67	10	18	1.50	10269	0.99	8	19	1.67	10	18	1.50	10708	1.03
aes128	9	8	3921	6	12	1.17	8	13	0.95	4777	1.22	6	12	1.17	8	13	0.95	4777	1.22	6	12	1.17	8	13	0.95	4777	1.22
Total	136	342	46912	80	321	–	121	252	–	44188	–	80	321	–	116	303	–	42778	–	87	432	–	109	368	–	48338	–
Ratio	1.00	1.00	1.00	0.59	0.94	1.46	0.89	0.74	1.23	0.94	0.94	0.59	0.94	1.46	0.85	0.89	1.26	0.91	0.91	0.64	1.26	1.15	0.80	1.08	1.09	1.03	1.03

1. A formal logic chain definition is presented that encompasses both arithmetic and non-arithmetic operations.
2. ChainMap creates generic logic chains in polynomial time without HDL arithmetic chain macros.
3. An area trade-off is necessary due to the exclusivity constraint of current FPGA carry chain architectures, but can be mitigated with relaxation.
4. Three different relaxation techniques have been proposed: shallowest logic branch trimming, least critical branch trimming, and global least critical relaxation.
5. Optimal ChainMap results describe a baseline of performance for logic chains.

The definition of a logic chain has been formalized as a series of nodes, such that there is a directed $edge(u, v)$ between adjacent nodes $\{u, v\}$, that causes the logic depth of v to increase while not increasing its routing depth. This definition addresses the fact that there is a clear difference in the speed of routing versus chain nets and guides their use. The average speedup of ChainMap versus a traditional mapping algorithm with HDL chains is 1.4x optimally and 1.25x relaxed, for $K = \{4, 5, 6\}$ and reasonable average routing delays. While all K provide performance gains, when $K = \{5, 6\}$, underpopulated HDL macro LUTs can more often be packed together, yielding slightly higher average speedup and LUT savings. This result concurs with results for general networks, where $K = \{5, 6\}$ yield the best depth for LUT-based FPGAs [68]. An assessment of the impact ChainMap has on place and route is presented in Chapter 7.

ChainMap requires an area/speedup trade off, an artifact of FPGAs enforcing the *exclusivity constraint*. However, the shallowest logic branch trimming relaxation heuristic presented allows ChainMap to produce consistent area reductions. Area reductions of up to 0.71x LUTs are witnessed ($cfft$, $K = 5$, $before$) with neutral speedup, and the potential to increase speed through shorter wires. Optimal solutions, while prohibitive from an area standpoint, indicate that better relaxation techniques have the potential to yield ubiquitous speedup increases.

The results presented in this work are indicative of LEs which can operate in both *(K-1)* and K-LUT modes, as depicted in Figure 3.1, and supported by the Stratix and chain reuse cell of Chapter 3 [36]. Sub-width chains can be encouraged by choosing alternative minimum depth cuts that possess the smallest node cut size, $n_{min}(X, \overline{X})$. This technique is useful because it allows more nodes to comply with the $K-1$ input aspect of the exclusivity constraint, thus encouraging LE formation. The ChainMap algorithm can be adapted to support pure carry-select, *(K-1)*-LUT chains, by searching for a *(K-1)*-feasible cut when $d \neq t$, and a K-feasible cut when $d = t$. However, the pure carry-select results in Figure 5.7 indicate that valuable performance is lost by not using the reuse cell of Chapter 3 to achieve full-width chains.

The average estimated performance difference between disregarding HDL macros completely and inserting chains before mapping is within 5%, indicating HDL preservation might potentially be abandoned. This could affect the entire FPGA design flow, allowing CAD designers to expand algorithms past the partitions created by HDL. Since the best area/speedup is usually achieved by the insertion of arithmetic chains *before* mapping, the inference is that they are already highly optimized in terms of literal count, and resynthesis creates sub-optimality. ChainMap demonstrates that generic logic chains perform better than solely arithmetic ones, a result that could lead to innovative FPGA architectures.

CHAPTER 6. POST-TECHNOLOGY MAP CHAIN HANDLING

Logic chains are primarily intended to increase the speed and efficiency of designs, but they also have an overlooked effect on the area of a design. Even though chains are a single fanout connection between adjacent LEs, they are a dual-fanout net between LUTs, as currently supported in FPGA architectures. This allows two sub-width LUTs to be packed into a single LE. This property of logic chains creates an additional opportunity for a reduction in the area of a design, as measured by the number of LEs required. Packing LUTs into LEs using chains can be applied to a valid network after technology mapping to reduce overall LE consumption.

The problems of clustering, placement, and routing have already been widely studied, and a plethora of solutions exist. However, there is a relively small body of publicly available work from FPGA vendors and researchers addressing the handling of logic chains. HierARC is presented as a generic clustering tool which can also accommodate the special constraints of chains. The chain clustering techniques presented by HierARC are simple strategies used to assess the effectiveness of ChainMap solutions across the entire design flow. They provide an intuitive and/or initial basis for dealing with chains, although they are quite possibly not the most effective solutions. However, due to the reluctance of industry to divulge commercial techniques, the lack of published literature on the subject, and the inability of current tools to deal with chains, they borrow very little from existing approaches.

6.1 ChainPack: Chains for Area Reduction

ChainMap selects nets to implement in the fast chain routing when it must do so for purposes of routing depth optimality. However, it does not select every chain possible, rather avoiding the use of a chain net if a node is equidistant from its predecessors. This has the effect of leaving many nets to be implemented in general routing when they don't improve depth, but may overlook opportunities to reduce the number of LEs.

Figure 6.1(a) shows a possible solution found by ChainMap, where each path in the network is of optimal routing depth, or suitable relaxed routing depth. Much of the connectivity of the network has been omitted for clarity, except for the key chain and routing nets, depicted by solid and dashed lines, respectively. The chain nets implemented are only those necessary to fulfill optimality and exclusivity, but they are not the only possible ones that can be used. Each LUT in Figure 6.1(a) must be implemented in its own LE due to Lemma 5.4.1(iii).

In this example, assume that all of Lemma 5.4.1 has been fulfilled for all node pairs with a common predecessor, except for the heterogeneous LE output constraint expressed in item (iii). Heterogeneous outputs refer to the characteristic that an LE has one available output to the general routing array, and one output to the next adjacent LE in a logic chain. This constraint, coupled with the solution generated by ChainMap, creates a design that has 13 LUTs and 13 LEs. However, if non-critical routing nets can instead be implemented by chain nets, the number of LEs can be drastically reduced by LUT pairs sharing physical LE resources.

Figure 6.1(b) shows how LEs can be formed first from node pairs $\{g, h\}$ and $\{j, k\}$ by implementing $edge(d, h)$ and $edge(f, k)$ in chain nets. Subsequently, an LE can be formed of $\{d, e\}$ if $edge(b, e)$ is converted to a chain net. Finally, $\{b, c\}$ can share an LE if $edge(a, c)$ is implemented in a chain net. LEs can be formed of LUT pairs by traversing a Boolean network in reverse topological order and identifying those that comply with Lemma 5.4.1 given a net's change from routing to chain. In this example, the number of LEs in the network is reduced from 13 to 9 without disrupting solution routing depth.

ChainPack is given in Algorithm 6.1, and works by identifying LEs that can be formed from the fanouts of each possible node for the purposes of area reduction. Network N is traversed in reverse topological order for all $t \in N$ and LEs are identified among $\{u, v : u, v \in output(t), u \neq v\}$ pursuant to Lemma 5.4.1. This ordering of nodes allows the output characteristics of t to be determined before it is processed as the fanout of a node. This is important because the node's compliance with Lemma 5.4.1(iii) is known when it is being considered for LE membership. If t

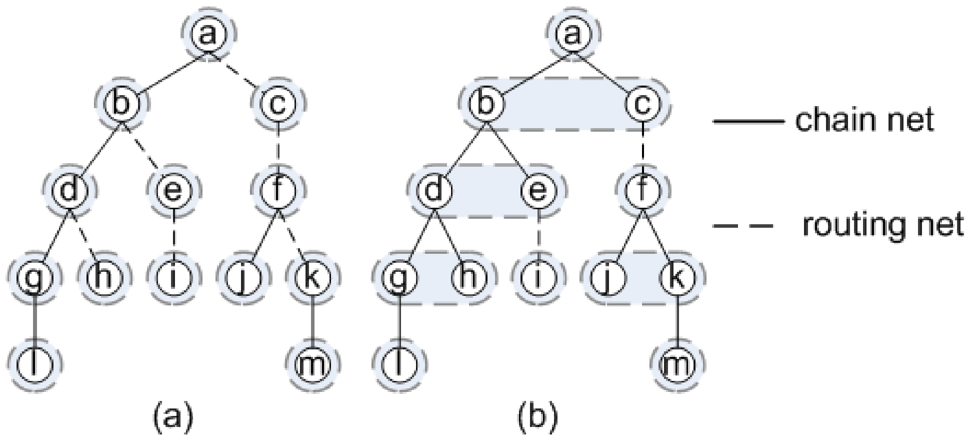


Figure 6.1 ChainPack example with 13 LUTs, (a) initial ChainMap solution with 13 LEs and, (b) after ChainPack with 9 LEs.

already belongs to an LE, no changes can be made that would cause it to violate Lemma 5.4.1. Nets can only be converted from routing to chain, thus preserving the routing depth of the ChainMap solution.

Algorithm 4 ChainPack

```
1:  procedure ChainPack($N$)
2:      $S = N - PI(N) - PO(N)$ in reverse topological order
3:      while $S \neq \emptyset$ do
4:          $S = S - \{s\}$
5:          $D = \{v : v \in output(s), delay(s, v) = c\}$          ▷ All chain outputs of $s$
6:          while $D \neq \emptyset$ do          ▷ While there exists chain outputs to be processed
7:              Define $\{u, v\}$ s.t. $u, v \in D$ and $u, v$ are a valid LE          ▷ $u = \emptyset$ occurs if no LE mate for $v$
8:              if $D - \{u, v\} \neq \emptyset$ then          ▷ If this is not the last LE
9:                  Create $s'$          ▷ The duplicate node of $s$
10:                 $input(s') = input(s)$          ▷ Give $s'$ the same inputs as $s$
11:                 for $v \in input(s)$ do          ▷ Add $s'$ to the inputs of $s$
12:                     $output(v) = output(v) \cup s'$
13:                 end for
14:                 $output(s') = \{u, v\}$          ▷ The duplicate sources the LE formed by $\{u, v\}$
15:                 $input(v) = (input(v) - \{s\}) \cup \{s'\}$          ▷ Remove $s'$ from half of the LE
16:                 if $u \neq \emptyset$ then          ▷ If $v$ has a valid LE mate $u$
17:                     $input(u) = (input(u) - \{s\}) \cup \{s'\}$          ▷ Remove $s'$ from the other half of the LE
18:                 end if
19:                 Let $s'$ have the same function as $s$          ▷ Set the function of $s'$
20:                 $D = D - \{u, v\}$          ▷ Update duplicate set
21:             end if
22:         end while
23:     end while
24: end procedure
```

ChainPack can only be applied after relaxation and duplication because it relies on a valid netlist. Table 6.1 presents results typical of the application of ChainPack on a full-width, global least critical relaxed ChainMap solution. The results indicate that measuring area from

Table 6.1 Sample ChainPack results for full-width, $before$, global least critical relaxation, $K = 5$.

	Normal			ChainPack			
	LUT	FF	LE	LUT	FF	LE	%LE
xtea	1300	189	1264	1322	189	1206	-4.81
usb	3061	1758	3301	3124	1758	3281	-0.61
rsa	1313	428	1349	1314	428	1341	-0.60
md5	2924	910	3416	2998	910	3374	-1.24
des3_area	867	9	957	885	9	874	-9.50
ethernet	242	121	299	243	121	290	-3.10
reed_sol	1208	539	1210	1222	539	1202	-0.67
cfft	3835	1853	3945	3843	1853	3802	-3.76
Total	14750	5807	15741	14951	5807	15370	-2.41
%change	–	–	–	1.36	0.00	-2.36	–

the standpoint of LUTs can be misleading if the target architecture supports the traditional LE model. Full-width solutions can be encouraged to form more LEs through the selection of alternative cuts with smaller node cut size, as described in Section 5.2. This has the effect of producing more $(K-1)$-LUTs, which are more likely to comply with the LE input constraints expressed in Lemma 5.4.1(i,ii).

6.2 Hierarchical Clustering with HierARC

Inspiration for alternative clustering techniques can be drawn from other disciplines such as bioinformatics. DNA microarrays are used to measure cellular gene expression in response to a stimulus. While much of the underlying biology with which they are concerned is not germane to a discussion of FPGA CAD, the general problem and solution of microarray clustering shares many similarities to FPGA clustering. Microarrays contain thousands of data points, each corresponding to a gene; some genes share a response pattern, while others are unrelated. Microarray clustering groups genes together based on similar expression patterns, in much the same way FPGA clustering seeks to group LEs together based on similar resource requirements.

HierARC, presented in [39], is a **Hier**archical **A**gglomerative **R**econfigurable fabric **C**lustering technique that is polynomial in run-time, deterministic, scalable with regard to scoring metrics, and capable of easily considering multiple resource constraints. HierARC deviates from the cluster-seed model by incorporating a *bottom-up* approach that merges the highest gain cluster pair each iteration. It avoids post-application of resource constraints by considering them each iteration. HierARC also addresses chain clustering, which is important because most commercial architectures offer them as a resource, despite the lack of support in currently available FPGA CAD tools.

Hierarchical clustering, the most commonly used microarray clustering technique, is readily employed by FPGA CAD. Agglomerative clustering comes in a variety of flavors, but most algorithms are derived from single-link, complete-link, and minimum-variance algorithms [44]. Though they differ in goals, the basic concept of each of these algorithms is the same: to combine clusters that are heavily correlated using a distance function. The basic hierarchical clustering algorithm works by merging clusters. Initially, all LEs occupy their own cluster, corresponding to the worst case clustering solution. Iteratively, each cluster is compared to all others, and the best cluster pair is merged into one. The new cluster's correlation to all others is then updated according to employed scoring functions. The process repeats until no valid cluster pair is found. Potential cluster pairs are also subject to resource constraints, including inputs per cluster (I), clocks per cluster (M), LEs per cluster (N), and all chain constraints. Algorithm 6.2 presents HierARC, which has been implemented as an extension to VPR [12].

Algorithm 5 HierARC Clustering Algorithm

```
1:  procedure HIERARC($G$)                              ▷ Input Boolean network $G = (V, E)$
2:      $U = \emptyset$                                    ▷ Initialize set of clusters
3:      for $v_i \in V(G)$ do
4:          $C_i = v_i; U = U \cup C_i$                     ▷ Create cluster for each LE and add to $U$
5:      end for
6:      for $\forall C_i, C_j \in U$ do                    ▷ Initialize gain matrix
7:          $gain(C_{ij}) = \sqrt{\sum_{k=1}^{m}(P_k(ij))^2}$   ▷ Compute Euclidian gain
8:      end for
9:      $max_{ij} = max\{P(Cij)\}$                         ▷ Identify cluster pair with the highest gain
10:     while $max_{ij} > 0$ do                            ▷ While there are still valid cluster pairings
11:         $C_i = C_i \cup C_j$;                          ▷ Create $C_{ij}$ by merging $C_j$ into $C_i$
12:         for $\forall C_i, C_j \in U$ do                ▷ Update gain matrix
13:             $gain(C_{ij}) = \sqrt{\sum_{k=1}^{m}(P_k(C_{ij}))^2}$   ▷ Recompute Euclidian gain
14:         end for
15:         $max_{ij} = max\{P(Cij)\}$                     ▷ Identify cluster pair with the highest gain
16:     end while
17: end procedure
```

One advantage of agglomerative clustering is that it removes the undue priority placed on the cluster seed, and the danger of choosing incorrectly. The bottom-up nature also avoids having to partition the network and invoke depopulation to comply with resource constraints. Instead, resource constraints can be considered on the fly. Each iteration requires a $O(n^2)$ search for the maximum gain cluster pair, among n LEs. After the merging of two clusters is performed, the gain of each affected cluster pair, $O(n^2)$, is re-computed. The algorithm executes for up to n iterations, giving HierARC a complete run-time of $O(n^3)$.

In the realm of data clustering, random variables are measured for behavior correlation. LEs in an FPGA are not correlated in the same way, thus requiring some modification to the basic scheme. HierARC uses Euclidian distance to combine multiple performance metrics and choose the best pair of clusters to merge. Given m performance metrics $\{P_0, ..., P_m\}$ and clusters $\{C_g, C_h, C_i, C_j\}$, the Euclidian distance between pairs C_{ij} and C_{gh} is given by Equation 6.1. However, since the goal is to select the maximum gain pair, C_{ij} must be compared to a pair with 0 gain. If $P_k(C_{gh}) = 0$ is a no gain cluster for all $1 \leq k \leq m$, the distance between C_{gh} and C_{ij} is given by Equation 6.2. Each metric is normalized to its maximum value. Combining Euclidian gain with normalization allows HierARC to conveniently scale with additional metrics.

$$dist(C_{ij}, C_{gh}) = \sqrt{\sum_{k=1}^{m} (P_k(C_{ij}) - P_k(C_{gh}))^2} \tag{6.1}$$

$$gain(C_{ij}) = \sqrt{\sum_{k=1}^{m} \left(\frac{P_k(C_{ij})}{max_{P_k}} \right)^2} \tag{6.2}$$

A scoring function that facilitates clustering with an eye toward routing complexity is presented in [15]. It judges routing cost, R_{cost}, via Equation 6.3 and Equation 6.4, where x is a net with multiple terminals, $\alpha(x)$ is its routability weight, and the terminals of x are denoted $pins(x)$. The pins of a net which are visible from the general routing array (external) are denoted $ext(x)$, while the locally implemented (internal) portion of a net is $int(x)$. This nomenclature is abused for clusters, such that $ext(C_i)$ refers to the external pins of cluster C_i. Studies in routability have found that $\alpha(x)$ should increase with $|pins(x)|$ by a non-linear factor [16]. In this manner, nets with more terminals are regarded as more difficult to route, but when the number of pins of a net becomes sufficiently large, there is no discernible change in difficulty as net size continues to increase. Equation 6.3 captures the condition of a floating net, i.e. one that possess no general routing sink, by giving nets with fewer than 2 terminals a routing weight of 0 because they have no impact on general routing cost.

$$\alpha(x) = \begin{cases} 2 - \frac{1}{|ext(x)|} & |ext(x)| \geq 2 \\ 0 & otherwise \end{cases} \tag{6.3}$$

$$R_{cost} = \sum_{x \in Nets} \alpha(x) \tag{6.4}$$

HierARC uses routing cost as a metric and judges its gain explicitly, through the selection of the best merged pair, instead of implicitly through point-based resource or attraction techniques. It explicitly identifies the best possible next state of the network by selecting the cluster pair offering the greatest routing cost reduction. Given an arbitrary cluster pair, C_{ij}, there exists a set of external nets common to both clusters, $ext(C_i) \cap ext(C_j)$. If C_{ij} results in the violation of I, M, N, or a chain constraint, then $gain(C_{ij}) = -\infty$, i.e. the clusters are incompatible. If $ext(C_i) \cap ext(C_j) = \emptyset$, then no nets would gain from the merging of the clusters, and $gain(C_{ij}) = 0$, i.e. not incompatible but undesirable.

The total change in R_{cost} is reflected directly by Equation 6.5. Here, net $x \in ext(C_i) \cap ext(C_j)$ is transformed to net $x' \in ext(C_{ij})$ by the subtraction of a pin from $ext(C_j)$ and the addition of a pin to $int(C_{ij})$. However, if $ext(x') = \emptyset$, Equation 6.3 yields a 0, resulting in a $\alpha(x) - 0 = 1.5$ contribution to $gain(C_i, C_j)$. This value that far exceeds that of a 3 pin external net, possessing a score $\alpha(x) - \alpha(x') = 0.167$. This exceedingly simple gain computation enables HierARC to more effectively produce clustering solutions with minimal routing cost.

$$P_{route\_cost}(C_{ij}) = \sum_{x \in ext(C_i) \cap ext(C_j)} \alpha(x) - \alpha(x') \tag{6.5}$$

There are many clustering tools in literature, but few are available publicly or comprehensively discuss FPGA architectures used to assess solution effectiveness. However, comparison

Table 6.2 MCNC Clustering, $K = 4$, $N = 8$, $I = 18$

	Clusters			Channel Width, W			Wire (segments)			Area (10^6 trans)			Delay (ns)		
Circuit	V	TV	Hi	V	TV	Hi	V	TV	Hi	V	TV	Hi	V	TV	Hi
alu4	198	193	192	48.7	35.3	32.1	13622	9154	8179	2.80	1.82	1.66	112.92	80.33	77.94
apex2	241	240	242	51.5	48.4	41.0	17308	16065	14027	3.33	3.17	2.74	95.28	85.63	84.11
apex4	163	165	166	46.9	50.2	45.2	10340	10646	9900	2.04	2.17	2.00	90.30	84.34	84.79
bigkey	214	214	226	37.0	24.2	13.4	14689	11670	9080	6.83	4.61	2.77	80.90	87.27	92.06
clma	1056	1055	1051	74.8	63.6	47.3	92934	81496	60476	20.05	17.30	13.00	379.63	351.31	387.73
des	204	200	200	32.6	21.0	16.9	18031	15862	13445	8.47	5.71	4.72	179.49	221.71	170.50
diffeq	188	189	199	29.7	28.6	19.1	7740	7081	5041	1.54	1.49	1.20	68.19	58.11	70.60
dsip	198	172	226	38.5	21.1	11.9	15536	11165	8422	7.11	4.11	2.52	86.51	74.64	86.48
elliptic	453	454	497	59.6	51.8	37.0	30709	27997	18091	7.16	6.31	5.00	132.96	160.00	154.04
ex101	595	601	595	56.2	57.2	41.3	42364	40160	31967	8.76	9.00	6.75	220.91	253.19	204.45
ex5p	136	138	139	44.8	46.7	40.8	8722	9129	8190	1.69	1.76	1.56	87.38	79.01	72.26
frisc	448	446	454	57.6	52.6	40.6	31632	28384	22407	7.01	6.50	5.10	175.96	179.22	166.07
misex3	178	179	177	43.7	39.5	36.7	11131	10055	8884	2.21	2.05	1.91	84.86	93.12	93.24
pdc	593	582	586	83.4	73.7	66.8	58683	50531	45110	12.83	11.46	10.36	231.21	259.28	226.93
s298	246	243	242	45.2	32.2	25.9	13592	8820	7308	2.95	2.15	1.76	135.75	125.71	125.33
s38417	803	802	820	44.4	37.7	25.1	42035	35689	23266	9.27	8.01	5.48	192.07	161.57	151.91
s38584.1	806	806	889	38.2	37.2	25.1	36178	35994	19947	8.13	8.00	5.93	200.36	179.03	315.40
seq	223	221	221	51.7	47.9	40.4	15639	13762	12326	2.96	2.78	2.40	93.52	79.37	85.20
spla	476	469	470	74.4	58.3	54.5	43529	33523	29143	9.00	7.03	6.63	228.66	206.71	167.20
tseng	132	133	141	31.2	27.7	16.6	5891	5168	3023	1.20	1.08	0.69	54.59	46.01	50.30
Total	7551	7502	7733	990.1	854.9	677.7	530304	462351	358232	125.34	106.51	84.20	2931.44	2865.55	2866.54
%change	2.41	3.08	–	-31.55	-20.73	–	-32.45	-22.52	–	-32.83	-20.95	–	-2.21	0.03	–

against T-Vpack using VPR [12] PNR results is ubiquitous. Therefore, a baseline of performance can be established by comparing HierARC against T-Vapck using an FPGA architecture similar to that used in [15]. Table 6.2 presents clusters, routing channel width (W), total routing wire segments, transistor area with buffer sharing, and critical path delay (T_{crit}). The FPGA device under test uses single-length segments, $Fc_{input} = Fc_{output} = 1$, with net and timing driven place and route. To get an accurate picture of performance, 20 independent PNR experiments of each MCNC circuit have been performed, and normalized to the HierARC result. This more closely ascertains the expected result obtained in a real-world design process, wherein a designer usually accepts the first PNR solution produced by a CAD tool.

Results indicate that HierARC performs substantially better in most categories than Vpack or T-Vpack. HierARC uses slightly more clusters, although number of clusters is a misleading metric, as most clustering tools tend toward full cluster utilization though research has shown that underutilization of clusters can actually improve design routability [71]. HierARC achieves

Table 6.3 Approximate Comparison to Published Results

	[58]		RT-Pack[47]		[47]		iRAC+iRAP[67]		
	W	T_{crit}	W	Wire	W	Wire	W	Wire	Area
%Other	-15.38	-0.85	-5.56	-4.25	-19.44	-17.77	-23.40	-19.98	-24.26
%HierARC	-20.73	0.03	-20.73	-22.52	-20.73	-22.52	-20.73	-22.52	-20.95

average channel width reductions of -32% and -21%, total routed wire length reductions of -33% and -23%, and transistor area reductions of -33% and -21%. The average critical path delay change produced by HierARC is negligible at -2.2% and +0.03%. This is a byproduct of the 1-D routing gain function currently employed. HierARC is easily extended to multiple performance metrics and work is currently being performed to consider timing and power consumption in addition to routing cost.

Table 6.3 creates an approximate context for HierARC performance. Although tool availability and exact experimental parameters are often limited, published results indicate that HierARC performance meets or exceeds other tools. The results in Table 6.3 are only to be used as an approximate comparison of performance because of the high variability in target FPGA architectures, PNR settings, and metrics. In each case, performance is measured in percent change relative to T-Vpack. It should be noted that all of the results in Table 6.3 are based on $K = 4$, $N = 8$, and in most cases $I = 18$ ([47] uses $I = 32$). Additionally, RT-Pack results obtained from [47] (thus using $I = 32$) concur with those in [15] and are included because they present wire length.

Clustering chains presents challenges because LEs in the same chain typically have very low commonality to each other. For example, a 32-bit adder uses 64 operand bits that are often independent and travel in parallel via a datapath through the circuit. Chains largely lack the freedom to choose other members of their cluster, but can be segmented to maximize what commonality they do possess. Segmenting is the process of partitioning a chain into pieces which are completely contained within clusters, subject to the resource constraints of the architecture. Each LE that is part of a chain must comply with the following rules:

1. Each cluster can only contain one chain head segment.

2. Each cluster can only contain one chain tail segment.
3. Non-contiguous members of the same chain are not allowed in the same cluster.
4. If preempting the merging of chains, each cluster can only contain either a head or a tail.

The chain rules are designed to observe port availability constraints and can be modified accordingly for non-traditional clusters, such as those possessing a multiple-port chain. Because there is only one *cin*/*cout* port pair available, each cluster can only contain one head and one tail unless they belong to the same chain (a condition corresponding to an intra-cluster chain). Second, non-contiguous LEs in the same chain are not allowed to reside in the same cluster to prevent chain interruption. Finally, if preempting the merging of chains, clusters can contain either a head or a tail from separate chains, but not both. If a head from one chain and a tail from another are allowed to join the same cluster, a de facto chain is created by the merging of two chains, as in Figure 6.2(d) where chains A and B have merged. In general, increasing the number of cluster chains decreases the freedom of the PNR tool.

Chains are handled by HierARC in one of three ways: cluster chains from head to tail individually before free LEs subject only to resource constraints, cluster each chain individually before free LEs according to gain, or allow chain LEs to be clustered at the same time as logic LEs. Chains reduce the number of possible clustering solutions, but can be segmented into clusters by gain to minimize their cost and observe cluster constraints. HierARC has the option of applying the default segmentation of chains (head to tail), or by allowing them to be decided by gain. HierARC does not require the formation of contiguous segments, i.e. segments are allowed to contain empty LEs, those which simply pass values along the chain. Clustering chains has the following concerns:

1. Enable the formation of intermediate segments that comply with traditional values of I.
2. Limit the use of empty LEs.
3. Use chains with minimal segments when possible.
4. Avoid merging cluster chains whenever possible.

Clustering chains has implications on the value I. As has been previously discussed, every

LE input does not have to access the general routing array, as per Equation 6.9. Instead, local routing allows some inputs to be completely routed internally, with 98% logic utilization achievable using Equation 6.6 [4]. For chains, the value of I is strongly dependent on the maximum number of inputs required by a chain. Typically, all of the chain inputs are external to the cluster because the LEs in a chain typically possess very few nets in common. Two types of chains can be formed, full-width and sub-width (e.g. arithmetic) [38], and each have a different effect on I.

The worst case value of I for a chain segment with N full-width LEs occurs when the head of a chain requires K inputs, and the subsequent $N-1$ LEs require $K-1$ inputs (Equation 6.8). Likewise, for sub-width LEs, the head of a chain requires $K-1$ inputs, and the subsequent $N-1$ LEs require $K-2$ inputs (Equation 6.7). For architectures to accommodate all chains in a default clustering, they have to subscribe to either Equation 6.8 or Equation 6.7, regardless of Equation 6.6.

$$I_{norm} = \frac{K}{2} \cdot (N+1) \tag{6.6}$$

$$I_{sub} = (K-2) \cdot (N-1) + K - 1 \tag{6.7}$$

$$I_{full} = (K-1) \cdot (N-1) + K \tag{6.8}$$

$$I_{all} = K \cdot N \tag{6.9}$$

Empty LEs can be used to give chains the latitude to use clusters designed with Equation 6.6 in mind. In most FPGAs, the chain logic is highly optimized, and contributes negligible wire and logic delay to a design. To allow chains to comply with an input-limited cluster, LUTs can be configured as buffers and simply pass chain values. While this may increase the delay of the design, a design that uses a higher value of I will have a more complex routing array, as each input pin on a cluster generally requires 30 or more support general routing wires [43]. Because empty LEs have no inputs beyond the chain logic, they contribute a delay commensurate to the chain logic, instead the entire K-LUT.

The chain in Figure 6.2(a) is contiguous and it occupies the minimum number of clusters,

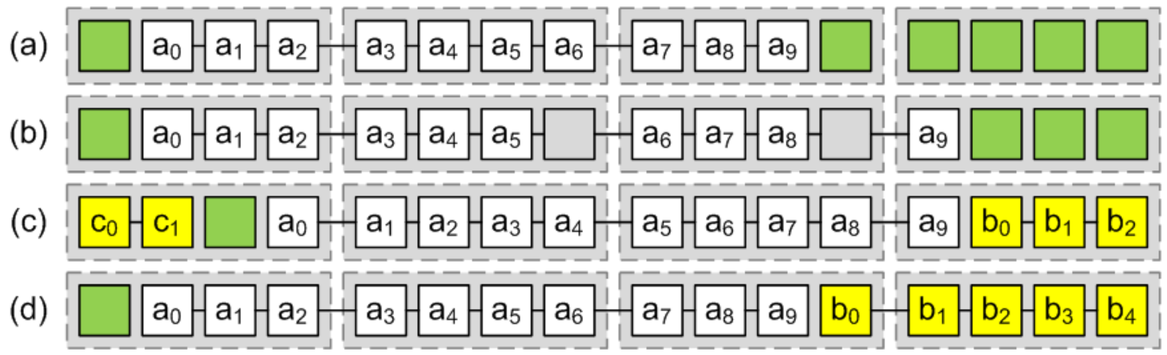


Figure 6.2 Clustering a chain for $L = 10$, $N = 4$.

but its head does not necessarily occupy the entire first cluster. A chain that is clustered contiguously has $\lceil \frac{L}{N} \rceil$ segments. Chain A in Figure 6.2(b) has two empty LEs, its head does not occupy the entire first cluster, and it occupies an extra cluster. This situation could arise if I limits the number of chain members per cluster or gain dictates an alternative solution.

Allowing chain LEs to be clustered concurrently with logic LEs potentially increases the number of intra-cluster chains. The reason for this is that each LE within a chain has very low commonality to adjacent chain LEs and much higher commonality to other LEs. Accordingly, when chain LEs which are part of the head or tail of chain are clustered, they tend to join non-chain LEs and result in solutions requiring more than the minimum number of segments. Figures. 6.2(a,c) depict the effect of clustering chains first versus clustering them concurrently with non-chain LEs. Aside from tending to increase intra-cluster chains, concurrent clustering also increases the prevalence of empty LEs.

To lower the occurrence of chain clusterings with non-minimal segments, HierARC includes a user option to cluster chains first, and independent from one another. Each LE chain in the design is clustered according to the HierARC algorithm, with the exception that only LEs within the same chain are considered, and the gain of each chain LE relative to all other LEs residing outside its chain is $-\infty$.

Another method to encourage the formation of minimal chains is the addition of a chain bias factor to Equation 6.5, resulting in Equation 6.10. This tie breaking system is similarly structured to the T-Vpack timing-based scoring function [12]. It employs a scaling factor,

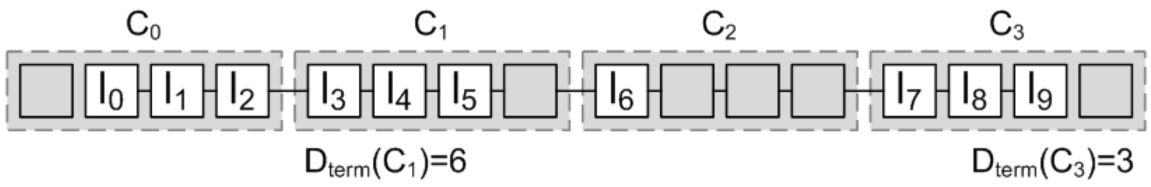


Figure 6.3 Tie breaking a chain for $L = 10$, $N = 4$.

$\epsilon \approx .01$, that uses the contiguous distance from terminus, D_{term} as a tie breaker. Distance from terminus is computed as the maximum number of LEs which the merged cluster will connect to either the head or tail of the chain. A terminus is not simply the cluster containing either end of a chain, but also the clusters which extends it; if the cluster containing an end cannot be merged with the next consecutive cluster in the chain, that cluster is also considered a terminus. Figure 6.3 depicts this as $D_{term}(C_1, C_2) = 6$, where C_0 and C_1 are assumed not compatible due to resource constraints. Because C_0 and C_1 are adjacent, but not compatible, C_1 is regarded as a terminus, and $\{a_0, ..., a_5\}$ is 6 LEs in length. In this situation, where $D_{term}(C_1, C_2) > D_{term}(C_1, C_2)$, the preferred solution is that which creates the longest contiguous cluster chain. This encourages minimum chain segments and fewer empty LEs.

$$P_{chain}(C_{ij}) = P_{route\_cost}(C_{ij}) + \epsilon \cdot D_{term} \tag{6.10}$$

Table 6.4 shows results for clustering chains head first ($head1^{st}$), first ($chain1^{st}$), and at the same time ($during$) as arbitrary LEs for designs from [62], and normalized to head first. Allowing non-traditional clustering produces a 11.6%-26.1% increase in the number of clusters each chain spans, and average chain length by 5-11 LEs. Clustering head first is by far the best solution when considering only chain lenght and empty LEs. When $K = 4$, the number of inputs required to support a sub-width chain is one less than the normal cluster inputs, $I_{sub} = 17 < 18 = I_{norm}$. This indicates that the clustering solution should accommodate chains without empty LEs and with minimum number of segments. This is not reflected in Table 6.4, because the scoring function used by HierARC has not deemed it to be the most routing cost advantageous solution and I_{norm} does not accommodate the sophisticaed D flip-flops (DFFs) present in real-world designs which contain asynchronous and synchronous FF

Table 6.4 OpenCores Clustering, $K = 4$, $N = 8$, $I = 18$

	Chains	Segments			Ave. LE Len.			Empty LEs		
Circuit	-	Head 1^{st}	Chain 1^{st}	During	Head 1^{st}	Chain 1^{st}	During	Head 1^{st}	Chain 1^{st}	During
cfft	61	264	279	306	30.0	31.5	32.3	77	166	220
xtea	10	40	40	49	30.5	31.2	32.4	6	13	25
usb	36	41	46	80	7.9	7.9	9.1	0	0	44
mult3x3	12	42	48	54	22.3	24.3	27.3	0	24	60
jpeg	85	239	310	325	20.2	22.8	24.0	0	224	321
sha512	20	146	146	177	58.2	58.2	61.1	0	0	58
fir	24	72	73	73	17.5	17.5	17.7	0	0	5
Total	248	844	942	1064	24.1	25.5	26.8	83	427	733
%change	–	–	11.6	26.1	–	5.7	10.9	–	414.5	783.1

controls, which increase I. More comprehensive PNR experiments are necessary to justify the increase in empty LEs and segments due to deciding chain segmentation by gain, and are left as future work. However, the HierARC technique does accommodate chains regardless of wheteher or not I accommodates contiguous chains.

6.3 Summary

ChainPack is presented as a method for reducing technology map LE consumption by using chains. Chains can be viewed not only as a single-fanout high-performance interconnect structure, but also as a dual-fanout connection between adjacent LUTs. That being said, it can be used to identify and implement two LUTs that comply with the exclusivity constraint within the same LE, thus reducing the total number of LEs in the network. Sample ChainPack results for $K = 5$ indicate an average 2.36%, maximum of 9.5%, LE consumption savings over an unpacked relaxed solution.

HierARC [39], a Hierarchical Agglomerative Reconfigurable fabric Clustering tool, is deterministic, has a polynomial run-time, scalable with respect to incorporating performance metrics through Euclidian distance, and can easily accommodate resource constraints without post-processing techniques such as cluster depopulation. Using only routing cost as performance metric, results indicate reductions in channel width (-21%), wire length (-23%), and transistor area (-22%) with minimal critical path change (+0.03%) relative to T-Vpack. The neutral improvement in delay is due to its current lack or a timing metric. However, HierARC easily supports the inclusion of metrics such as timing, power consumption, and fault tolerance.

As evidence of its ability to accommodate clustering constraints during execution, the problem of clustering chains has been presented. No published solutions to this problem are evident in literature, as the popular approach seems to be acceptance of a default clustering wherein the head of a sub-width chain occupies the first LE in a cluster, and all consecutive LEs are clustered contiguously. Such an approach does not take into account the performance impact of chains, allow for generic logic chains [38] to be clustered effectively, or subject chains to resource constraints. In effect, chains are allowed to dictate architecture, rather than the architecture dictating chain clustering solutions. This work presents an initial foray into chain clustering that indicates HierARC is able to produce chain clusterings with an average chain length increase of 5-11 LEs.

Using T-Vpack as a common point of reference for published clustering tools, HierARC produces results that meet or exceed those of other tools. HierARC's performance, coupled with its flexibility, determinism, polynomial run-time, and on-the-fly resource constraint accounting, makes it a viable solution to clustering for FPGAs. Together, ChainPack and HierARC produce technology mapped logic chain solutions with reduced area and higher routability.

CHAPTER 7. CHAINMAP FULL DESIGN FLOW EXPERIMENTS

To test the performance of ChainMap, a complete FPGA design flow is necessary so that the effects of using logic chains can be fully ascertained. Unfortunately, there is no academic design flow for FPGAs that supports arithmetic chains, let alone generic logic chains. Conversely, commercial tools are highly proprietary in nature. The Xilinx ISE design flow is difficult to use directly or with third-party tools. The Altera Quartus II flow is relatively open, allowing academic users to substitute for various parts of the flow including synthesis and technology mapping, but its target architecture is restricted to Altera products. The Stratix architecture is the only one capable of supporting any type of ChainMap flow, in particular sub-width logic chains for $K = 4$.

The standard academic design flow consists of SIS [66] and RASP [24] technology mapping or ABC [59] synthesis/mapping, T-Vpack clustering, and VPR place and route [12]. All in all, these are useful open source tools, but none of them support chains internally. Therefore, they need to be augmented to support logic chains. A Quartus II VQM netlist will be used to identify macro generated arithmetic chains and coarse grained components from HDL and designate them to SIS. Using the reuse cell design of Chapter 3, the performance of ChainMap will be presented across a variety of experiments and metrics:

- Publicly available HDL-based benchmarks from OpenCores [62].
- Performance assessment using metrics including depth, critical path clock frequency, area, routing utilization, and area-delay product.
- LUT sizes of 4 to 6 inputs to encompass all commercial FPGA logic widths.
- Insertion of arithmetic chains before technology mapping, after mapping, and complete disregard of HDL after elaboration.

- Least critical branch trimming and global least critical relaxation.
- Path and net timing driven routing metrics.

SIS and RASP have been chosen because they support FlowMap, which is the inspiration for ChainMap. T-Vpack will be abandoned in favor of HierARC, as Section 6.2 has shown it can produce solutions with more than a 20% improvement in routability while also accepting chains and sophisticated FFs. VPR will be modified to handle chains and FFs. The range of testing conditions provides ample opportunity to test ChainMap designs for a variety of architectures and CAD parameters.

7.1 Testing Methodology and Architecture Description

The testing methodology will be the same as in Section 5.6, except that the entire design flow will be tested. This includes the addition of least critical branch trimming and global least critical relaxation techniques, clustering with HierARC (Section 6.2), and a placement and routing with a modified version of VPR. In this manner, a full accounting can be taken of using ChainMap to decide chains versus traditional HDL-based arithmetic chains. Figure 5.5 depicts the augmented design flow; the elaboration, synthesis, and initial technology mapping stages remain identical to Section 5.6. After technology mapping, the ChainMap flows are relaxed according to least critical branch trimming and global least critical relaxation techniques, followed by ChainPack. FlowPack is then applied to all non-chain Boolean nodes. All flows are then clustered with HierARC and placed and routed by VPR using two different architectures, fixed-size and scaled.

A cluster size of $N = 8$ is used because it is the most common amongst commercially available architectures. Clusters with 8 LEs possess 23% less delay and use 14% less area relative to clusters with $N = 1$, representing the best area-delay balance among clusters sizes in the range of $N = [3, 20]$ [57]. The number of inputs per cluster is $I_{norm} = \frac{K}{2} \cdot (N + 1)$, with the exception that each cluster will be granted four additional inputs to provide capacity for a subset of the FF control signals (*aload*, *sload*, *dload*, *aclr*, *sclr*, *clken*). The four additional inputs allow for designs to contain one load and its data input (synchronous or asynchronous),

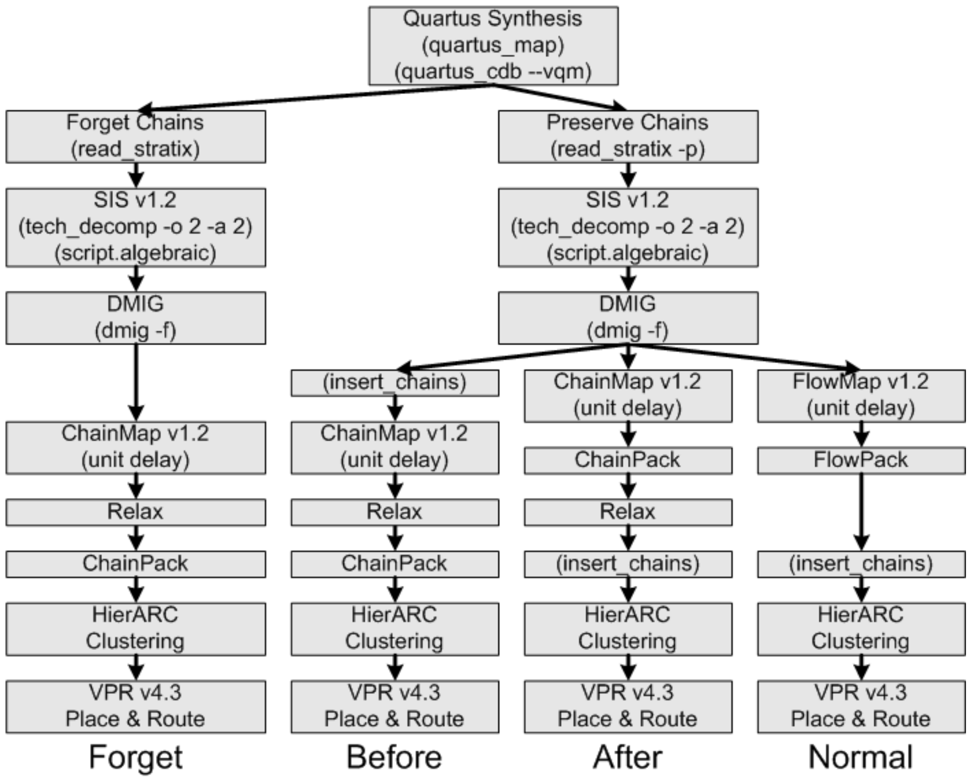


Figure 7.1 Experimental Design Flows

one clear (synchronous or asynchronous), and the clock enable. FFs using all six asynchronous and synchronous control signals are rare, so they must compete for cluster inputs with standard data inputs. The MCNC benchmarks upon which I_{norm} is established in [4] do not contain sophisticated FFs and thus capacity for them must be added to ensure reasonable clustering solutions. Chains will be clustered according to the default head-first technique, wherein they are mapped from head to tail individually and are only subject to resource constraints.

The routing architecture will be fashioned after the Stratix II, which offers routing segments of 1, 4, 16, and 24 clusters. VPR features a binary-search parameterizable architecture that enables researchers to perform experiments which measure solution performance with the minimum amount of resources available. The scaled architecture tailors the cluster array size to the minimum required to accommodate I/O pads or clusters. Placement occurs as normal, but routing is done according to a binary search for minimum channel width. The minimum channel width is the minimum number of tracks which are necessary to successfully route the design. For the fixed architecture, the array size is limited 34 x 34 clusters, which is the minimum size chip necessary to accommodate the maximum number of I/O pads and clusters found in the benchmark set (*des3_area* I/O pads). The routing channel width will be set to 60 tracks per channel to accommodate all minimum channel width requirements. Because the array size will be at most 34 x 34 clusters, segment lengths are scaled to segments of 1,2,4, and a long line spanning the entire chip. Using the iFAR design repository [49], a 65nm based architecture (the feature size of Stratix II) has been constructed that is characterized by the timing parameters in Appendix 8.4.

7.2 Tool Development

To test complete designs that contain arithmetic chain constructs, the SIS synthesis tool requires the ability to input arithmetic chains and support them within its internal netlist format. Furthermore, the ChainMap and relaxation algorithms have been implemented in SIS/RASP along with improvements for FFs. The additions/alterations to the SIS command library are as follows:

- $read\_stratix[-p < file.vqm >]$
- $read\_blif$, $write\_blif$ - The input and output netlist formats are augmented to describe chains and LEs. Additionally, the SIS latch has been granted full FF functionality.
- $insert\_chains$, $remove\_chains$ - Chains can be removed and inserted into the netlist at any time.
- $flowmap[-C < K >, -A < g|b|l >]$ - The invocation of ChainMap using K-width chains and duplication algorithm **gl**obal least critical relax, least critical **b**ranch trimming, or shallowest **l**ogic depth branch trimming.

QUIP allows designers to modify steps of the FPGA design flow through providing access to the VQM or BLIF netlist products of Quartus II synthesis and technology mapping. To implement an HDL-aware netlist in SIS internal representation, QUIP will be used to elaborate the HDL. A VQM netlist parser has been created for SIS that will take the technology map output of Quartus and input it to SIS internal netlist format. The VQM parser is invoked using the *read_stratix* command. Arithmetic chains can be alternately preserved, $read\_stratix - p$, or allowed to become part of the logic network, *read_stratix*.

Each LE in the VQM netlist, with ports as defined by Figure 7.2, is implemented in SIS pursuant to its Stratix functionality. LEs in normal operation mode are implemented as simple logic nodes with their function defined by an *lut_mask*. Each SIS node has the equivalent of a *combout* port. Arithmetic LEs have to be handled such that their functionality and mapping are both preserved. The arithmetic mode Stratix LE, in Figure 7.3(a), is implemented in the SIS internal netlist representation as the pair of $(K - 1)$-LUTs shown in Figure 7.3(b), one for the *cout* computation and one for *sum*. Dynamic add and subtract, *inverta*, is explicitly implemented as an XOR gate combining *dataa* and *inverta*, the output of which serves as an input to the LUT. The combinational mode of the LE is implemented as in Figure 7.3(c).

If chains are to be preserved with $read\_stratix - p$, any LE making use of either its *cin* or *cout* port is considered part of an arithmetic chain and is partitioned into a separate network. Arithmetic chain partitioning in SIS is depicted in Figure 7.4(a) and is also invoked using the command *remove_chains*. All inputs to the chain are output from the logic network with

```
stratix_lcell <lcell_name>
(
  .clk(<clock source>),
  .dataa(<data_a source>),
  .datab(<data_b source>),
  .datac(<data_c source>),
  .datad(<data_d source>),
  .aclr(<asynchronous clear source>),
  .aload(<asynchronous load source>),
  .sclr(<synchronous clear source>),
  .sload(<synchronous load source>),
  .ena(<clock enable source>),
  .cin(<carry in source>),
  .inverta(<inverts .dataa into the lut>),
  .combout(<combinational output>),
  .regout(<registered output>),
  .cout(<carry output>)
);
defparam <lcell_name>.operation_mode = <operation mode>;
defparam <lcell_name>.synch_mode = <synchronous usage mode>;
defparam <lcell_name>.sum_lutc_input = <sum lut input choice>;
defparam <lcell_name>.lut_mask = <lut mask>;
```

Figure 7.2   Stratix cell primitive.

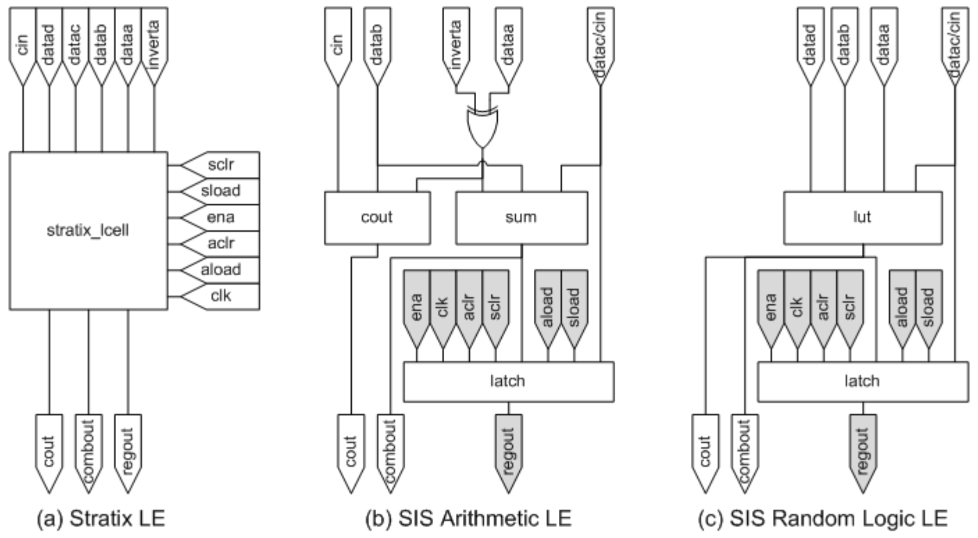
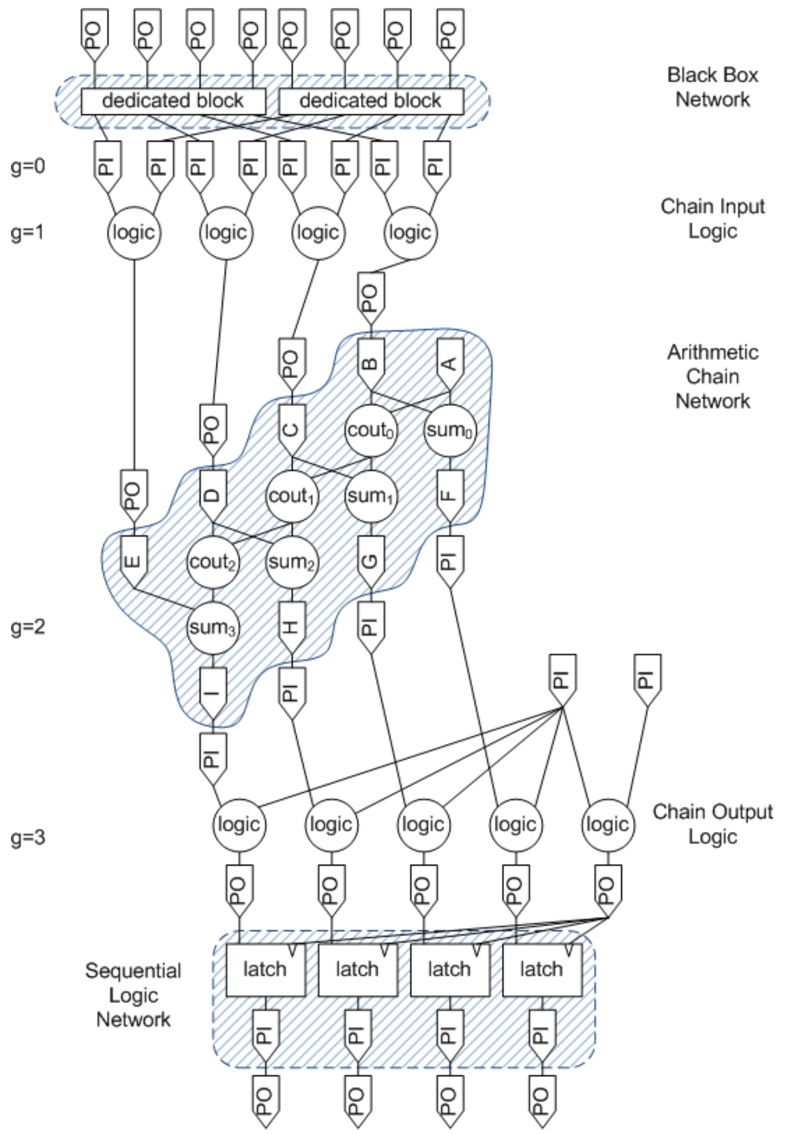


Figure 7.3   (a) Stratix primitive, (b) SIS arithmetic LE, (c) SIS combinational LE

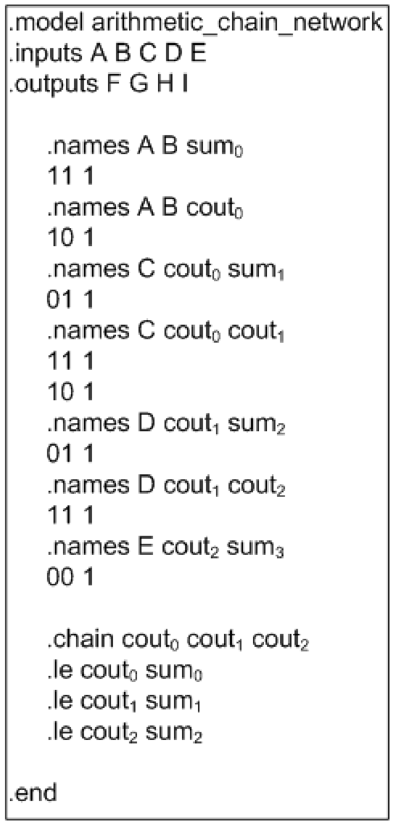
POs and input to the chain network with PIs. Likewise all external outputs of the chain are output from the chain network with POs and input to the logic network with PIs. In this manner, the logic network can undergo technology decomposition, SIS synthesis, and FlowMap or ChainMap technology mapping, while the chain network is preserved. The chain and logic networks can be recombined at any point in the flow with *insert_chains*, yielding a unified netlist.

Another reason for the integrated Quartus/SIS design flow is that more complex designs also contain memory, multipliers, register controls, and clock synthesis structures for implementation in dedicated or specialized FPGA components: RAM (RAM) modules, digital clock managers (DCMs), and carry chains. RAM, DCMs, and dedicated multipliers are extracted from the VQM netlist and treated as black box I/O modules to the system that are interfaced with PIs/POs. Figure 7.4(a) shows how black box components are extracted from SIS and how arithmetic chains are partitioned from the logic network. To more closely model the capabilities of commercial architectures, the SIS latch has been upgraded to incorporate full flip-flop functionality, including asynchronous load/clear (*aload*, *aclr*), synchronous load/clear (*sload*, *sclr*), and clock enable (*ena*). Valid SIS latch behaviors are rising edge, falling edge, active high, or active low.

As per Definition 5.3.3, a chain can be described as a set of contiguous nodes possessing the same routing depth. For a netlist that adheres to the exclusivity constraint, synthesis and technology mapping chains can be designated to the clustering and PNR engines using a modified BLIF that specifies the chain's "backbone" and ancillary LE mates. The backbone consists of all depth-increasing *cout* nodes identified through HDL or ChainMap. Each LE consists of a pair of nodes, one of which must be the *cout* node present in the backbone, and the other its accompanying *sum* node. For LEs operating in $(K-1)$-LUT mode (Figure 3.1), $cout \neq sum$. If an LE is operating in K-LUT mode, $sum = cout$. Note that the terms *cout* and *sum* only refer to a node's connectivity with respect to LEs and chains, and do not imply its Boolean function is the traditional sum or carry out of a full adder.



(a) SIS Chain



.model arithmetic_chain_network
.inputs A B C D E
.outputs F G H I

.names A B sum_0
11 1
.names A B $cout_0$
10 1
.names C $cout_0$ sum_1
01 1
.names C $cout_0$ $cout_1$
11 1
10 1
.names D $cout_1$ sum_2
01 1
.names D $cout_1$ $cout_2$
11 1
.names E $cout_2$ sum_3
00 1

.chain $cout_0$ $cout_1$ $cout_2$
.le $cout_0$ sum_0
.le $cout_1$ sum_1
.le $cout_2$ sum_2

.end

(b) BLIF Description

Figure 7.4 SIS arithmetic chain with black box module partitioning and BLIF representation.

The chain backbone of carry propagation is modeled in BLIF format through the ".chain" and ".le" descriptions. Figure 7.4(b) shows an example BLIF representation of the adjoining arithmetic chain network. The chain description contains all of the *cout* nodes in the chain from head to tail. To model LE associations, the *.le* primitives map each node in the chain definition to a *sum* node. The chain/LE specifiers do not change how the nodes of the chain are modeled by SIS, but simply provide ancillary connectivity and grouping information. This BLIF definition also shows how a chain network is protected from synthesis by *remove_chains*.

After SIS synthesis and ChainMap/FlowMap technology mapping, the network is clustered, placed, and routed. HierARC (Section 6.2) is designed to cluster arbitrary LEs and chains through cluster merging, as outlined in Section 6.2. VPR has been augmented to read the modified BLIF netlist and represent chains and FFs in its ".net" netlist format and has been augmented to handle chains according to the modifications outlined in Section 2.2.4 to emulate commercial FPGA architectures and intelligently handle chains. Full FF functionality is included in HierARC and VPR by treating these signals as ordinary nets, not specialized control signals that use global routing structures as do clocks. The modifications to VPR are summarized as:

- HierARC has been implemented in VPR v4.30 and given a BLIF netlist parser.
- The VPR netlist (.net) format has been augmented to describe chains and FFs.
- VPR's internal connectivity, timing, and routing graphs have been augmented to support chains with placement following [8].
- VPR FFs have been augmented with standard control signals, which are treated the same as cluster data inputs.
- VPR has been augmented to model LE capability, i.e. two $(K-1)$-LUTs.
- VPR has been improved to perform independent PNR experiments rather than generate single solutions, allowing more accurate data collection.
- HierARC clusters chains individually from head to tail and before arbitrary LEs.

VPR has been augmented to accommodate chains pursuant to the technique outlined in [8], as well as given several other useful features. This includes netlist definition with the ".chain" construct, integration with the connectivity, timing, and routing graphs, sophisticated FF handling, chain placement, and independent experiment generation and data collection. One reason VPR doesn't support carry chains is that its resource graph does not model them. A chain connection can be modeled with a delay-less switch that connects the output pin of a source node directly to the input pin of a sink node. To make use of chains, the routing resource graph incorporated by VPR [12] has been altered to provide chain connectivity between source and sink nodes; the routing array in Figure 7.5(a) is represented as 7.5(b).

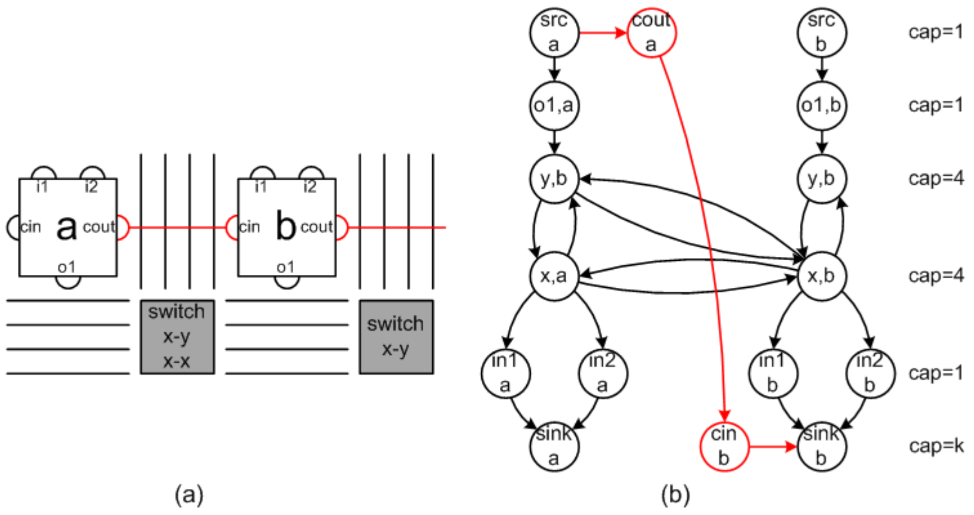


Figure 7.5 Alterations to the VPR routing resource graph.

A chain is a dedicated connection between LEs, which requires that it must connect both adjacent LEs residing in the same cluster and those in different clusters. Routing resource graphs do not have to model intra-cluster chain connections because they only model the connectivity of the general routing array and clusters. The inter-cluster chain connections are modeled as a connection between a cluster output pin (OPIN) and a cluster input (IPIN) that bypass routing tracks. Correspondingly, non-chain connections are precluded from using this routing structure, as they are not necessarily compliant with the unique constraints imposed by chains. Clusters that are connected via a chain, as specified by the input netlist, can only use this dedicated routing resource and do not have the option of using general routing resources.

To model the timing of an LE with the *cout* and *sum* LUTs present in chains, the timing graph must allow each LE to have multiple outputs that potentially depend on a different set of inputs. In an LE, the *cout* and *sum* nodes must be separated because their output edges are separate. The *cout* node is purely combinational and its output only connects to the *cin* input of the next node in the chain. The *sum* node can operate in either combinational or sequential mode, therefore, it can either serve as the output of the LE or connect to the FF sink. Synchronous control signals connect directly to the FF sink node with the combinational output, while asynchronous control signals connect to the FF source along with the clock.

For example, the circuit in Figure 7.6(a) contains a chain consisting of LEs $\{a, b, c\}$, and LEs $\{c, d\}$ are operating in sequential mode. LEs $\{a, b\}$ have *cout* nodes because they connect directly to the *cin* nodes of $\{b, c\}$, respectively. The delay of edge (a_{cout}, b_{cin}) is a user defined parameter, akin to the wire delay of a chain ($\approx 0ps$), while the delay of edges (a_{cout}, b_{cout}) or (a_{cout}, b), user defined parameters, correspond to the logic delay of a chain, i.e. a 2:1 multiplexer. Conversely, the combinational output of LEs $\{a, b\}$ link directly to the output pads $\{out0, out1\}$, respectively. Note that timing graph I/O pads are actually implemented as two nodes each, such that the edge connecting them models their delay because nodes have no delay. Figure 7.6 does not reflect this for sake of simplicity.

The combinational output of LEs $\{c, d\}$ are handled differently, as the LEs operate in sequential mode. They connect to FF sink nodes, $\{c_D, d_D\}$ with a delay corresponding to the setup time of the FF, T_{su}. The synchronous signals also drive the FF sink nodes to provide clear and preset capability via a user defined timing parameter. To isolate combinational paths (avoid loops), the FF sink and source nodes are not connected via an edge. Instead the FF source generates the output with a delay after the clock arrives (clock-to-out), T_{co}. This also holds true for the asynchronous control signals, which generate a FF output after a user defined delay similar to T_{co}. For added timing accuracy, intra-cluster connections between

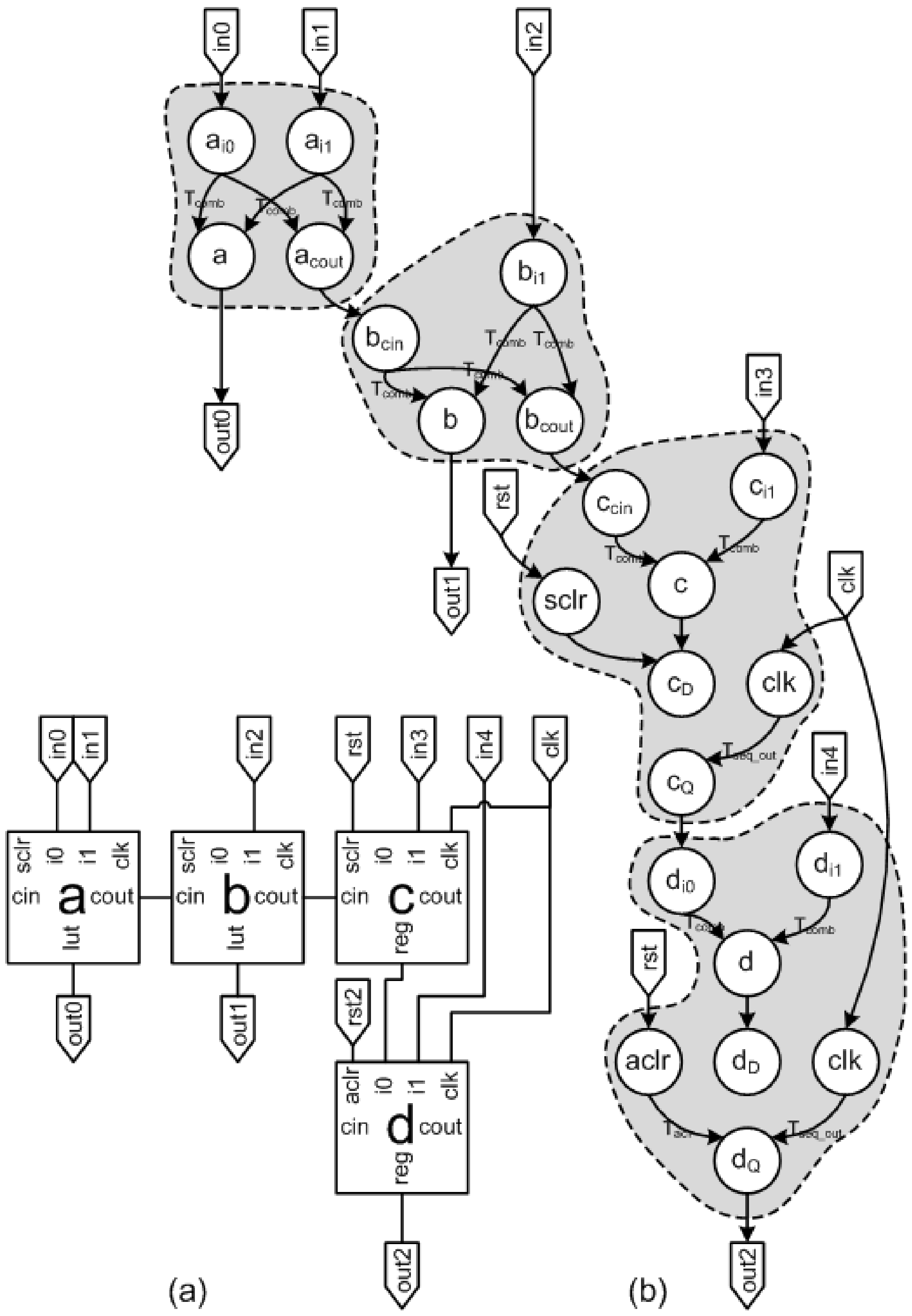
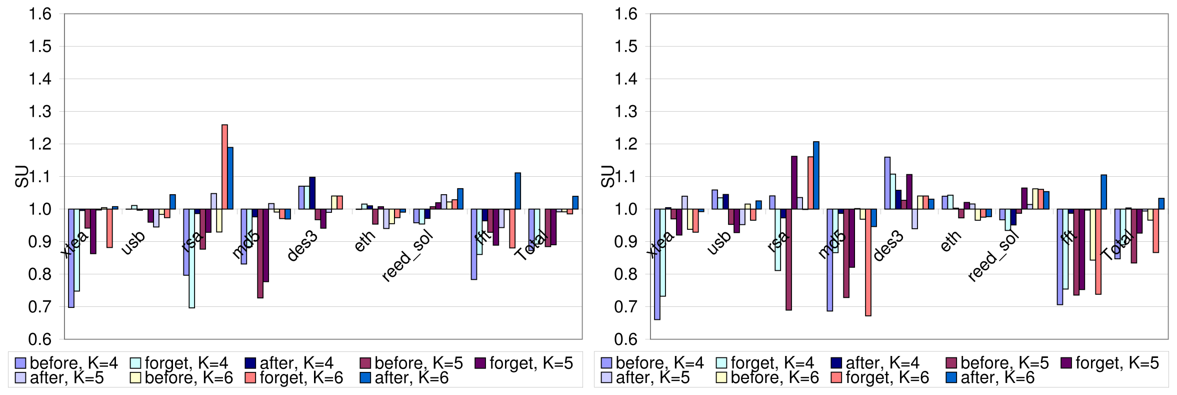


Figure 7.6 Alterations to the VPR timing graph.

LEs are also modeled by VPR. The modified chain timing graph models intra-cluster and inter-cluster chains with potentially different edge latency values.

7.3 Fixed Architecture Performance Assessment

Fixed architecture place and route performance results for ChainMap solutions are mixed, as given by Figures 7.7(a) and 7.7(b). In some cases, speedups of up to 1.26x are witnessed, while in others performance is reduced to 0.67x. Global least critical relaxation (*critical*) tends to produce more consistent results, while least critical branch trimming (*trimming*) can offer greater speedup to many circuits, but tends to suffer more severe performance degradation in other cases. Performance increases with K, as more sub-width arithmetic operations are able to be packed into wider LUTs.

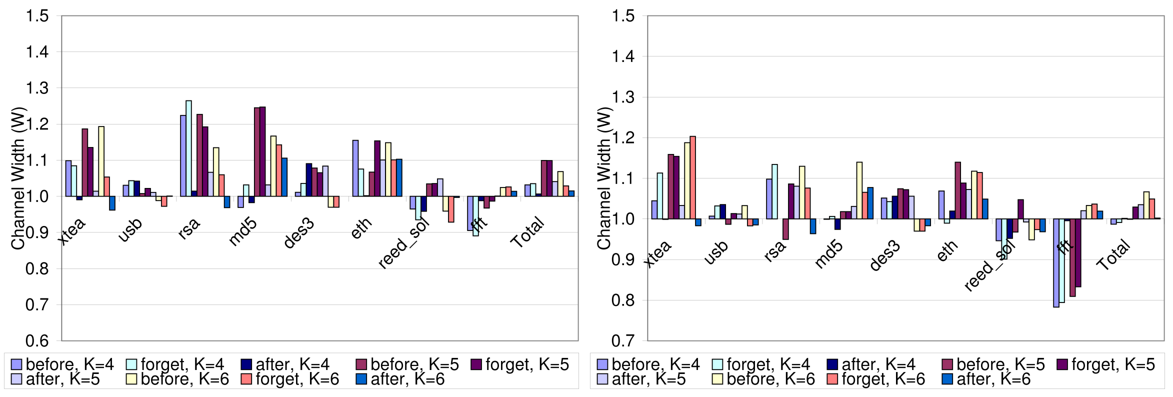


(a) Global least critical relaxation (b) Least critical branch trimming

Figure 7.7 Speedup, $N = 8$

It is not immediately clear why ChainMap solutions suffer from performance degradation, but a look into routing utilization reveals some characteristics of ChainMap solutions. Two routing metrics under consideration are channel width (W) and total wire length. Channel width specifies the maximum number of tracks required by each routing channel necessary to route the design. It is a useful indication of routability, as the more tracks per channel, the more each cluster is dependent on other clusters within the array. Total routed wire length is the sum of all routing paths, in 1-unit segments, contained in the circuit. Wire length gives an indication of the size of the circuit and also of its connectivity. Higher total wire length can be an indication of overall circuit size, as the more clusters a design has, the larger its average diameter, and further the distance between any given cluster pair. It is also indicative of the connectivity of a design, such that clusters using more external pins are harder to place relative to the clusters they source/sink connections to/from. This results in increased use of the general routing array because of the the pure number of connections each cluster has to make and/or poor locality to other clusters. Ratios of channel width or wire length that are less than 1 indicate resource savings.

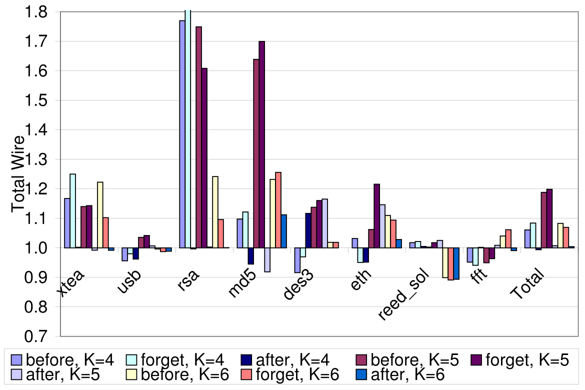
Figures 7.8(a) and 7.8(b) indicate mixed PNR routability results for ChainMap. Maximum channel width ranges from 0.8x-1.2x, with the average change being 1.05x. Likewise, total routed wire length, given by Figures 7.9(a) and 7.9(b), gives results in the range of 0.8x-1.8x, with the average change in wire length being a 1.1x increase. This is indicative of the increase in cluster connectivity prognosticated by cluster pin utilization. While increased pin utilization does not manifest as acutely in channel width, the total wire length results indicate consistent increases. The *after* flow, whose solutions most emulate HDL, yield the best routability results, in the average case resulting in neutral to reduced wire and channel width. The *before* and *forget* flows fall victim to the increased cluster pin requirements.



(a) Global least critical relaxation

(b) Least critical branch trimming

Figure 7.8 Channel width, $N = 8$


(a) Global least critical relaxation

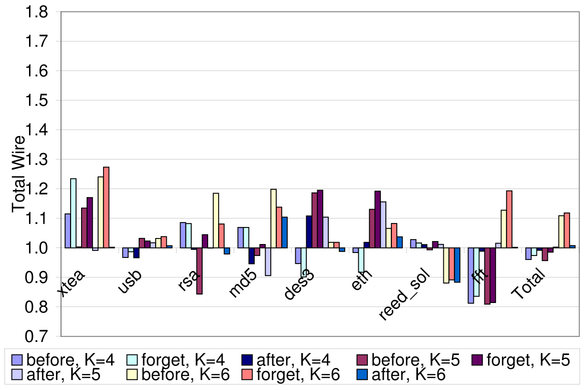

(b) Least critical branch trimming

Figure 7.9 Total routed wire length, $N = 8$

Table 7.1 Routing complexity for OpenCores Benchmarks, $K = 6$, *critical*

	Normal			Before			Forget			After		
Circuit	Clusters	Nets	R_{cost}	Clusters	Nets	R_{cost}	Clusters	Nets	R_{cost}	Clusters	Nets	R_{cost}
xtea	81	704	1040.9	83	654	1012.1	89	604	971.0	80	704	1040.7
usb	352	1575	2532.8	355	1581	2564.7	352	1587	2574.6	352	1594	2569.9
rsa	123	847	1277.9	119	741	1174.9	121	706	1142.4	121	822	1246.0
md5	255	1316	2022.6	283	1541	2416.3	292	1452	2317.6	283	1539	2357.8
des3_area	54	676	1051.1	54	676	1051.1	54	676	1051.1	54	676	1051.1
ethernet	36	203	320.3	34	202	326.1	34	203	327.6	34	203	324.3
reed_sol	158	586	973.8	140	504	847.5	140	506	850.4	141	507	851.9
cfft	428	3224	4766.3	412	2576	4137.4	424	2591	4154.2	408	3229	4773.1
Total	1487	9131	13985.7	1480	8475	13530.2	1506	8325	13388.9	1473	9274	14214.8
%change	–	–	–	-0.47	-7.18	-3.26	1.28	-8.83	-4.27	-0.94	1.57	1.64

Figures 7.8 and 7.9 indicate that ChainMap solutions have difficulty realizing their potential performance during PNR due to a decrease in routability. The primary contributors to increased routing complexity are an increase in the total number of nets, higher fanout per net, cluster connectivity, and architectural constraints such as chains. Tables 7.1, 7.2, and 7.3 give a pre-routing estimate of complexity as judged by number of clusters, number of nets, routing cost as computed with Equation 2.10 (R_{cost}, which captures fanout per net), total inter-cluster chains, average cluster/LE chain length ($L_{cluster}$, L_{LE}), and external cluster pin utilization for ($K = 6$, *critical*).

Table 7.2 Chain utilization for OpenCores Benchmarks, $K = 6$, *critical*

	Normal			Before			Forget			After		
Circuit	Chains	$L_{cluster}$	L_{LE}	Chains	$L_{cluster}$	L_{LE}	Chains	$L_{cluster}$	L_{LE}	Chains	$L_{cluster}$	L_{LE}
xtea	9	4.33	29.90	14	3.00	9.10	8	2.25	5.72	9	8.89	29.90
usb	5	2.00	7.86	0	–	3.06	0	–	3.02	5	70.40	5.38
rsa	8	5.00	33.50	14	2.57	8.29	10	2.00	5.56	8	15.13	30.11
md5	8	4.00	29.11	13	2.69	4.84	10	2.30	3.80	8	35.38	6.36
des3_area	0	–	5.00	0	–	–	0	–	–	0	–	5.00
ethernet	2	2.00	11.33	0	–	3.00	0	–	3.00	2	17.00	5.27
reed_sol	0	–	5.00	0	–	3.00	0	–	3.00	0	–	3.24
cfft	52	4.71	28.74	40	5.48	6.69	39	4.33	6.68	52	28.33	28.32
Total	84	4.40	22.28	81	2.70	5.76	67	2.52	4.84	84	4.40	12.83
%change	–	–	–	-3.57	-38.62	-74.17	-20.24	-42.73	-78.29	0.00	0.00	-42.41

For the most part, the routability factors indicate that ChainMap solutions exhibit very good characteristics, with average reductions in clusters, nets, and routing cost commonplace. Furthermore, the number of inter-cluster chains, and both $L_{cluster}$ and L_{LE} indicate that ChainMap solutions contain fewer overall chains as well as ones that are significantly shorter and often contained completely within clusters. Table 7.2 shows that the number of inter-cluster chains decreases in each case by -3.6% and -20.2% for $before$ and $forget$, respectively, and the length of such chains is significantly reduced by about -40% in terms of clusters, and -76% in terms of LEs. These results indicate that placement in Chainmap solutions is generally easier with regard to honoring inter-cluster chain relationships. Fewer intra-cluster chains give the clustering tool more flexibility, as fewer LEs are required to be grouped together. Likewise, fewer inter-cluster chains give the placement engine more freedom with which to swap clusters.

While resources, cost, and chain factors favor ChainMap solutions, cluster connectivity does not. Pin utilization per cluster is significantly increased by ChainMap, with external inputs (+33%), total external pins (+18%), and maximum number of external inputs per cluster (+12%) all sharply increased for $before$ and $forget$. This indicates higher connectivity between clusters, leading to increased competition for channel resources and more inter-cluster relationships that must be honored. The increase in cluster pin utilization serves as ChainMap's Achilles' heel. Contrasted with the predicted routability factors presented in Tables 7.1 and

Table 7.3 Cluster pin utilization, $K = 6$, *critical*

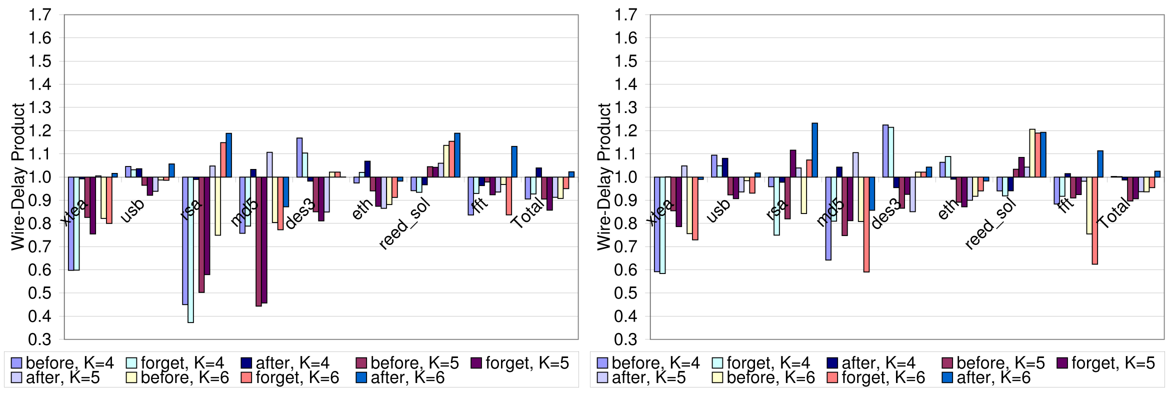
	Normal			Before			Forget			After		
Circuit	$\frac{ext\ in}{cluster}$	$\frac{ext}{cluster}$	$\frac{max\ ext}{cluster}$	$\frac{ext\ in}{cluster}$	$\frac{ext}{cluster}$	$\frac{max\ ext}{cluster}$	$\frac{ext\ in}{cluster}$	$\frac{ext}{cluster}$	$\frac{max\ ext}{cluster}$	$\frac{ext\ in}{cluster}$	$\frac{ext}{cluster}$	$\frac{max\ ext}{cluster}$
xtea	4.65	10.69	27	11.06	16.52	38	10.18	14.75	38	4.58	10.69	27
usb	8.19	12.46	34	8.21	12.44	36	8.28	12.53	35	7.97	12.32	36
rsa	8.75	14.63	30	15.36	20.94	34	15.21	20.33	36	8.86	14.64	29
md5	11.70	16.29	36	11.63	16.66	33	12.05	16.74	33	10.94	15.96	35
des3_area	13.85	20.00	26	13.94	20.06	26	13.94	20.06	26	13.85	20.00	26
ethernet	6.69	11.89	32	8.03	13.47	38	7.74	13.21	38	7.56	13.06	35
reed_sol	16.54	20.20	34	15.90	19.50	35	15.94	19.56	35	15.81	19.42	35
cfft	6.91	13.96	29	14.94	20.82	36	15.05	20.79	38	6.77	14.18	29
Total	9.33	14.71	248	12.41	17.46	276	12.49	17.37	279	9.05	14.62	252
%change	–	–	–	32.91	18.67	11.29	33.79	18.04	12.50	-3.00	-0.66	1.61

7.2, routing results indicate that increased cluster pin utilization reduces the overall routability of the design.

The area-delay product can be used to judge the overall effectiveness of each design. It has been shown in [57] to be a reasonable measure of overall FPGA performance and can be adapted to solution performance. Equation 7.1 gives area-delay as a product of the total routed wire length and the maximum clock frequency ($P_{wire-delay}$). A wire length or period increase causes $P_{wire-delay}$ to decrease. Wire length is used because a fixed-size FPGA array contains an equal number of transistors regardless of the design size. Wire length gives a sense of the active part of the array because, as more wire is used, more transistors are actively being incorporated in the design. Figure 7.10 gives the $P_{wire-delay}$ response for the *normal* flow divided by the *before* and *forget* flows. $P_{wire-delay}$ ratios greater than 1 indicate performance increase.

$$P_{wire-delay} = \frac{1}{A_{wire} \cdot T_{max}} \tag{7.1}$$

In the average case, $P_{wire-delay}$ of ChainMap designs reflect the PNR routability results. Simply, the difficulty in routing ChainMap solutions translates to increased wire length and path delay, resulting in a decrease in $P_{wire-delay}$. Figure 7.11 depicts routed solutions of $cfft$, for ($before$, $K = 6$, $critical$) and ($normal$, $K = 6$). It shows the ChainMap solution as



(a) Global least critical relaxation

(b) Least critical branch trimming

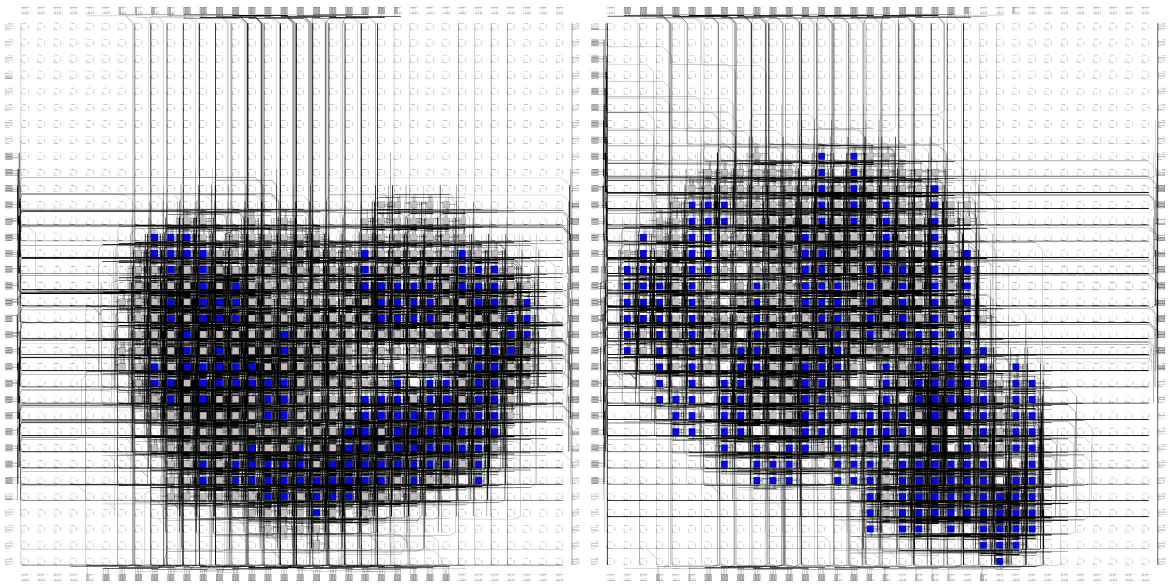
Figure 7.10 $P_{wire-delay}$, $N = 8$

generally more congested and dense. The long arithmetic chains of *normal*, shown in dark, have the effect of spreading the design out. Ultimately, to rectify the technology map performance estimates of Section 5.6 and the fixed array PNR results, ChainMap has to produce solutions that are more place and route friendly.

7.4 Scaled Architecture Performance Assessment

The parameterized (scaled) architecture enables the assessment of solutions implemented in an FPGA with minimum resources. The array size is tailored to the minimum number of I/O pads, maximum chain length, or clusters, and the interconnection network is the minimum channel width necessary to route the design. The aspect ratio of the cluster array is kept at 1.0x, i.e. the number of clusters per row is the same as the clusters per column.

The scaled architectural results differ significantly from those of the fixed architecture. Figure 7.12 indicates that if cluster resources are limited, ChainMap solutions can often yield a significant decrease in critical path latency. The primary reason for this is alluded to by the chain results of Table 7.2, ChainMap solutions have reduced placement constraints. On average, they require fewer inter-cluster chains that are shorter in cluster/LE length, than

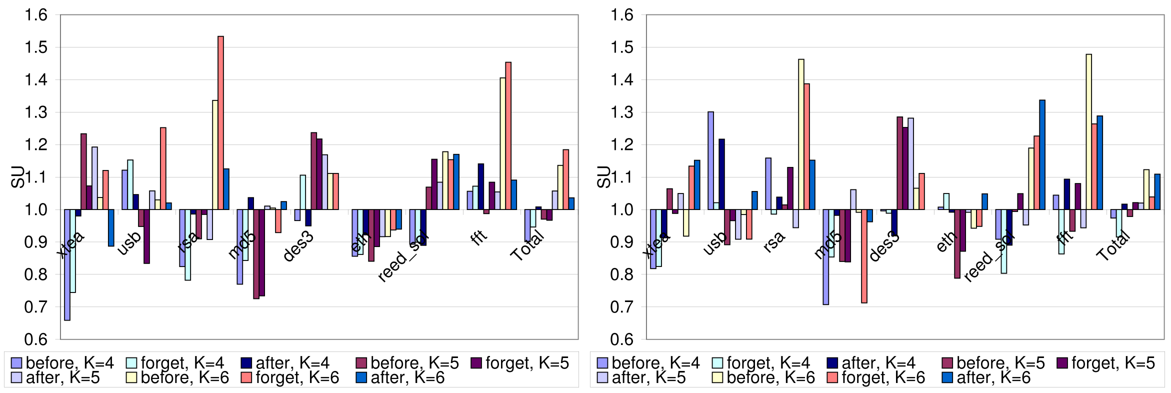


(a) $before$, $\Delta X = 26$, $\Delta Y = 22$, $d = 34.1$ (b) $normal$, $\Delta X = 26$, $\Delta Y = 26$, $d = 36.8$

Figure 7.11 Post-routing implementations for $before$ global least critical relaxation and $normal$, $K = 6$.

traditional designs. The reduction in the number of inter-cluster chains allows the placement tool to test and accept/reject more chain swaps.

Recall from Section 2.2.4 that chain placement using the method of [8] works by establishing two identically sized regions incorporating the chains of the swap. If either of these regions violates the cluster array boundary or severs an existing chain, the potential swap is deemed invalid and a new random swap is generated, repeating the process. The invalid swap does not count as a simulated annealing accept/reject, but instead is simply discarded. If the swap is deemed valid, it is executed, rated according to VPR's cost metric, and accepted/rejected. The shorter cluster chains generated by ChainMap solutions increase the likelihood of a legal swap, thus increasing the number of potential acceptances/rejections. Additionally, shorter chains result in smaller swap regions and fewer clusters and nets being relocated. The less the network is perturbed, the more likely a swap is accepted, therefore resulting in better overall chain placement. An increase in valid swaps and a higher rate of acceptance allow the development of better placements for ChainMap solutions.



(a) Global least critical relaxation (b) Least critical branch trimming

Figure 7.12 Speedup, $N = 8$

Table 7.4 Cluster array dimensions, $K = 6$, *critical*

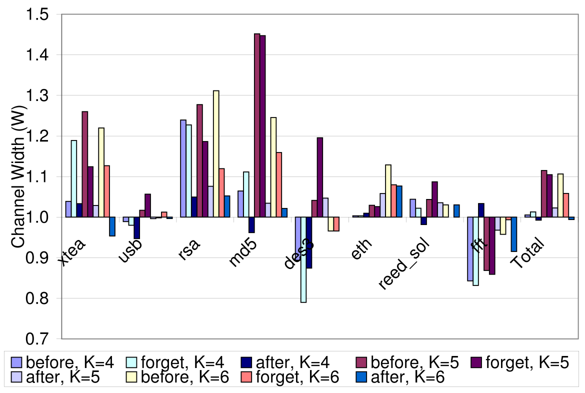
	Normal				Before				Forget			
	Fixed		Scaled		Fixed		Scaled		Fixed		Scaled	
Circuit	X, Y	d_{fix}	X, Y	d_{scale}	X, Y	d_{fix}	X, Y	d_{scale}	X, Y	d_{fix}	X, Y	d_{scale}
xtea	11.0,11.2	15.78	10.8,11.1	15.54	10.7,11.2	15.67	10.6,10.6	15.12	11.4,10.4	15.50	10.4,11.4	15.40
usb	21.8,21.7	30.84	19.0,19.0	26.87	21.3,22.4	30.99	19.0,19.0	26.87	21.0,21.9	30.36	19.0,19.0	26.87
rsa	14.8,18.3	23.85	11.9,12.0	16.90	14.3,16.3	22.04	11.0,11.0	15.56	14.5,15.4	21.40	11.0,11.0	15.56
md5	18.9,18.0	26.17	17.0,17.0	24.01	18.5,20.1	27.40	17.0,17.0	24.04	20.3,19.5	28.51	18.0,18.0	25.42
des3_area	34.0,34.0	48.05	34.0,34.0	47.98	34.0,34.0	48.08	34.0,34.0	48.08	34.0,34.0	48.08	34.0,34.0	48.08
ethernet	26.9,26.2	39.00	6.0,6.0	8.49	29.6,26.7	40.46	6.0,6.0	8.49	26.3,27.1	38.72	6.0,6.0	8.49
reed_sol	15.4,13.7	20.71	13.0,13.0	18.38	13.5,13.8	19.36	12.0,12.0	16.97	13.5,14.1	19.69	12.0,12.0	16.97
cfft	26.5,24.6	36.19	21.0,21.0	29.70	25.0,25.0	35.41	21.0,21.0	29.70	24.4,24.6	34.75	21.0,21.0	29.70
Total	169,168	241	133,133	188	167,169	239	131,131	185	165,167	237	131,132	187
%change	–,–	–	–,–	–	-1.5,1.1	-0.49	-1.4,-1.8	-1.62	-2.4,-0.3	-1.49	-0.9,-0.5	-0.74

The second reason for improvement is that arrays with an over-abundance of resources, relative to the design size, allow the design to spread out. Table 7.4 verifies what Figure 7.11 demonstrates; if given virtually unlimited resources, the *normal* flow will generate designs that have higher average diameter. The dimensions of the circuit in the fixed array, in terms of cluster diameter (d) and X, Y width indicate that ChainMap designs have a smaller footprint. This leads to congestion when resources are plentiful, but aids the solution when resources are at a premium.

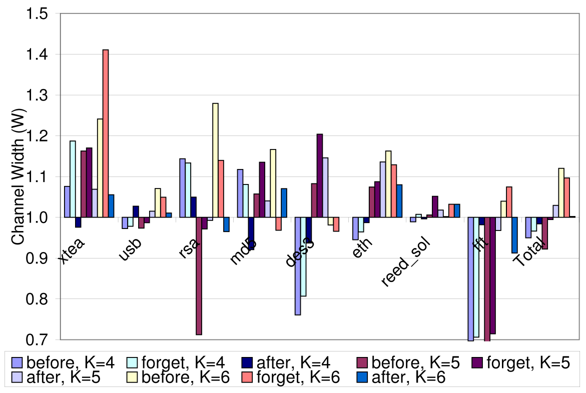
The final reason for improved results for scaled arrays is that ChainMap solutions often require fewer LEs and, consequently, fewer clusters (Table 7.1). In some cases, this allows the use of a smaller cluster array. The scaled solution dimensions (X, Y) in Table 7.4 are in all cases the same size of the array. In most cases, the $normal$ and $before$ dimensions are the same, however in the cases of rsa and $reed\_sol$, the required array size actually decreases. Only in the case of $md5$ does the ChainMap solution require a bigger array than $normal$, and the speedup results of Figure 7.12 indicate the repercussions.

Routability in the scaled solutions changes little from that of the fixed architecture. Figures 7.13 and 7.14 indicate that ChainMap designs still record higher minimum channel width and more total wire length than $normal$. This is still a symptom of the increased connectivity between clusters which hinders ChainMap solutions. However, an increase in wire length does not necessarily mean performance degradation. The amount of wire available in the general routing array is the same whether or not the circuit uses it; ChainMap results, though they require much more routing, are simply making use of what is available. The small increase in average channel width versus the large increase in wire length indicates that ChainMap solutions do not require a prohibitively increase in static routing resources, but rather that they more completely use the available wire. Channel width does not have to increase substantially, but use of the channel will for ChainMap circuits. Despite the close quarters in the scaled architecture, clusters continue to require the same I/O and thus result in similar channel width and wire length.

The scaled architecture allows the measurement of the area-delay product using the transistors in the design. Area is measured in terms of the number of minimum-width transistors the architecture uses, where minimum width includes the area of the transistor plus its spacing for a given process size. If non-minimum width transistors are required to implement a specific component, such as when increased drive strength is necessary, the number of minimum-width transistor equivalents (MTEs) are tallied. The area of the design includes the transistors in the interconnection matrix and local cluster interconnection for any combination of architectural parameters [12]. Area estimation is not useful in the fixed architecture because the array size and interconnection array are identical between designs, thus giving all solutions identical

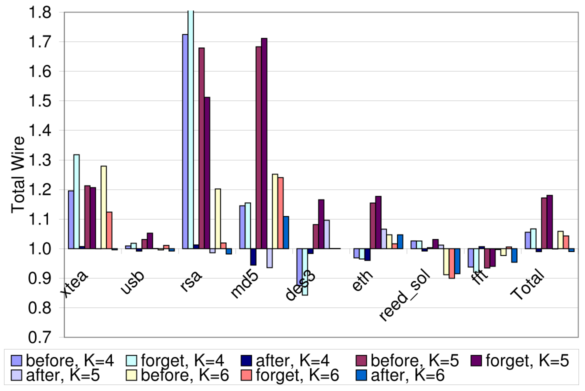


(a) Global least critical relaxation

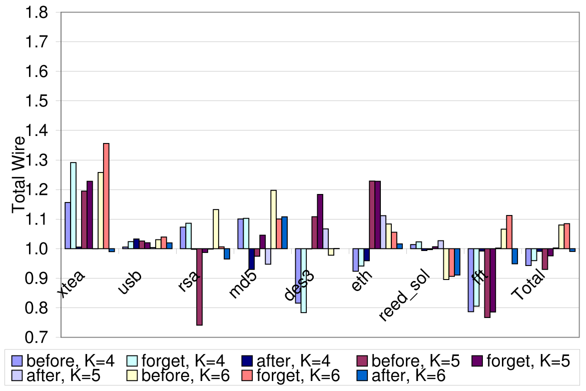


(b) Least critical branch trimming

Figure 7.13 Channel width, $N = 8$



(a) Global least critical relaxation

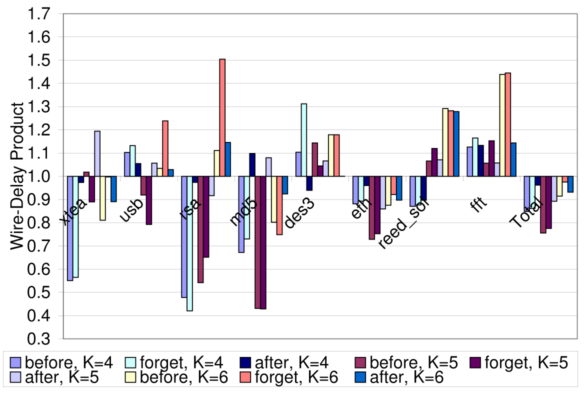


(b) Least critical branch trimming

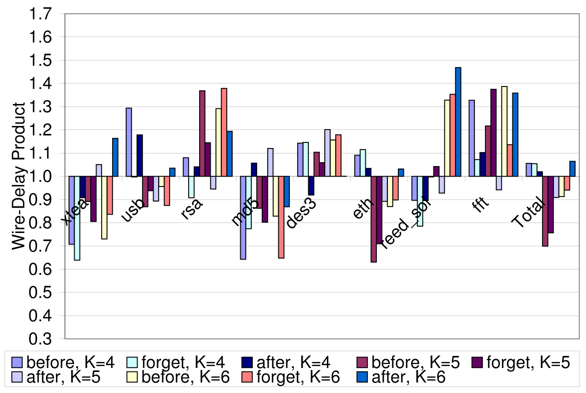
Figure 7.14 Total routed wire length, $N = 8$

transistor area. $P_{area-delay}$, given by Equation 7.2, for the scaled architecture accounts for minimum cluster and routing array sizes. $P_{area-delay}$ greater than 1 indicates better overall performance.

$$P_{area-delay} = \frac{1}{A_{MTE} \cdot T_{max}} \tag{7.2}$$

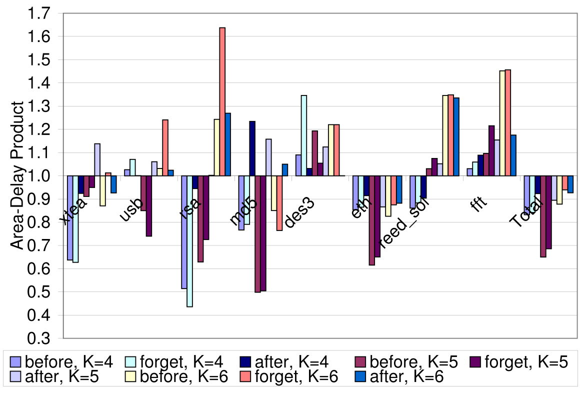


(a) Global least critical relaxation

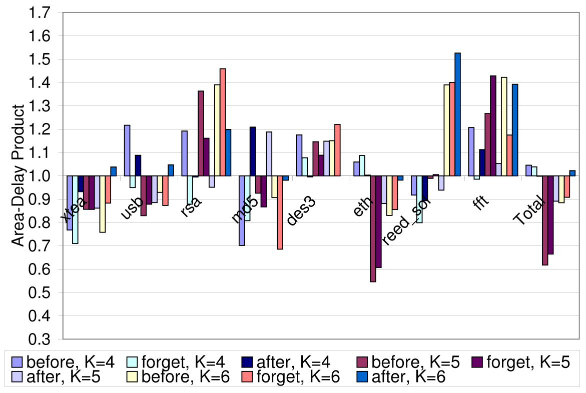


(b) Least critical branch trimming

Figure 7.15 $P_{wire-delay}$, $N = 8$



(a) Global least critical relaxation



(b) Least critical branch trimming

Figure 7.16 $P_{area-delay}$, $N = 8$

Figures 7.15 and 7.16 present $P_{wire-delay}$ and $P_{area-delay}$, respectively. They indicate that in cluster-constrained arrays, given the required minimum channel width, ChainMap yields solutions that balance area and delay effectively, resulting in up to a 1.65x increase in $P_{area-delay}$ (rsa, $forget$, $critical$, $K = 6$). The biggest improvements in area-delay occur in the cases where the size of the cluster array can be reduced as a result of lower cluster utilization, such as in rsa and $reed\_sol$. Conversely, when the cluster array size is increased, such as in $md5$, area-delay decreases sharply.

7.5 Summary

The full design flow experiments indicate that, while ChainMap endows designs with good characteristics across a variety of factors, these do not immediately translate to overall design performance when resources are abundant. However, when the chip size or alloted array space is tailor-fit to the design and routing resources are sufficient, ChainMap has the ability to produce very significant performance increases. Post-technology map performance estimations and the lion share of routability measures favor ChainMap, but one very important metric, external pins per cluster, does not. An increase in the number of external connections per cluster causes each them to be increasingly dependent on each other, and results in neutral average maximum clock frequency change and, in some cases, frequency degradation for large cluster arrays. When cluster array sizes are small, ChainMap's affinity toward less numerous, shorter cluster chains allows it to create higher quality placements that translate to better routed solutions.

Among the designs that fare well, in the fixed-size architecture, are those that have few to no arithmetic chains in HDL. Designs such as $des3\_area$ and $reed\_sol$ realize performance increases of up to 1.10x in terms of critical path latency, and up to 1.19x in $P_{wire-delay}$. Designs that fare poorly are those that contain a modest number of arithmetic chains, such as $xtea$ and $md5$. In these cases, path latency degradations of 0.7x ($xtea$, $before$, $K = 4$, $critical$) and 0.73x ($md5$, $before$, $K = 5$, $critical$) are witnessed. This is a direct result of increased cluster connectivity caused by the packing of multiple arithmetic bitslices into large K-LUTs and the tendency of long chains to decrease design density. In general, increasing LUT size has the effect of increasing performance slightly for the average case, but in some cases, like $xtea$ and $md5$ at $K = 4$, causes sharp decreases.

For cluster-constrained architectures, high-arithmetic designs witnessed the most substantial performance increases, contrary to the fixed-size architecture. Long chains that lead to sparse placements and better routed solutions for fixed-sized arrays prove to be handicaps when resources are limited. Among those designs that fare well in scaled architectures are the heavily arithmetic rsa and $cfft$, at 1.26x ($before$, $K = 6$, $critical$) and 1.51x ($before$, $K = 6$, $critical$), respectively. In the average case, scaled architecture performance is 1.11x that of $normal$ for ($before$, $K = 6$, $critical$).

With regard to relaxation techniques, global least critical relaxation tends to yield more predictable performance relative to least critical branch trimming. Due to its no-duplication policy, branch trimming has the potential to yield significant performance increases in individual cases, but is also prone to higher performance degradation. Nevertheless, both techniques prove the need to develop more effective relaxation strategies.

While the $before$ and $forget$ flows suffer the greatest from increased cluster connectivity in large cluster arrays, the $after$ flow is capable of taking advantage of ChainMap as an immediate addition to HDL arithmetic chains. In heavily arithmetic designs such as $cfft$, this means that the overall effect of ChainMap is negligible. However, in designs that feature few to no arithmetic chains such as $des3\_area$ and $reed\_sol$, ChainMap is an appropriate addition to the design flow. Designs that previously left the carry chain idle can now take full advantage of the low latency resource.

The results convey the need for improved cut selection, relaxation, clustering, and placement techniques that can take advantage of the potential performance increases outlined in Section 5.6 and Table 7.1. Because cluster connectivity is the largest contributing factor to an increase in routing complexity, not inter-cluster chains, the challenge becomes how to evoke better cut selection during the technology mapping stage where LE connectivity is established. Nevertheless, fewer inter-cluster chains are shown to significantly improve design performance when cluster resources are limited, indicating ChainMap's utility in production systems. Furthermore, designs with multiple IP cores typically constrain each core to a subset of the array clusters, inducing cluster constraints that emulate the scaled array results.

CHAPTER 8. DISCUSSION, CONTRIBUTIONS, AND CONCLUSION

ChainMap provides a polynomial time solution to the problem of identifying generic logic chains in an arbitrary Boolean network. By looking at the problem of circuit depth from the perspective of minimizing routing depth, the optimal baseline of performance for chains is established. Methods for relaxing optimal chain mappings are presented and shown to retain some of the potential performance indicated by post-technology map experiments. Final place and route experiments show that ChainMap can offer performance increases in some cases when resources are abundant, but results in performance degradation in others. However, when cluster resources are limited, ChainMap designs provide significant performance improvements. It is demonstrated that ChainMap is a viable solution to the problem of mapping chains without HDL.

8.1 Discussion

Ultimately ChainMap's performance is dependent on the entire design flow; efficient synthesis, minimal-area and routability based cut selection, relaxation that balances area, routability, and/or delay, clustering that reduces connectivity, placement that deals with chains effectively, and routing that combats congestion. While the techniques presented are sufficient to demonstrate proof of concept and conditions under which performance can be significantly increased are identified, new techniques need to be developed to take full advantage of optimal chain selection in all situations and for all circuits. The measures of routing complexity discussed in Tables 7.1 and 7.2 (clusters, nets, routing cost, chains) indicate that ChainMap solutions possess very good characteristics that, in many cases, position them as better solutions. However, full place and route results on fixed-size arrays do not bear this observation out. The primary culprit is in the connectivity demanded by each cluster. ChainMap solutions require, on average, 30% more input and 18% more total connections from the general routing array per cluster. This contributes to cases where channel width and total routed wire length are increased. Increased connectivity can make the design harder to place and ultimately degrade maximum clock frequency. Results change drastically when considering resource-constrained arrays, as the less numerous and shorter chains favored by ChainMap facilitate the generation of better placement solutions. The placement complexity created by increased cluster connectivity is far out-weighed by the constraints imposed by long HDL arithmetic chains. The seriousness of HDL constraints are exposed when the cluster resources in the array are limited.

There is one main reason for an increase in cluster connectivity–node duplication. ChainMap solutions lead to an increase in implicit and explicit node duplication during technology mapping. Implicit duplication occurs when multiple K-feasible cuts include the same Boolean node. Many such cases result in the inputs of the duplicated node having to map to multiple K-LUTs. Nodes in an arithmetic chain typically share very few inputs between each other and when conglomerated into LUTs via large K-feasible cuts, they are often done so across multiple LUTs, resulting in implicit duplication. Figure 5.3 shows how nodes are implicitly duplicated by ChainMap. The K-feasible cuts used by two separate nodes to generate their respective LUTs can overlap, and result in each LUT implementing the overlapping nodes. Explicit duplication occurs during relaxation, and although it protects critical delay paths through the network, it also requires the connectivity of the node be duplicated.

Working in its favor, ChainMap significantly reduces the prevalence and size of inter and intra-cluster chains. Intra-cluster chains dictate the clustering solution by grouping together LEs that would normally have little in common. Inter-cluster chains create cluster dependencies and constrain placement. The longer chains favored by HDL are a hindrance when cluster resources are limited, but an advantage if resources are unlimited. They are a double-edged sword that can alternately constrain placement and spread the design out and reduce overall routing complexity. The duality of longer cluster chains is witnessed in scaled architecture performance. The traditional approach to chains requires longer cluster chains that are difficult to manipulate by the placement engine. This is manifested in a larger design diameter relative to ChainMap solutions (Table 7.4). ChainMap designs are able to generate much more efficient placement solutions in limited array space because fewer inter-cluster chain constraints need be honored. If given sufficient routing resources, ChainMap placement solutions yield better routed solutions in area-constrained arrays.

Work in [28] finds that while LUT utilization is an admirable goal, it is inappropriate if it causes the interconnect not to be used to its full potential. Realizing the full potential of one resource demands that the other must go underutilized. In the case of ChainMap, if LEs are fully utilized, they cause an increase in LE connectivity and ultimately cluster connectivity. Consequently, when the number of LEs and inputs available in each cluster are fully utilized, they tend to degrade the performance of interconnect through congestion. This conclusion is corroborated by [71], which finds that limiting the number of LEs per cluster is an effective strategy in reducing channel width. This extends to HierARC, which may prove to be too efficient at achieving high cluster utilization.

Table 8.1 concurs, indicating an increase in average fanout per node that, ironically, occurs concurrent with a decrease in routing cost. This indicates that routing cost does not necessarily indicate overall routability, and that there is a need to sacrifice increased LE utilization for decreased net and LE fanout. The performance of clustering, placement, and routing can only be as good as the underlying technology map. While ChainMap solutions hold great promise, further work needs to be done to encourage cuts that not only minimize routing depth, but also reduce node fanout.

Performance also hinges on the number of LEs in a design. In most cases, ChainMap does a good job of reducing the total number of LUTs, but in some cases has difficulty translating LUT reduction to LE reduction. Table 8.2 shows sample LE/LUT utilization results for (*critical*, $K = 5$). It indicates that the magnitude of change of LUTs for ChainMap solutions does not always extend to the number of LEs. In many cases, this means an inordinate increase in the number LEs in a design at the same time as a reduction in the number of LUTs. For example, the number of LUTs relative to *normal* is reduced to 0.94x for (*xtea*, $before$), but the number

Table 8.1 Technology map complexity, $K = 6$, *critical*

	Normal			Before			Forget			After		
Circuit	$\frac{fanout}{net}$	$\frac{fanout}{LE}$	R_{cost}	$\frac{fanout}{net}$	$\frac{fanout}{LE}$	R_{cost}	$\frac{fanout}{net}$	$\frac{fanout}{LE}$	R_{cost}	$\frac{fanout}{net}$	$\frac{fanout}{LE}$	R_{cost}
xtea	3.10	4.14	1821.23	4.17	5.34	1732.42	4.05	4.70	1706.91	3.10	4.15	1816.66
usb	4.80	5.15	5127.51	5.21	5.37	4939.63	5.20	5.34	4933.15	4.77	5.23	5214.02
rsa	4.02	5.03	2083.06	5.32	6.29	1976.37	5.18	5.91	1930.86	3.99	5.00	2095.30
md5	4.68	5.25	3646.24	4.74	5.42	4371.28	4.69	5.35	4559.44	4.30	4.96	4248.94
des3_area	2.28	2.29	1263.10	2.29	2.29	1248.88	2.29	2.29	1248.88	2.28	2.29	1263.10
ethernet	4.19	4.58	543.77	4.71	5.01	501.95	4.72	4.98	498.82	4.15	4.79	537.96
reed_sol	5.71	5.73	2005.57	6.04	6.15	1854.96	6.04	6.16	1855.32	5.99	6.14	1868.13
cfft	3.74	5.44	8391.16	5.78	6.93	6839.31	5.75	6.84	6858.65	3.75	5.52	8201.61
Total	4.17	5.03	24882	5.11	5.74	23465	5.07	5.62	23592	4.13	5.06	25246
%change	–	–	–	22.65	13.96	-5.69	21.61	11.59	-5.18	-0.92	0.46	1.46

Table 8.2 Area Paradox for OpenCores Benchmarks, $K = 5$, *critical*

	Normal		Before				Forget				After			
Circuit	LUT	LE	LUT	λ_{LUT}	LE	λ_{LE}	LUT	λ_{LUT}	LE	λ_{LE}	LUT	λ_{LUT}	LE	λ_{LE}
xtea	1009	731	944	0.94	872	1.19	935	0.93	854	1.17	974	0.97	696	0.95
usb	3157	3112	3119	0.99	3202	1.03	3123	0.99	3206	1.03	3283	1.04	3159	1.02
rsa	1162	911	1010	0.87	981	1.08	1009	0.87	982	1.08	1162	1.00	911	1.00
md5	3002	2759	3111	1.04	3000	1.09	3141	1.05	3026	1.10	2796	0.93	2475	0.90
des3_area	824	908	894	1.08	886	0.98	926	1.12	915	1.01	903	1.10	886	0.98
ethernet	260	287	244	0.94	290	1.01	244	0.94	290	1.01	265	1.02	283	0.99
reed_sol	1227	1223	1221	1.00	1206	0.99	1219	0.99	1204	0.98	1229	1.00	1208	0.99
cfft	4764	3345	3517	0.74	3517	1.05	3522	0.74	3524	1.05	4645	0.98	3222	0.96
Total	15405	13276	14060	0.91	13954	1.05	14119	0.92	14001	1.05	15257	0.99	12840	0.97

of LEs jumps to 1.19x. This is mainly due to the the overpopulation of wider K-LUTs that causes them to violate the input limit aspect of the exclusivity constraint (Lemma 5.4.1). In two cases, *reed_sol* and *des3_area* (*before*), the ratio of LEs actually decreases relative to the ratio of LUTs. This occurs because these two designs are strongly non-arithmetic (Table 1.1) at less than 1.5%. ChainMap and ChainPack actually encourage LE formation and reduced consumption in these cases.

An example of the area paradox, found in *xtea*, is shown in Figure 8.1 which depicts nodes and their chain connections (general routing nets are omitted). The *normal* implementation

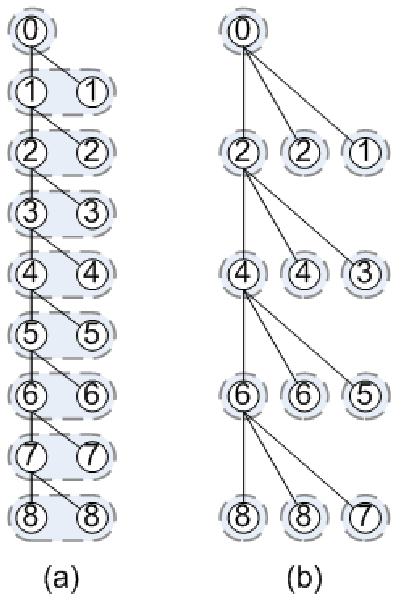


Figure 8.1 Area paradox for xtea, (a) *normal* with 17 LUTs, 9 LEs and, (b) ChainMap with 13 LUTs, 13 LEs.

depicted in (a) uses 17 LUTs, but all are paired up as *sum* and *cout* members of an LE. Because each LUT has a mate, the number of LEs required in the *normal* case is 9. The ChainMap solution in (b) reduces the overall LUT count to 13, but each LUT is independent and must be implemented in its own LE. Any of the reasons outlined in the exclusivity constraint of Lemma 5.4.1 can require LUTs to occupy separate LEs, including too many distinct inputs, homogeneous outputs, and inter-dependence. The most prevalent reason that so few LE pairs can be formed in the ChainMap solution is that each LUT becomes overpopulated to the point where LE pairs are disqualified due to too many distinct inputs. Because of the area paradox, using LUTs as a measure of area can be misleading. On average, it results in the *forget* and *before* flows requiring slightly more LEs than *normal*. LUTs can be encouraged to form more LEs by choosing cuts that are not as wide, but still depth optimal, as discussed in Section 5.2. The area paradox is also related to connectivity concerns, as each over-populated LUT that ChainMap forms increases its fanin, thus simultaneously disqualifying it as an LE and increasing its connectivity.

8.2 Contributions

ChainMap establishes an optimal baseline of performance for generic logic chains in FPGAs. In doing so, it finds that HDL-based chains are sub-optimal solutions that don't fully realize their potential. However, they are a very good heuristic solution to using the high performance carry chain resource available in most modern FPGAs. This dissertation's contributions are:

1. A formal logic chain definition is presented that encompasses both arithmetic and non-arithmetic operations.
2. A logic cell design capable of driving a carry chain with a full K-LUT operation is shown to have a minimal effect on area and performance while facilitating generic logic chains.
3. ChainMap is presented as a technology mapping algorithm that creates optimal generic logic chains in polynomial time without HDL arithmetic chain macros.
4. Global least critical relaxation and least critical branch trimming techniques convert optimal chain mappings to feasible solutions.
5. HierARC is presented as a deterministic, polynomial run-time, scalable, and effective clustering tool for island-style FPGAs.

The definition of a logic chain has been formalized as a series of depth increasing nodes, such that there is a directed $edge(u, v)$ between adjacent nodes $\{u, v\}$, that causes the logic depth of v to increase while not increasing its routing depth. This definition addresses the fact that there is a clear difference in the speed of routing versus chain nets, and guides their use.

The carry chain reuse cell [36] makes it possible to use the high performance carry chain to propagate full K-input operations. Traditional carry-select adders have the ability to compute two arbitrary $(K-1)$-input functions using the same inputs and transmit one along along the carry chain to a subsequent LE and the other to the general routing array. The reuse cell allows designers to compute the same two $(K-1)$-input functions, or one K-input function whose output is simultaneously transmitted to the carry chain and general routing array. To facilitate this, the reuse cell incorporates two additional 2:1 pass-transistor multiplexers that have negligible delay and area effect. The reuse cell requires no additional inter-LE connectivity to achieve this.

The optimal logic chain technology mapping algorithm, ChainMap, achieves an estimated post-technology mapping average speedup of 1.4x optimally versus traditional HDL-defined chains. The average performance difference, post-technology mapping, between disregarding HDL macros completely and inserting chains before mapping is within 5%, indicating HDL preservation might potentially be abandoned. This could affect the entire FPGA design flow, allowing CAD designers to expand algorithms past the partitions created by HDL. Since the best area/speedup is usually achieved by the insertion of arithmetic chains $before$ mapping, one inference is that they are already highly optimized in terms of literal count, and resynthesis creates sub-optimality. Another possibility is that improved synthesis techniques, such as those offered by ABC [59], could potentially generate Boolean networks more suited to logic chains. ChainMap demonstrates that generic logic chains have the potential to perform better than solely arithmetic ones.

ChainMap requires an area/speedup trade off, an artifact of FPGAs enforcing the *exclusivity constraint*. However, three simple relaxation heuristics allow ChainMap to produce consistent reductions in LUT consumption. LUT reductions of up to 0.71x are witnessed ($cfft$, $K = 5$, $before$, $shallow$). Optimal solutions, while prohibitive from an area standpoint, indicate that better relaxation techniques have the potential to yield ubiquitous speedup increases. Global least critical relaxation, shallowest logic branch trimming, and least critical branch trimming all mitigate area expansion while preserving some of the optimal performance.

The post-technology map results set a baseline for potential performance, but complete design flow experiments indicate that additional improvements to the entire design flow, including ChainMap, are necessary to realize estimated performance in the final placed and routed solution. While final place and route ChainMap solutions for large cluster arrays are, on average, comparable to the *normal* approach, speedup ranges from 0.7x to 1.25x. When cluster resources are limited, ChainMap solutions yield significant performance improvements through higher-quality placement solutions. ChainMap favors fewer, smaller chains that have the effect of reducing the constraints on placement posed by long HDL-based chains. This allows placement of ChainMap solutions to generate more valid swaps and an increased number of swap acceptances. Critical path latency in cluster-constrained architectures ranges from 0.67x to 1.53x with 1.1x average. These results indicate the need for future work to unleash the full potential of optimal generic logic chains, but also that ChainMap is a valid solution for cluster-constrained architectures.

Finally, HierARC [39] is presented as a Hierarchical Agglomerative Reconfigurable fabric Clustering tool. HierARC, possessing only a routability metric, is capable of reductions in channel width (-21%), wire length (-23%), and transistor area (-21%) with neutral critical path change (+0.03%) relative to T-Vpack. It also compares favorably to other clustering tools published in literature, often times exceeding their relative T-Vpack performance. HierARC is polynomial in run-time, scalable with respect to incorporating performance metrics through Euclidian distance, deterministic, and can easily accommodate resource constraints without post-processing techniques such as cluster depopulation.

8.3 Future Work

Sections 5.6 and 7.2 indicate that relaxation, placement, and routing fail to fully harness ChainMap performance, primarily due to increased cluster connectivity. Although ChainMap solutions typically use fewer resources (i.e. nets, clusters) and have lower routing cost, they tend to possess higher total routing utilization which erodes potential performance gains. The primary culprit is ChainMap itself; it has the tendency to over-pack LUTs, resulting in higher LUT fanout and higher LUT fanin, which it passes on throughout the entire design flow.

Another side effect of over-packing LUTs is the area paradox. Overpopulated LUTs have difficulty forming LEs because they often violate the limit on total number of distinct inputs outlined in the exclusivity constraint (Lemma 5.4.1). Overpopulated LUTs have to be implemented in their own LE, causing a decrease in the number of LUTs to become an increase in LEs, eventually leading to an increase in the number of clusters. One way to combat the area paradox is to create a more intelligent version of ChainPack, wherein techniques such as Roth-Karp decomposition can be employed to force LUTs to conform to the arithmetic LE architectural model and provide a pathway for LUT reduction to extend to LE reduction. Another solution to LUT over-packing is to choose cuts during ChainMap's execution that reduce implicit LUT duplication. This problem is common in technology mapping has been found to be NP-hard [30]. Avenues for improvement involve extending the basic ChainMap algorithm to incorporate an arbitrary net-delay model, re-synthesis and re-timing, or adapt it to a cut-enumeration technique. Other technology mapping algorithms such as [17] incorporate cut-enumeration, successive iterations of FlowMap use re-synthesis [24] to minimize area, and re-timing is used in ABC [59]. Such techniques are also readily applied to ChainMap. Future work includes a cut-enumeration version of ChainMap with the ability to select cuts that have the efficacy to reduce duplications and over-packing of LUTs, thus reducing cluster connectivity and improving overall performance.

The relaxation techniques applied to optimal ChainMap solutions is another candidate for improvement. Global least critical relaxation and the branch trimming approaches specifically target chains that are structured similar to arithmetic chains by relaxing connections based on a simple delay model. Because it targets arithmetic chains, its application to designs with a multitude of small chains is, at times, ineffective. A more exact timing model coupled with routability and area based relaxation have the potential to drastically improve relaxed solutions. The presented relaxation techniques could also be applied to HDL chains to reduce the constraints they impose on place and route.

HierARC does a superb job of reducing the routing cost of a design. However, in its current implementation it only targets routing cost and consequently yields delay-neutral results. The Euclidian gain function it incorporates is inherently capable of accommodating a multitude of performance metrics. Future work is to include metrics such as timing, power consumption, and fault tolerance. Another potential improvement to HierARC is to make it *less* effective at eliminating small fanout nets, and incorporate constraints to limit cluster packing. Increased cluster connectivity has been shown to degrade performance, and constraining cluster packing could reduce contention for routing resources as per [71]. This applies directly to ChainMap solutions, which commonly suffer from cluster over-connectivity. Finally, a full exploration of how to cluster chains effectively is necessary to precipitate better placement and routing of chains.

The placement techniques used by VPR minimize net and path timing, and, coupled with the technique presented in [8], provide a method for dealing with chains. However, they also provide opportunities for great improvement. The net and path driven placement techniques of VPR already capture the timing impact of chains well, but were not originally designed with them in mind and can therefore be further tuned to their unique characteristics. Future work includes place and route experimentation that draws from other disciplines, just as HierARC draws upon biology. Clustering and placement can be performed using other DNA microarray techniques like K-means, machine learning approaches such as self organizing maps [13], or game theory for peer-to-peer networks [40].

A radical approach to harnessing the potential performance of ChainMap is to design FPGA architectures that deviate from the arithmetic LE chain model. Chapter 3 presents a reuse cell that is capable of operating in both sub-width and full-width LE modes. However, abandoning the traditional LE and/or 1-dimensional chain interconnection model(s) may prove to be better options. The traditional arithmetic LE model requires two LUTs, the *sum* and *cout*, to implement members of a chain. To do so, extra logic and memory are typically required to support independent *sum* and *cout* outputs, define carry input multiplexing, support dynamic addition/subtraction, and dictate normal or arithmetic operation mode. A valid alternative LE designed specifically for use with generic logic chains only requires a single K-LUT that outputs the same operation on both the *cout* and *sum* ports at all times (i.e. only operates in K-LUT mode). Furthermore, a 1-dimensional chain only provides chain net connectivity between adjacent cells, and necessitates the relaxation and duplication phases of ChainMap. An FPGA architecture supporting 2-dimensional chain trees could potentially reduce or eliminate the need for these phases, and more directly support optimal ChainMap solutions.

8.4 Conclusion

ChainMap is not the completion of a work, but rather the beginning of a body of work whose goal is to fully exploit the performance of optimal technology map solutions. Several areas for future work have already been proposed, including chain-aware synthesis, cut-enumeration for chains, more effective relaxation, development of novel architectures, and improved clustering, placement, routing. The goal of all of these is to unleash the potential for generic logic chains that technology mapping experiments have indicated is possible.

Area-constrained and unconstrained experiments divulge the effective range of ChainMap. When cluster resources are plentiful, ChainMap designs suffer from density and routing congestion. HDL dictated arithmetic chains tend to spread out the design and reduce congestion. However, if cluster resources are restricted, the shorter, less numerous chains produced by ChainMap become an asset. With fewer inter-cluster constraints to be honored, ChainMap designs are able to produce more effective placement solutions, ultimately yielding higher performance routed solutions.

Area-constrained and unconstrained architectures represent opposite ends of the product life cycle. Initially, FPGAs with vast cluster resources are used to develop systems and serve as prototypes. During production, smaller FPGAs are chosen to reduce system cost and preserve resources like area and power. Additionally, multiple-IP designs can benefit as they typically impose cluster constraints in an effort to preserve the high efficiency of mature cores. As FPGAs increasingly become part of production designs, ChainMap provides an avenue to exploit every available resource they possess.

ChainMap has shown the ability to identify optimal chain implementation without the use of HDL macros, positioning it as a replacement to the traditional approach to arithmetic operations, and enabling the creation of innovative architectures and tools that don't enforce artificial partitions. By rethinking technology mapping as an exercise in the minimization of routing depth rather than logic depth, ChainMap portends performance gains for all designs. It realizes these performance gains under resource-constrained architectures. Arithmetic HDL macros can be discarded in favor of allowing the CAD flow to decide when and where logic chains should be created in a Boolean network. With this approach, both FPGA hardware and computer aided design can move beyond the arithmetic constraint, and start considering all chains as having been created equal.

APPENDIX
Architectural Description

VPR architectural parameters used to perform full place and route experiments with ChainMap. The general routing array is described in Table A.3, and characterized by the segment length, frequency, and switch boxes. Table A.4 provides timing parameters for each individual logic element. Finally, Table A.5 gives the interface each logic element has with its inter-cluster input and output ports, and intra-cluster interconnection between logic elements.

Table A.3 VPR Routing Architectural Parameters

Component	Parameter
Single Line	freq = 22%, Rmetal = 4.16Ω, Cmetal: 81e-15F
Double Line	freq = 28%, Rmetal = 4.16Ω, Cmetal: 81e-15F
Quad Line	freq = 42%, Rmetal = 4.16Ω, Cmetal: 81e-15F
Long Line	freq = 8%, Rmetal = 4.16Ω, Cmetal: 81e-15F
Switch Box	Subset
Switch 0	R = 196.728Ω, Cin = 20.574e-15F, Cout = 20.574e-15F, Tdel = 0s
Switch 1	R = 393.47Ω, Cin = 7.512e-15F, Cout = 20.574e-15F, Tdel = 524e-12s, buffered
Switch 2	R = 786.9Ω, Cin = 7.512e-15F, Cout = 10.762e-15F, Tdel = 456e-12s, tristate

Table A.4 VPR LE Architectural Parameters

Timing parameter	$K = 4$	$K = 5$	$K = 6$
T_{comb}	1.902e-10s	2.567e-10s	3.002e-10s
T_{seq_in}	1.692e-10s	2.359e-10s	2.849e-10s
T_{seq_out}	9.585e-11	9.516000e-11	9.765e-11s
T_{seq_async}	9.585e-11	9.516000e-11	9.765e-11s
T_{seq_sync}	6.700e-12	6.900e-12	1.240e-11s
T_{cout}	2.770e-11	2.770e-11	2.770e-11s

Table A.5 VPR Component Parameters

Timing parameter	$K = 4$	$K = 5$	$K = 6$
C_ipin_cblock	0	0	0
T_ipin_cblock	9.279e-11	9.289e-11	8.855e-11
T_ipad	5.616e-11	5.657e-11	5.667e-11
T_opad	1.835e-11	1.913e-11	1.919e-11
T_sblk_cout_to_clb_cout	0	0	0
T_clb_cin_to_sblk_cin	0	0	0
T_sblk_cout_to_sblk_cin	0	0	0
T_clb_cout_to_clb_cin	0	0	0
T_sblk_opin_to_sblk_ipin	6.079e-11	6.141e-11	6.258e-11
T_clb_ipin_to_sblk_ipin	6.129e-11	5.996e-11	6.086e-11
T_sblk_opin_to_clb_opin	0	0	0

Bibliography

[1] Actel. *Actel FPGAs*. http://www.actel.com.

[2] Actel. *QuickLogic FPGAs*. http://www.quicklogic.com.

[3] A. Aggarwal and D. Lewis. Routing architectures for hierarchical field programmable gatearrays. *Computer Design: VLSI in Computers and Processors, 1994. ICCD'94. Proceedings., IEEE International Conference on*, pages 475–478, 1994.

[4] E. Ahmed and J. Rose. The effect of lut and cluster size on deep-submicron fpga performance and density. *IEEE Transactions on VLSI Systems*, 12(3):288–298, March 2004.

[5] Altera. *Quartus II Handbook*. www.altera.com.

[6] Altera. *Stratix Series User Guides*. www.altera.com.

[7] F. Barat, R. Lauwereins, and G. Deconinck. Reconfigurable instruction set processors from a hardware/software perspective. *IEEE Transactions on Software Engineering*, 28(9):847–862, September 2002.

[8] M. J. Beauchamp, S. Hauck, K. D. Underwood, and K. S. Hemmert. Embedded floating-point units in fpgas. In *FPGA '06: Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, pages 12–20, New York, NY, USA, 2006. ACM Press.

[9] L. Benini and G. D. Micheli. A survey of boolean matching techniques for library binding. *ACM Trans. Des. Autom. Electron. Syst.*, 2(3):193–226, 1997.

[10] V. Betz and J. Rose. Using architectural families to increase fpga speed and density. In *FPGA '95: Proceedings of the 1995 ACM third international symposium on Field-programmable gate arrays*, pages 10–16, New York, NY, USA, 1995. ACM Press.

[11] V. Betz and J. Rose. Cluster-based logic blocks for fpgas: area-efficiency vs. input sharing and size. In *Proceedings of the IEEE 1997 Custom Integrated Circuits Conference*, pages 551–554, May 1997.

[12] V. Betz, J. Rose, and A. Marquardt. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer, Norwell, MA, 1999.

[13] D. C. Blight and R. D. McLeod. Self-organizing kohonen maps for fpga placement. *Field-Programmable Gate Arrays: Architecture and Tools for Rapid Prototyping*, 705:88–95, 1993.

[14] E. Bozorgzadeh, S. Ogrenci-Memik, and M. Sarrafzadeh. Rpack: routability-driven packing for cluster-based fpgas. In *ASP-DAC '01: Proceedings of the 2001 conference on Asia South Pacific design automation*, pages 629–634, New York, NY, USA, 2001. ACM Press.

[15] E. Bozorgzadeh, S. Ogrenci-Memik, and M. Sarrafzadeh. Routability-driven packing: Metrics and algorithms for cluster-based fpgas. *Journal of Circuits Systems and Computers*, 13(1):77–100, 2004.

[16] A. Caldwell, A. Kahng, S. Mantik, I. Markov, and A. Zelikovsky. On wirelength estimations for row-based placement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(9):1265–1278, 1999.

[17] D. Chen and J. Cong. DAOmap: a depth-optimal area optimization mapping algorithm for FPGA designs. In *IEEE/ACM International Conference on Computer Aided Design*, pages 752–759, 2004.

[18] K. Chen, J. Cong, Y. Ding, A. Kahng, and P. Trajmar. DAG-Map: graph-based FPGA technology mapping for delay optimization. *IEEE Design and Testing of Computers*, 9(3):7–20, 1992.

[19] K. Chung. Architecture and synthesis of field-programmable gate arrays with hard-wired connections. *Phd, University of Toronto*, 1995.

[20] J. Cong and Y. Ding. FlowMap: an optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(1):1–12, 1994.

[21] J. Cong, Y. Ding, T. Gao, and K. Chen. LUT-based FPGA technology mapping under arbitrary net-delay models. *Computers & Graphics(Pergamon)*, 18(4):507–516, 1994.

[22] J. Cong and Y.-Y. Hwang. Simultaneous depth and area minimization in lut-based fpga mapping. In *FPGA '95: Proceedings of the 1995 ACM third international symposium on Field-programmable gate arrays*, pages 68–74, New York, NY, USA, 1995. ACM Press.

[23] J. Cong and K. Minkovich. Optimality study of logic synthesis for lut-based fpgas. *IEEE Transactions On Computer-Aided Design Of Integrated Circuits And Systems*, 26(2):230–239, 2007.

[24] J. Cong, J. Peck, and Y. Ding. Rasp: a general logic synthesis system for sram-based fpgas. In *FPGA '96: Proceedings of the 1996 ACM fourth international symposium on Field-programmable gate arrays*, pages 137–143, New York, NY, USA, 1996. ACM Press.

[25] J. Cong, C. Wu, and Y. Ding. Cut ranking and pruning: enabling a general and efficient fpga mapping solution. In *FPGA '99: Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, pages 29–35, New York, NY, USA, 1999. ACM.

[26] T. Cormen, C. Leiserson, and R. Rivest. *Algorithms.* MIT Press, Cambridge, MA, 1990.

[27] A. DeHon. *Reconfigurable architectures for general-purpose computing.* Ph.D. dissertation, Massachusetts Institute of Technology, 1996.

[28] A. DeHon. Balancing interconnect and computation in a reconfigurable computing array (or, why you don't really want 100% lut utilization). In *FPGA '99: Proceedings of the*

1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays, pages 69–78, New York, NY, USA, 1999. ACM.

[29] C. Ebeling, L. McMurchie, S. Hauck, and S. Burns. Placement and routing tools for the triptych fpga. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 3(4):473–482, 1995.

[30] A. Farrahi and M. Sarrafzadeh. Complexity of the lookup-table minimization problem for fpga technology mapping. *IEEE Transactions On Computer-Aided Design Of Integrated Circuits And Systems*, 13(11):1319–1332, 1994.

[31] A. Farrahi and M. Sarrafzadeh. Fpga technology mapping forpower minimization. In *Proceedings of International Workshop in Field Programmable Logic and Applications*, 1994.

[32] L. R. Ford and D. R. Fulkerson. *Flows in Networks.* Princeton Univ. Press, Princeton, NJ, 1962.

[33] R. Francis. *Technology Mapping for Lookup-Table Based Field-Programmable Gate Arrays.* PhD dissertation, University of Toronto, 1993.

[34] R. Francis, J. Rose, and Z. Vranesic. Chortle-crf: fast technology mapping for lookup table-based FPGAs. In *28th ACM/IEEE Design Automation Conference*, pages 227–233, 1991.

[35] R. Francis, J. Rose, and Z. Vranesic. Technology mapping of lookup table-based FPGAs for performance. In *Digest of Technical Papers for IEEE International Conference on Computer-Aided Design*, pages 568–571, 1991.

[36] M. Frederick and A. Somani. Non-arithmetic carry chains for reconfigurable fabrics. In *Proceedings of the 15th International Conference on Computer Design*, pages 137–143, October 2007.

[37] M. T. Frederick and A. K. Somani. Multi-bit carry chains for high-performance reconfigurable fabrics. In *Proceedings of 16th International Conference on Field Programmable Logic and Applications*, pages 275–280, August 2006.

[38] M. T. Frederick and A. K. Somani. Beyond the arithmetic constraint: depth-optimal mapping of logic chains in reconfigurable fabrics. In *Proceedings of the Sixteenth ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 37–46, February 2008.

[39] M. T. Frederick and A. K. Somani. Hierarchical agglomerative clustering for island-style fpgas. In *Proceedings of 18th International Conference on Field Programmable Logic and Applications*, submmitted 2008.

[40] R. Gupta, V. Sekhri, and A. Somani. CompuP2P: An Architecture for Internet Computing Using Peer-to-Peer Networks. *IEEE Transactions on Parallel and Distributed Systems*, 17(11):1306–1320, 2006.

[41] S. Hauck, M. M. Hosler, and T. W. Fry. High performance carry chains for fpgas. *IEEE Transactions on VLSI Systems*, 8(2), April 2000.

[42] M. Huang, F. Romeo, and A. Sangiovanni-Vincentelli. An efficient general cooling schedule for simulated annealing. *IEEE International Conference on Computer Aided Design*, pages 381–384, 1986.

[43] M. Hutton, J. Schleicher, D. Lewis, B. Pedersen, R. Yuan, S. Kaptanoglu, G. Baeckler, B. Ratchev, K. Padalia, M. Bourgeault, A. Lee, H. Kim, and R. Saini. Improving FPGA Performance and Area Using an Adaptive Logic Module. *Proc. Int'l Conference on Field Programmable logic and its applications Proc. FPL-04*, pages 135–144, 2004.

[44] A. Jain, M. Murty, and P. Flynn. Data clustering: a review. *ACM Computing Surveys*, 31(3):264–323, 1999.

[45] G. Karypis and V. Kumar. Multilevel k-way hypergraph partitioning. *VLSI Design*, 11(3):285–300, 2000.

[46] K. Keutzer. Dagon: technology binding and local optimization by dag matching. In *DAC '87: Proceedings of the 24th ACM/IEEE conference on Design automation*, pages 341–347, New York, NY, USA, 1987. ACM Press.

[47] M. Kobata, M. Iida, and T. Sueyoshi. Clustering technique to reduce chip area and delay for fpga. *Electronics and Communications in Japan (Part II: Electronics)*, 90(6):34–46, 2007.

[48] I. Kuon and J. Rose. Measuring the gap between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26:203–215, February 2007.

[49] I. Kuon and J. Rose. Area and delay trade-offs in the circuit and architecture design of fpgas. In *Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, pages 149–158, 2008.

[50] Y.-T. Lai and P.-T. Wang. Hierarchical interconnection structures for field programmable gate arrays. *IEEE Trans. Very Large Scale Integr. Syst.*, 5(2):186–196, 1997.

[51] J. Lam and J. Delosme. Performance of a new annealing schedule. *Design Automation Conference, 1988. Proceedings., 25th ACM/IEEE*, pages 306–311, 1988.

[52] E. L. Lawler, K. N. Levitt, and J. Turner. Module clustering to minimize delay in digital networks. *IEEE Trans. Comput.*, 18(1):47–57, 1969.

[53] K. Leijten-Nowak and J. L. van Meerbergen. An fpga architecture with enhanced datapath functionality. In *Proceedings of the 11th Int'l Symposium on FPGAs*, pages 195–204, 2003.

[54] A. Ling, D. P. Singh, and S. D. Brown. Fpga technology mapping: a study of optimality. In *Proceedings of the 42nd annual conference on Design automation*, pages 427–432, New York, NY, USA, 2005. ACM Press.

[55] S. Malhotra, T. Borer, D. Singh, and S. Brown. The quartus university interface program: enabling advanced fpga research. In *Proceedings of the 2004 IEEE Int'l Conference on Field-Programmable Technology*, pages 225–230, Dec. 2004.

[56] V. Manohararajah, S. Brown, and Z. Vranesic. Heuristics for area minimization in LUT-based FPGA technology mapping. In *Proceedings of the International Workshop on Logic and Synthesis*, pages 14–21, 2004.

[57] A. Marquardt, V. Betz, and J. Rose. Speed and area tradeoffs in cluster-based fpga architectures. *IEEE Trans. Very Large Scale Integr. Syst.*, 8(1):84–93, 2000.

[58] Z. Marrakchi, H. Mrabet, and H. Mehrez. Hierarchical fpga clustering based on multilevel partitioning approach to improve routability and reduce power dissipation. In *Proceedings of International Conference on Reconfigurable Computing and FPGAs*, 2005.

[59] A. Mishchenko. *An Integrated Technology Mapping Environment.* http://www.eecs.berkeley.edu/ alanmi/abc/.

[60] A. Mishchenko, S. Chatterjee, R. Brayton, and P. Pan. Integrating logic synthesis, technology mapping, and retiming. In *Proceedings of the International Workshop on Logic and Synthesis*, pages 177–181, 2005.

[61] R. Murgai, N. Shenoy, R. Brayton, and A. Sangiovanni-Vincentelli. Performance directed synthesis for table look up programmable gate arrays. In *Digest of Technical Papers from the IEEE International Conference on Computer-Aided Design*, pages 572–575, 1991.

[62] OpenCores. www.opencores.org.

[63] J. Rose, R. J. Francis, D. Lewis, and P. Chow. Architecture of field-programmable gate arrays: the effect of logic block functionality on area efficiency. *IEEE Journal of Solid-State Circuits*, 25:1217–1225, October 1990.

[64] J. Rose and D. Hill. Architectural and physical design challenges for one-million gate fpgas and beyond. In *FPGA '97: Proceedings of the 1997 ACM fifth international symposium on Field-programmable gate arrays*, pages 129–132, New York, NY, USA, 1997. ACM.

[65] S. Safarpour, A. Veneris, G. Baeckler, and R. Yuan. Efficient sat-based boolean matching for fpga technology mapping. In *DAC '06: Proceedings of the 43rd annual conference on Design automation*, pages 466–471, New York, NY, USA, 2006. ACM.

[66] E. Sentovich, K. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Sis: A system for sequential circuit synthesis. Technical Report UCB/ERL M92/41, EECS Department, University of California, Berkeley, 1992.

[67] A. Singh, G. Parthasarathy, and M. Marek-Sadowska. Efficient circuit clustering for area and power reduction in fpgas. *ACM Trans. Des. Autom. Electron. Syst.*, 7(4):643–663, 2002.

[68] S. Singh, J. Rose, P. Chow, and D. Lewis. The effect of logic block architecture on fpga performance. *Journal of Solid-State Circuits*, 27:281–287, March 1992.

[69] W. Swartz and C. Sechen. New algorithms for the placement and routing of macro cells. *Computer-Aided Design, 1990. ICCAD-90. Digest of Technical Papers., 1990 IEEE International Conference on*, pages 336–339, 1990.

[70] M. Teslenko and E. Dubrova. Hermes: LUT FPGA technology mapping algorithm for area minimization with optimum depth. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design*, pages 748–751, 2004.

[71] M. Tom and G. Lemieux. Logic block clustering of large designs for channel-width constrained fpgas. In *Proceedings of 42nd conference on design automation*, pages 726–731, 2005.

[72] N. Viswanathan and C. Chu. Fastplace: efficient analytical placement using cell shifting, iterative local refinement,and a hybrid net model. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(5):722– 733, May 2005.

[73] M. Wu, Y.-L.; Marek-Sadowska. Graph based analysis of fpga routing. In *Design Automation Conference, 1993, with EURO-VHDL '93. Proceedings EURO-DAC '93. European*, pages 104–109, September 1993.

[74] Xilinx. *Virtex Series User Guides.* www.xilinx.com.

[75] Xilinx. *XC4000E and XC4000X Series Field Programmable Gate Arrays Product Specification.* http://www.xilinx.com.

[76] S. Yang. Logic synthesis and optimization benchmarks, version 3.0. *Tech. Report, Microelectronics Centre of North carolina*, 1991.

[77] P. S. Zuchowski, C. B. Reynolds, R. J. Grupp, S. G. Davis, B. Cremen, and B. Troxel. Hybrid asic and fpga architecture. In *Proceedings of the International Conference on Computer-Aided Design*, pages 187–194, 2002.