ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Πρωτότυπος Αλγόριθμος Επίλυσης του Προβλήματος της Τοποθέτησης σε Επαναδιαμορφούμενες Αρχιτεκτονικές Τριών Διαστάσεων με χρήση Αλγορίθμων Αποικιών Μυρμηγκιών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

ΠΑΝΑΓΙΩΤΗ Β. ΔΑΝΑΣΗ

Επιβλέπων : Δημήτριος Σούντρης
Αναπληρωτής Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούνιος 2015

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Πρωτότυπος Αλγόριθμος Επίλυσης του Προβλήματος της Τοποθέτησης σε Επαναδιαμορφούμενες Αρχιτεκτονικές Τριών Διαστάσεων με χρήση Αλγορίθμων Αποικιών Μυρμηγκιών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

ΠΑΝΑΓΙΩΤΗ Β. ΔΑΝΑΣΗ

Επιβλέπων : Δημήτριος Σούντρης
Αναπληρωτής Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 11η Ιουνίου 2015.

.....	.....	.....
Δημήτριος Σούντρης Αναπλ. Καθηγητής Ε.Μ.Π.	Κιαμάλ Πεκμεστζή Καθηγητής Ε.Μ.Π.	Γιώργος Οικονομάκος Επίκ. Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούνιος 2015

.....

ΠΑΝΑΓΙΩΤΗΣ Β. ΔΑΝΑΣΗΣ

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

Περίληψη

Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Δίπλωμα Ηλεκτρολόγου Μηχανικού και Μηχανικού Υπολογιστών

Πρωτότυπος Αλγόριθμος Επίλυσης του Προβλήματος της Τοποθέτησης σε Επαναδιαμορφούμενες Αρχιτεκτονικές Τριών Διαστάσεων με χρήση Αλγορίθμων Αποικιών Μυρμηγκιών

από τον Παναγιώτη Δανασή

Για περισσότερο από 30 χρόνια τώρα, από το 1984 που έκαναν πρώτη φορά την εμφάνισή τους, οι Επαναδιαμορφούμενες Αρχιτεκτονικές (FPGA) αποτελούν ένα από τα δημοφιλέστερα μέσα υλοποίησης ψηφιακών κυκλωμάτων. Συνδυάζοντας τις υψηλές επιδώσεις των ενσωματωμένων κυκλωμάτων ειδικού σκοπού (ASIC) με την ευελιξία των επεξεργαστών γενικού σκοπού (CPU) έχουν καταστήσει δυνατή την υλοποίηση εντελώς καινοτόμων εφαρμογών. Ένα σημαντικό μέρος της επιτυχίας τους οφείλεται στην αρχιτεκτονική τους, η οποία παντρεύει συστοιχίες προγραμματιζόμενης λογικής με ένα δίκτυο προγραμματιζόμενης διασύνδεσης, παρέχοντας ευελιξία και δυνατότητα ταχείας ανάπτυξης εφαρμογών. Σε σύγκριση με άλλες εναλλακτικές τεχνολογίες, τα FPGA διαθέτουν ένα πλήθος πλεονεκτημάτων όπως:

- γρήγορο χρόνο διάθεσης στην αγορά,
- απουσία μη-επαναλαμβανόμενων δαπανών μελέτης για την κατασκευή (NRE)
- παροχή στους σχεδιαστές κυκλωμάτων υλικό που είναι εκ των προτέρων δοκιμασμένο,
- δυνατότητα επαναπρογραμματισμού.

Στην προσπάθεια να ξεπεραστούν οι περιορισμοί που προκύπτουν από την συνεχόμενη συρρίκνωση του μεγέθους των σημερινών τρανζίστορ, αναπτύχθηκε μία επαναστατική μέθοδος ολοκλήρωσης τρανζίστορ σε τρεις διαστάσεις (three dimensional (3-D) chip stacking). Η μέθοδος αυτή υπόσχεται βελτιωμένη απόδοση, μείωση της κατανάλωσης ισχύος και μικρότερο κόστος σε σχέση με τις συμβατικές μεθόδους ολοκλήρωσης δύο διαστάσεων (2-D).

Ο σχεδιασμός σε τέτοιες τρισδιάστατες πλατφόρμες αποτελεί ένα πολύ σύνθετο

πρόβλημα. Γι αυτόν τον λόγο υπάρχει μία διαρκώς αυξανόμενη ανάγκη για αποδοτικά εργαλεία σχεδίασης με χρήση υπολογιστή (Computer-Aided Design CAD tools). Τα εργαλεία σχεδίασης επηρεάζουν σε μεγάλο βαθμό την επίδοση της τελικής υλοποίησης στο FPGA. Η επιπλέον πολυπλοκότητα που προσδίδει ο σχεδιασμός σε τρεις διαστάσεις σε συνδυασμό με τον διπλασιασμό της χωρητικότητας των FPGA ανά δύο με τρία χρόνια (σε εναρμόνιση με τον νόμο του Μουρ) κάνουν όλο και πιο έντονη την ανάγκη καινοτόμων αλγορίθμων σχεδίασης που θα μπορούν να εκμεταλλεύονται την σύγχρονη τάση προς τους πολυπύρηνους επεξεργαστές για να παράγουν υψηλής ποιότητας αποτελέσματα σε εύλογο χρονικό διάστημα.

Από την ροή σχεδίασης μιας εφαρμογής σε ένα FPGA, το πρόβλημα της τοποθέτησης (placement) θεωρείται το πιο χρονοβόρο, ενώ ταυτόχρονα η ποιότητα της παραγόμενης λύσης επηρεάζει σε μεγάλο βαθμό την μέγιστη συχνότητα λειτουργίας. Το πρόβλημα αυτό γίνεται ακόμα πιο έντονο στις αρχιτεκτονικές τριών διαστάσεων. Για την αντιμετώπιση αυτού του προβλήματος παρουσιάζουμε ένα καινοτόμο αλγόριθμο, βασιζόμενο στους αλγορίθμους Αποικιών Μυρμηγκιών (Ant Colony Optimization).

Οι αλγόριθμοι Αποικιών Μυρμηγκιών και γενικότερα οι αλγόριθμοι νοημοσύνης σμήνους (swarm intelligence) είναι κατανεμημένα συστήματα όπου, σε αντιδιαστολή με την απλότητα των πρακτόρων τους, παρουσιάζουν μία εξαιρετικά δομημένη κοινωνική οργάνωση και ως εκ τούτου έχουν την δυνατότητα να επιτύχουν σύνθετες λειτουργίες, κάνοντας χρήση της συλλογικής νοημοσύνης της κοινωνίας. Ο τομέας των αλγορίθμων αποικιών μυρμηγκιών προήλθε από την παρατήρηση της συμπεριφοράς ορισμένων ειδών πραγματικών μυρμηγκιών. Μία από τις πιο επιτυχημένες κατηγορίες αλγορίθμων, βασισμένους στις αποικίες μυρμηγκιών, είναι ο Ant Colony Optimization (ACO). Στους αλγορίθμους ACO, μία ομάδα πρακτόρων – τεχνητών μυρμηγκιών, ψάχνει για καλές λύσεις σε κάποιο δοσμένο πρόβλημα συνδυαστικής βελτιστοποίησης. Οι πράκτορες αυτοί μετακινούνται στον γράφο του προβλήματος, κτίζοντας βαθμιαία λύσεις, κάνοντας χρήση ενός στοχαστικού κανόνα ο οποίος επηρεάζεται από ένα μοντέλο έμμεσης επικοινωνίας. Η έμμεση αυτή επικοινωνία γίνεται με την κατάθεση φερομόνης (pheromone) από τα μυρμήγκια, τροποποιώντας με αυτόν τον τρόπο την αντίληψη του προβλήματος από τα άλλα μυρμήγκια.

Σε αυτήν την διπλωματική εργασία παρουσιάζουμε έναν καινοτόμο αλγόριθμο επίλυσης του προβλήματος της τοποθέτησης σε επαναδιαμορφούμενες αρχιτεκτονικές τριών διαστάσεων, κάνοντας χρήση των αλγορίθμων αποικιών μυρμηγκιών. Ο αλγόριθμος μας ενσωματώνει στοιχεία από τους MAX-MIN Ant System (MMAS) και Ant Colony System (ACS), οι οποίοι αποτελούν τους δύο πιο αποδοτικούς αλγορίθμους της οικογένειας

αλγορίθμων αποικιών μυρμηγκιών. Ο προτεινόμενος αλγόριθμος παρουσιάζει πολλά πλεονεκτήματα σε σχέση με υπάρχουσες προσεγγίσεις, όπως:

- εγγενή παραλληλισμό,
- άμεση εφαρμογή περιορισμών στην συνάρτηση κόστους,
- υποστήριξη ετερογενών αρχιτεκτονικών,
- ανοικτός πηγαίος κώδικας.

Πειραματικά αποτελέσματα αποδεικνύουν την αποτελεσματικότητα του προτεινόμενου αλγόριθμου, καθώς επιτυγχάνει κατά μέσο όρο 10% μείωση στο critical path delay. Κατά συνέπεια έχουμε βελτίωση στην μέγιστη συχνότητα λειτουργίας καθώς και μείωση στην κατανάλωση ενέργειας. Επιπροσθέτως ο αλγόριθμός μας πετυχαίνει επιτάχυνση (speedup) σε πολυπύρηνες αρχιτεκτονικές πολύ κοντά στην θεωρητικά μέγιστη. Αυτό σημαίνει ότι έχει την δυνατότητα να εκμεταλλευτεί πλήρως τους σύγχρονους πολυπύρηνους επεξεργαστές.

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Τρισδιάστατες Αρχιτεκτονικές (3Δ), Ετερογενείς Αρχιτεκτονικές, Επαναδιαμορφούμενες Αρχιτεκτονικές, FPGA, Εργαλεία Σχεδίασης με χρήση Υπολογιστή, CAD, Πρόβλημα της Τοποθέτησης, Placement, Παράλληλοι Αλγόριθμοι, Αλγόριθμοι Αποικιών Μυρμηγκιών, Ant Colony Optimization (ACO), Ant Colony System (ACS), MAX-MIN Ant System (MMAS).

National Technical University of Athens

Diploma Thesis

A Novel 3-D FPGA Placement Algorithm based on Ant Colony Optimization

Author:
Panayiotis Danassis

Supervisor:
Associate Prof. Dimitrios Soudris

A thesis submitted in fulfilment of the requirements
for the degree of Diploma in Electrical and Computer Engineering

in the

Division of Computer Science
School of Electrical and Computer Engineering

June 2015

"Poets say science takes away from the beauty of the stars – mere globs of gas atoms. Nothing is "mere". I too can see the stars on a desert night, and feel them. But do I see less or more? The vastness of the heavens stretches my imagination – stuck on this carousel my little eye can catch one million year old light... What is the pattern, or the meaning, or the why? It does not do harm to the mystery to know a little about it. For far more marvelous is the truth than any artists of the past imagined! Why do the poets of the present not speak of it? What men are poets who can speak of Jupiter as if he were like a man, but if he is an immense spinning sphere of methane and ammonia must be silent?"

Richard P. Feynman

NATIONAL TECHNICAL UNIVERSITY OF ATHENS

Abstract

Division of Computer Science
School of Electrical and Computer Engineering

Diploma in Electrical and Computer Engineering

A Novel 3-D FPGA Placement Algorithm based on Ant Colony Optimization

by Panayiotis Danassis

Placement is considered one of the most arduous and time-consuming processes in physical implementation flows for reconfigurable architectures, while it highly affects the quality of derived application implementation as it has impact on the maximum operating frequency. This problem becomes more acute for three-dimensional (3-D) architectures, because the complexity of these architectures imposes additional challenges that have to be sufficiently addressed. Throughout this thesis we introduced a novel placement algorithm, targeting 3-D reconfigurable architectures, based on Ant Colony Metaheuristics. Ant colonies are distributed systems that, in spite of the simplicity of their individuals, present a highly structured social organization and as a result can accomplish complex tasks using the collective intelligence of the group. One of the most successful examples of ant based algorithms is known as Ant Colony Optimization (ACO). ACO is inspired by the foraging behavior of ants. Our proposed algorithm incorporates concepts from both $MAX-MIN$ Ant System and Ant Colony System, the two best performing algorithms of the ACO family. It exhibits numerous advantages, such as inherent parallelism, direct enforcement of legality constrains into the cost function and support of heterogeneous architectures. Experimental results validate the effectiveness of our algorithm since it achieves on average 10% reduction on the critical path delay. This results to designs with increased maximum operating frequency and reduced power consumption. Additionally our placer can achieve speedup in multi-core architectures very close to the theoretical one. This means that our proposed algorithm can take full advantage of todays multi-core CPUs, further decreasing the execution run-time.

KEYWORDS: Three-Dimensional (3-D) Reconfigurable Architectures, Heterogeneous Architectures, FPGA, CAD, Placement, Parallel Algorithms, Ant Colony Optimization, Ant Colony System (ACS), Max-Min Ant System (MMAS)

Acknowledgements

At the end of my thesis I would like to take a moment to thank all the people who made this thesis possible and an unforgettable experience for me.

At first I offer my sincerest gratitude to my supervisor, Prof. Dimitrios Soudris who has been supportive since the day I began working in the lab and offered his continuous advice and encouragement throughout the course of this thesis.

Also I would like to express my deepest gratitude to my advisor, Dr. Kostas Siozios, who has supported me throughout my thesis with his excellent guidance, patience and knowledge whilst allowing me the room to work in my own way.

I am thankful to all my fellow labmates in the Microprocessors Laboratory and Digital Systems Lab (MicroLab) for providing me with an excellent and fun atmosphere for doing research and the means to achieve my goal.

Beyond the academic sphere, I would like to thank my friends for always being by my side and for making my years of study a wonderful adventure.

Finally, I take this opportunity to express my genuine gratitude from the bottom of my heart to my beloved parents, Vassili and Eva, and brother Costas for their love and continuous support, both spiritually and materially.

Contents

List of Figures

List of Tables

Abbreviations

ACO	**A**nt **C**olony **O**ptimization
ACS	**A**nt **C**olony **S**ystem
ANTS	**A**pproximate **N**ondeterministic **T**ree **S**earch
AS	**A**nt **S**ystem
ASIC	**A**pplication-**S**pecific **I**ntegrated **C**ircuit
BB	**B**ranch and **B**ound
CAD	**C**omputer-**A**ided **D**esign
CPLD	**C**omplex **P**rogrammable **L**ogic **D**evice
CPU	**C**entral **P**rocessing **U**nit
DSP	**D**igital **S**ignal **P**rocessor
EAS	**E**litist **A**nt **S**ystem
EDA	**E**lectronic **D**esign **A**utomation
EPROM	**E**rasable **P**rogrammable **R**ead-**O**nly **M**emory
FPGA	**F**ield-**P**rogrammable **G**ate **A**rray
GPU	**G**raphics **P**rocessing **U**nit
HDL	**H**ardware **D**escription **L**anguage
IC	**I**ntegrated **C**ircuit
ILP	**I**nstruction-**L**evel **P**arallelism
LUT	**L**ook-**U**p **T**able
MMAS	**M**ax-**M**in **A**nt **S**ystem
NRE	**N**on-**R**ecurring **E**ngineering costs
OpenMP	**O**pen **M**ulti-**P**rocessing
OTP	**O**ne-**T**ime **P**rogrammable
PAPI	**P**artially **A**synchronous **P**arallel **I**mplementation
PLD	**P**rogrammable **L**ogic **D**evice

PROM	**P**rogrammable **R**ead-**O**nly **M**emory
RAM	**R**andom-**A**ccess **M**emory
ROM	**R**ead-**O**nly **M**emory
RTL	**R**egister-**T**ransfer **L**evel
SB	**S**witch **B**ox
SI	**S**warm **I**ntelligence
SIMD	**S**ingle **I**nstruction, **M**ultiple **D**ata
SRAM	**S**tatic **R**andom-**A**ccess **M**emory
SSIT	**S**tacked **S**ilicon **I**nterconnect **T**echnology
TPR	**T**hree-dimensional **P**lace and **R**oute
TSV	**T**hrough-**S**ilicon **V**ia
VLSI	**V**ery-**L**arge-**S**cale **I**ntegration
VPR	**V**ersatile **P**acking, Placement and **R**outing

Dedicated to my parents, Vassili and Eva Danassi
for their love and support.

Chapter 1

Introduction

For more than thirty years now, since their introduction in 1984, field-programmable gate arrays (FPGAs) have become one of the most popular implementation media for digital circuits, rapidly changing the way digital logic is designed and deployed. By combining the high performance of application-specific integrated circuits (ASICs) and the flexibility of microprocessors, FPGAs have made possible entirely new types of applications. This has made FPGAs a compelling proposition for almost any type of design, helping FPGAs supplant other media such as ASICs, ASSPs and digital signal processors (DSPs) in some traditional roles. As semiconductor manufacturers have been shrinking transistor size in Integrated Circuits (ICs) to achieve the yearly increase in performance described by Moore's Law, the logic capacity of FPGAs has greatly increased, making FPGAs a viable implementation alternative for ever larger designs. To make the best out of today's resources FPGAs have to offer, one must develop efficient, high quality Computer-Aided Design (CAD) tools, since the quality of the CAD tools used to map a circuit into an FPGA, greatly determines the performance of the end result. In this chapter we introduce the world of programmable hardware and present the various technologies that are used today.

The rest of the chapter is organized as follows: Section 1.1 provides an overview of FPGAs, starting by defining what is a logic device and the differences between fixed and programmable logic and continues by presenting the FPGA's architecture and comparing it to other alternatives. Section 1.2 presents an overview of the FPGA CAD flow, and finally Section 1.3 summarizes the chapter.

1.1 Overview of FPGAs

1.1.1 Logic Devices

In the world of digital electronic systems, there are three basic kinds of devices[1]: memory, microprocessors, and logic. Memory devices store random information such as the contents of a spreadsheet or database. Microprocessors execute software instructions to perform a wide variety of tasks such as running a word processing program or video game. Logic devices provide specific functions, including device-to-device interfacing, data communication, signal processing, data display, timing and control operations, and almost every other function a system must perform.

Logic devices can be classified into two broad categories - fixed and programmable. As the name suggests, the circuits in a fixed logic device are permanent, they perform one function or set of functions - once manufactured, they cannot be changed. On the other hand, programmable logic devices (PLDs) are standard, off-the-shelf parts that offer customers a wide range of logic capacity, features, speed, and voltage characteristics - and these devices can be changed at any time to perform any number of functions.

1.1.2 Fixed Logic vs. Programmable Logic

With fixed logic devices, the time required to go from design, to prototypes, to a final manufacturing run can take from several months to more than a year, depending on the complexity of the device. And, if the device does not work properly, or if the requirements change, a new design must be developed. The up-front work of designing and verifying fixed logic devices involves substantial non-recurring engineering costs (NRE [1]). These NRE costs can run from a few hundred thousand to several million dollars[1].

With programmable logic devices, designers use inexpensive software tools to quickly develop, simulate, and test their designs. Then, a design can be quickly programmed into a device, and immediately tested in a live circuit. The PLD that is used for this prototyping is exactly the same PLD that will be used in the final production of a piece of end-equipment. There are no NRE costs and the final design is completed much faster than that of a custom, fixed logic device[1].

[1] NRE refers to the one-time cost to research, develop, design and test a new product. When budgeting for a project, NRE must be considered to analyze if a new product will be profitable. Even though a company will pay for NRE on a project only once, NRE costs can be prohibitively high and the product will need to sell well enough to produce a return on the initial investment. NRE is unlike production costs, which must be paid constantly to maintain production of a product. It is a form of fixed cost in economics terms[2].

Another key benefit of using PLDs is that during the design phase customers can change the circuitry as often as they want until the design operates to their satisfaction. That's because PLDs are based on re-writable memory technology - to change the design, the device is simply reprogrammed. Once the design is final, customers can go into immediate production by simply programming as many PLDs as they need with the final software design file[1].

1.1.3 History of Programmable Logic

The first PLDs where introduced in the late 60's, early 70's. Before that, systems were built from lots of individual discrete logic chips such as ANDs, ORs, flip-flops, etc. with a spaghetti-like maze between them (such as in Figure 1.1). As you can imagine, manufacturing such a system took a lot of time and effort and it was very difficult to modify and maintain it after it was built. A more versatile way to create arbitrary combinational logic functions was with the use of read-only memory (ROM) chips[3]. Consider a ROM with N inputs (the address lines) and M outputs (the data lines or word width). Any conceivable function of all possible combinations of the N inputs can be made to appear at any of the M outputs, making this the most general-purpose combinational logic device, limited only by the number of address lines and the word width. However, using ROMs had also several disadvantages:

- They are usually much slower than dedicated logic circuits,
- They cannot necessarily provide safe "covers" for asynchronous logic transitions so the PROM's outputs may glitch as the inputs switch,
- They consume more power,
- They are often expensive, especially if high speed is required.

To overcome all of the above problems, the chip makers introduced a revolutionary idea[4]: They placed an unconnected array of AND-OR gates in a single chip which contained an array of fuses that could be blown open or left closed to connect various inputs to each AND gate (Figure 1.2). And thus, the programmable logic device (PLD) was born! Programming such a PLD was considerably easier, using a set of boolean sum-of-product equations to perform the logic functions needed in your system. Modifying the function of a design was also pretty painless, since you could simply remove the PLDs, blow a new fuse pattern into them, and then place them back into the circuit board.

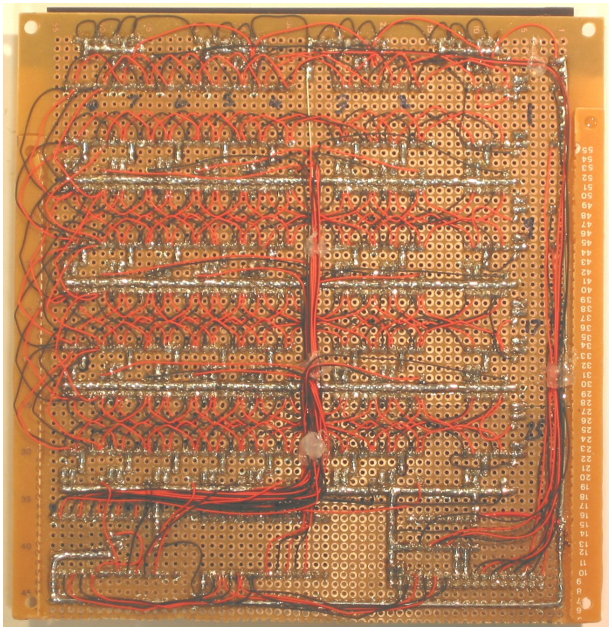

FIGURE 1.1: Spaghetti wiring (www.franksworkshop.com.au).

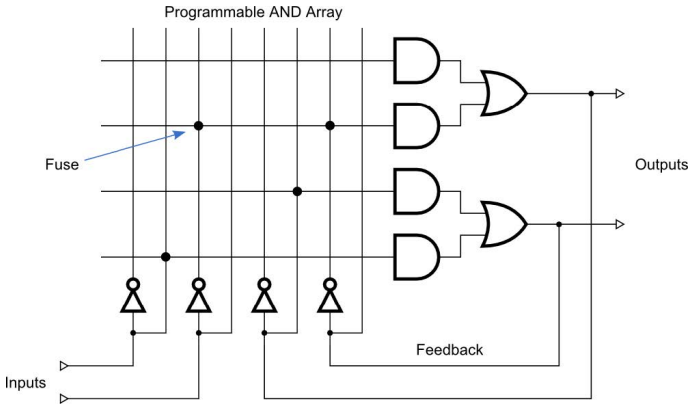

FIGURE 1.2: Programmable AND Array[4].

Unfortunately simple PLDs could only handle up to 10 – 20 logic equations, so you couldn't fit a very large logic design into just one of them. You had to figure out how to break your larger designs apart and fit them into a set of PLDs. This was time consuming and meant that you had to interconnect the PLDs with wires, which was a big drawback because eventually you would have to make some design change that couldn't be handled just by reprogramming the PLDs and then you would have to build a new circuit board. The chip makers came to the rescue again by building much larger programmable chips called complex programmable logic devices (CPLDs) and field-programmable gate arrays (FPGAs). With these, you could essentially get a complete system onto a single chip.

1.1.4 FPGA Architecture

Field-Programmable Gate Arrays (FPGAs) are pre-fabricated silicon devices that can be programmed to become almost any kind of digital circuit or system. Although One-Time Programmable (OTP) FPGAs are available, the dominant types are SRAM-based which can be reprogrammed as the design evolves. FPGAs allow designers to change their designs very late in the design cycle. They enable you to program product features and functions, adapt to new standards, and reconfigure the hardware even after the product has been manufactured and deployed in the field, hence the term "field programmable". In addition, FPGAs allow for field upgrades to be completed remotely, eliminating the costs associated with re-designing or manually updating electronic systems.

The basic architecture of FPGAs is very simple[5–9]. In general terms, there are only two types of resources: logic and interconnect. Logic resources are used to implement the digital logic of the circuit and interconnect is used to connect the logic block's input and outputs to form a larger circuit.

Logic Elements

Every k-ary Boolean function can be expressed as a propositional formula in k variables[10] which in turn can be expressed as a truth table[2]. Building on that, we can create complex structures that can do arithmetic, such as adders and multipliers, as well as decision-making structures that can evaluate conditional statements, such as the classic if-then-else. As a result, we can describe elaborate algorithms simply by using truth tables, making the truth table the computational heart of the FPGA. The most common way to implement a truth table is with a look-up table (LUT). The LUT operates as a memory with N address lines and 2^N memory locations. From a circuit implementation perspective, a LUT can be formed simply from an N:1 (N-toone) multiplexer and an N-bit memory. In simple terms a LUT enumerates a truth table. Therefore, using LUTs gives an FPGA the generality to implement arbitrary digital logic.

The LUT can compute any function of N inputs by simply programming the lookup table with the truth table of the function we want to implement. As shown in Figure

[2]Efficient implementation of Boolean functions is a fundamental problem in the design of combinational logic circuits. Modern electronic design automation tools often rely on a representation for truth tables or logic expressions called binary decision diagrams (BDDs). Compared with other methods for representing logic expressions, BDDs have unique features, such as unique concise representation, high processing speed, and low memory space consumption. They are a powerful means for computer processing of logic functions. As logic design has been done in recent years with computers, BDDs are extensively used because of these features. BDDs are used in computer programs for automation of logic design, verification (i.e., identifying whether two logic networks represent the identical logic functions), diagnosis of logic networks, simplification of transistor circuits, and many other areas (see [11]).

1.3, if we wanted to implement a 3-input exclusive-or (XOR) function with our 3-input LUT (3-LUT), we would assign values to the lookup table memory such as the pattern of select bits chooses the correct row's answer. Thus every row would yield a result of 0 except in the four cases where the XOR of the three select lines yields 1. To complete our logic block we need to give the FPGA the ability to maintain some sense of state. To do so we can add a simple storage element such as flip-flops. Now our logic element looks something like the one in Figure 1.4, which in reality bears a very close resemblance to those used in commercial FPGAs.

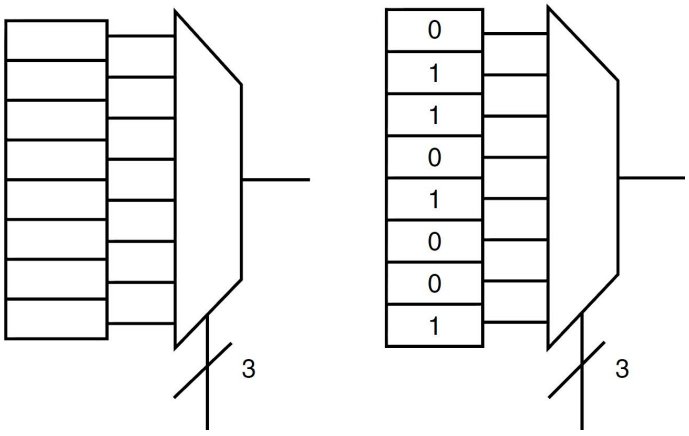

FIGURE 1.3: A 3-LUT schematic (a) and the corresponding 3-LUT symbol and truth table (b) for a logical XOR.[7].

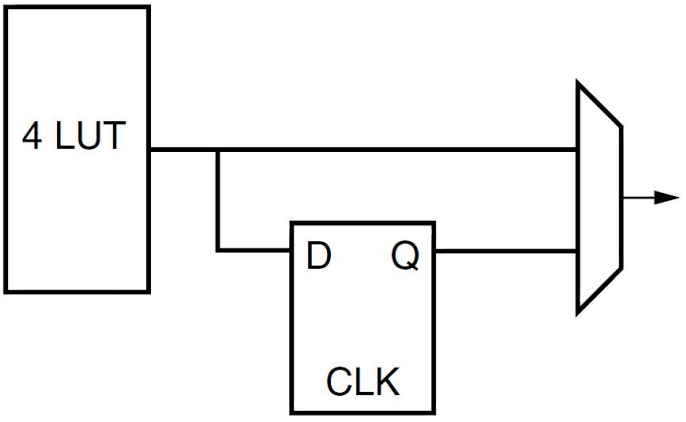

FIGURE 1.4: A simple look-up table logic block.[7].

Besides this simple logic block, the FPGA architects have augmented the logic capabilities of modern FPGAs to increase performance. Commercial FPGAs nowadays incorporate many extended logic elements such as Fast Carry Chains, Multipliers, RAM and dedicated CPUs.

Interconnect

Now that we have an understanding of how computation is performed in an FPGA at the single logic block level, we bring our attention to how these computation blocks can be tiled and connected together to form the fabric that is our FPGA. Generally FPGAs consist of an array of programmable logic blocks that are interconnected to each other as well as to the programmable I/O blocks through some sort of programmable routing architecture, as shown in Figure 1.5. Large computations are broken into LUT-sized pieces and mapped into physical logic blocks in the array. The interconnect is then configured to route signals between logic blocks appropriately. With enough logic blocks, we can make our FPGAs perform any kind of computation we desire. As with logic element structures, FPGA designers have used a variety of routing structures within their FPGAs. Generally some amount of routing is included within each logic cluster so that the logic elements can be combined to form larger functions. External to the logic clusters is the more global routing architecture of the FPGA, which is what we will focus on in this section.

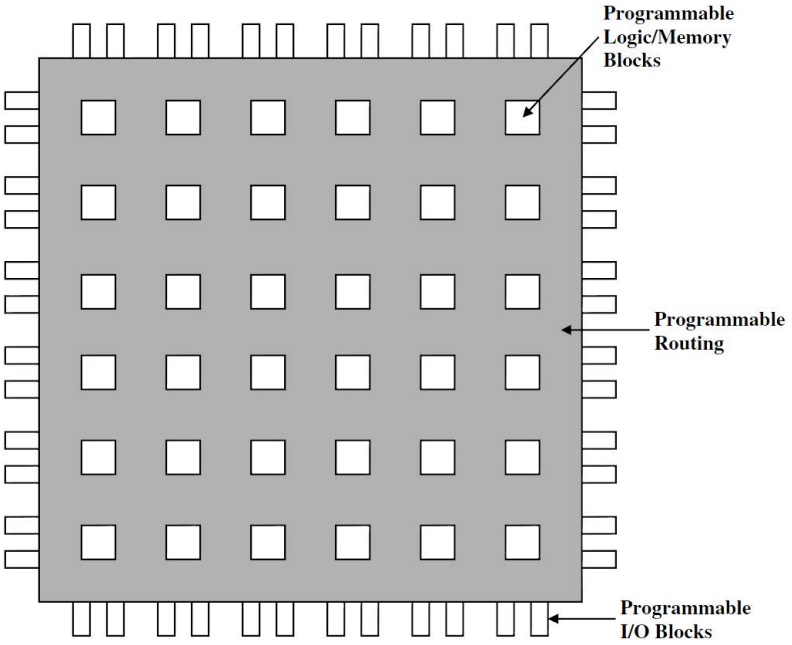

FIGURE 1.5: A Generic FPGA Architecture.[8].

Current popular FPGAs implement what is often called island-style architecture. This design has logic blocks tiled in a two-dimensional array and interconnected in some fashion. The logic blocks form the islands which float in a sea of interconnect. There are many interconnect structures varying from simple ones, such as nearest-neighbor connectivity which allows each logic block to communicate directly only with each of its immediate neighbors (Figure 1.6), to more complicated ones such as the hierarchical

routing which groups together computational units to form clusters of logic blocks and then uses long wires to connect clusters of the same hierarchy (Figure 1.7). Another common interconnect structure is the segmented routing. In this case the routing structure is more generic and mesh-like. We use connection blocks (Figure 1.8) and switch boxes (Figure 1.9) to interconnect the logic elements. Specifically, the logic blocks access nearby communication resources through the connection block, which connects logic block input and output terminals to routing resources through programmable switches, or multiplexers. The connection block allows logic block inputs and outputs to be assigned to arbitrary horizontal and vertical tracks, increasing routing flexibility. The switch block appears where horizontal and vertical routing tracks converge. In the most general sense, it is simply a matrix of programmable switches that allow a signal on a track to connect to another track. A visual representation of an island-style architecture using segmented routing is shown in Figure 1.10.

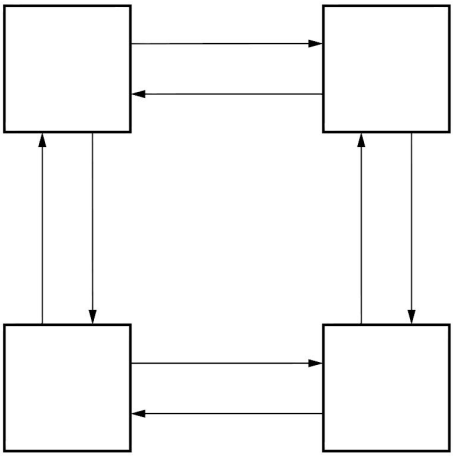

FIGURE 1.6: Nearest-neighbor Connectivity[7].

As with the logic blocks in a typical commercial FPGA, each switch point in the interconnect structure is programmable. For all of these programmable points, as in the logic blocks, modern FPGAs use SRAM bits to hold the user-defined configuration values.

In modern FPGAs, the silicon area consumed by interconnect greatly dominates the area dedicated to logic. Anecdotally, 90 percent of the available silicon is interconnect whereas only 10 percent is logic. With this imbalance, it is clear that interconnect architecture is increasingly important, especially from a delay perspective[7].

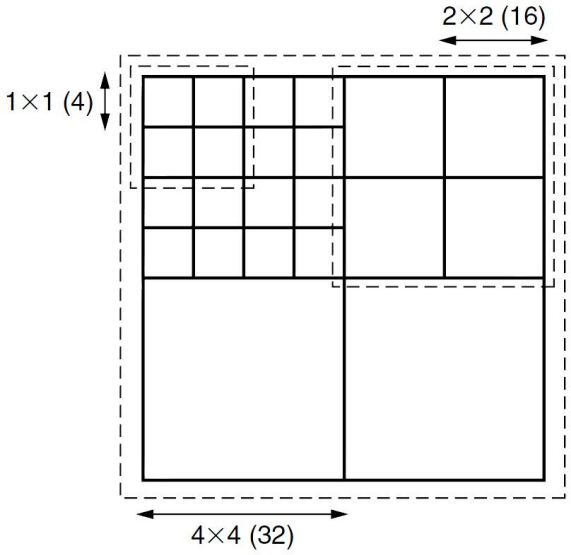

FIGURE 1.7: Hierarchical Routing[7].

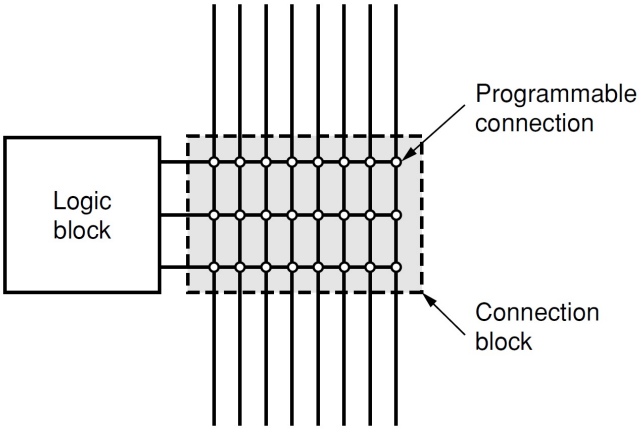

FIGURE 1.8: An example of a 2-D connection block[7].

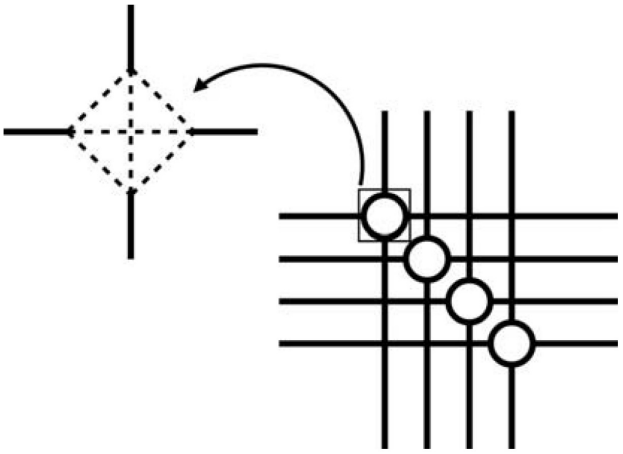

FIGURE 1.9: An example of a 2-D switch block[7].

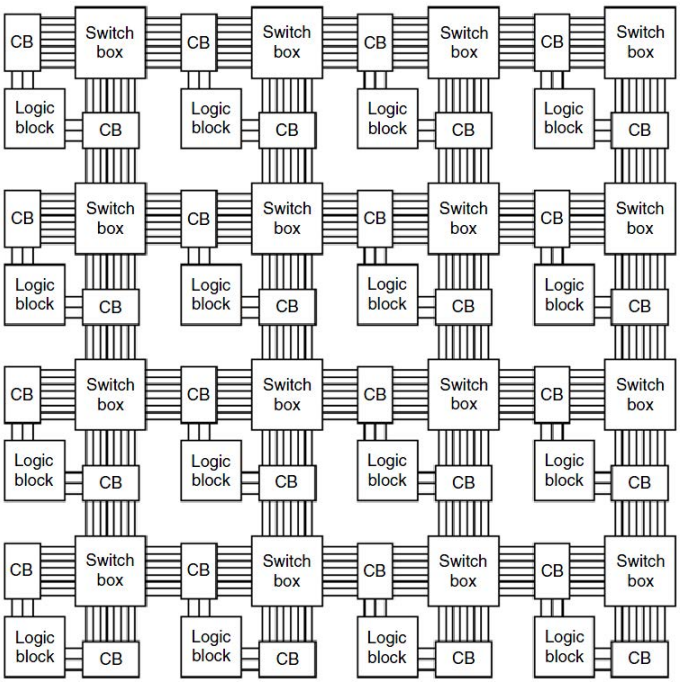

FIGURE 1.10: Segmented Routing[7].

1.1.5 FPGAs Compared To Other Platforms

FPGAs vs. CPLDs

The two major types of programmable logic devices are field programmable gate arrays (FPGAs) and complex programmable logic devices (CPLDs)[1]. The primary differences between CPLDs and FPGAs are architectural.

A CPLD is a combination of a fully programmable AND/OR array and a bank of macrocells[12]. The AND/OR array is reprogrammable and can perform a multitude of logic functions. Macrocells are functional blocks that perform combinatorial or sequential logic, and also have the added flexibility for true or complement, along with varied feedback paths. So a CPLD has two levels of programmability: each macrocell block can be programmed, and then the interconnections between the macrocells can be programmed as well. An example of the CPLD's block structure is shown in Figure 1.11.

Of the two, FPGAs offer the highest amount of logic density, the most features, and the highest performance. The largest FPGAs now shipping, provide millions of system gates[3]. These advanced devices also offer features such as built-in hardwired processors,

[3] Aldec announced HES-7, the largest off-the-shelf Xilinx Virtex-7 FPGA prototyping system at up to 288 million ASIC gates capacity, Feb 9, 2015, https://www.aldec.com/en/company/news/2015-02-09/250

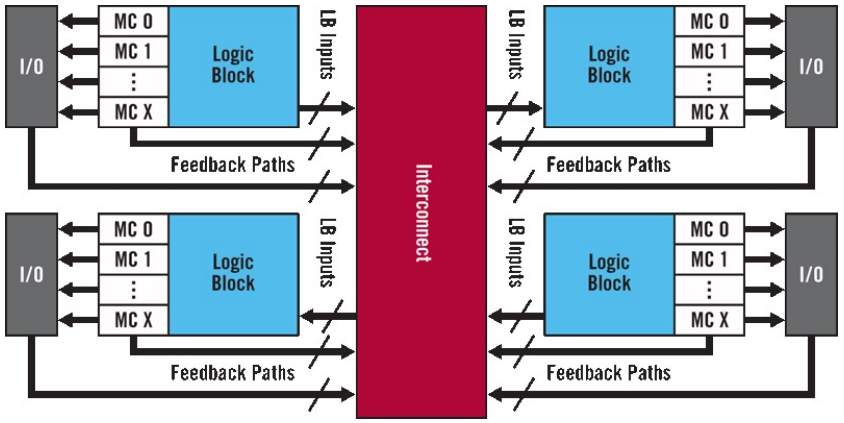

FIGURE 1.11: CPLD Block Structure[12].

substantial amounts of memory, clock management systems, and support for many of the latest, very fast device-to-device signaling technologies.

CPLDs, by contrast, offer much smaller amounts of logic - up to about 10,000 gates. But CPLDs offer very predictable timing characteristics and are therefore ideal for critical control applications. CPLDs also require extremely low amounts of power and are very inexpensive, making them ideal for cost-sensitive, battery-operated, portable applications such as mobile phones and digital hand-held assistants.

FPGAs vs. ASICs

An application-specific integrated circuit (ASIC) is an integrated circuit (IC) customized for a particular use, rather than intended for general-purpose use. An ASIC can be pre-manufactured for a special application or it can be custom manufactured (typically using components from a building block library of components) for a particular customer application. Compared to a programmable logic device, an ASIC can achieve higher speed and smaller power consumption, since it's designed to perform only a specific function, but it is usually much more expensive and time consuming to design and manufacture. In short there is a trade-off between design time, cost and risk reduction versus speed and power consumption, as shown in the following tables1.1 and 1.2[13]:

Even though FPGAs used to be selected for lower speed/complexity/volume designs in the past, nowadays the scales tip in their favor, even for high speed designs, since today's FPGAs can easily break the 500MHz performance barrier. Furthermore the FPGA design flow eliminates the complex and time-consuming floorplanning, place and route, timing analysis, and mask/re-spin stages of the project since the design logic is already synthesized to be placed onto an already verified, characterized FPGA device

FPGA Design

Advantage	**Benefit**
Faster time-to-market	No layout, masks or other manufacturing steps are needed
No upfront non-recurring expenses (NRE)	Costs typically associated with an ASIC design
Simpler design cycle	Due to software that handles much of the routing, placement, and timing
More predictable project cycle	Due to elimination of potential re-spins, wafer capacities, etc.
Field re-programmability	A new bitstream can be uploaded remotely

TABLE 1.1: FPGA Design Advantages

ASIC Design

Advantage	**Benefit**
Full custom capability	For design since device is manufactured to design specs
Lower unit costs	For very high volume designs
Smaller form factor	Since device is manufactured to design specs

TABLE 1.2: ASIC Design Advantages

(see Figure 1.12 for a comparison between the FPGA's and ASIC's design flow). Combining the simpler design cycle with an unprecedented increase in logic density and a host of other features, such as embedded processors, clocking and DSP blocks, make the FPGAs a compelling proposition for almost any type of design[13].

FPGAs vs. CPUs

A central processing unit (CPU) is the electronic circuitry within a computer that carries out the instructions of a computer program by performing the basic arithmetic, logical, control and input/output (I/O) operations specified by the instructions[14, 15]. The main and the most significant difference between the CPUs and the FPGAs is that FPGAs don't have a fixed hardware structure, on the contrary they are programmable according to user applications. On the other hand CPUs are designed to implement the Von Neumann architecture[4]. This means that all the transistors memory, peripheral

[4]The Von Neumann architecture, also known as the Von Neumann model and Princeton architecture, is a computer architecture based on John von Neumann's work in 1945 in the First Draft of a Report on the EDVAC[16]. This describes a design architecture for an electronic digital computer with parts consisting of a processing unit containing an arithmetic logic unit and processor registers, a control unit containing an instruction register and program counter, a memory to store both data and instructions, external mass storage, and input and output mechanisms. The meaning has evolved to be any stored-program computer in which an instruction fetch and a data operation cannot occur at the same time

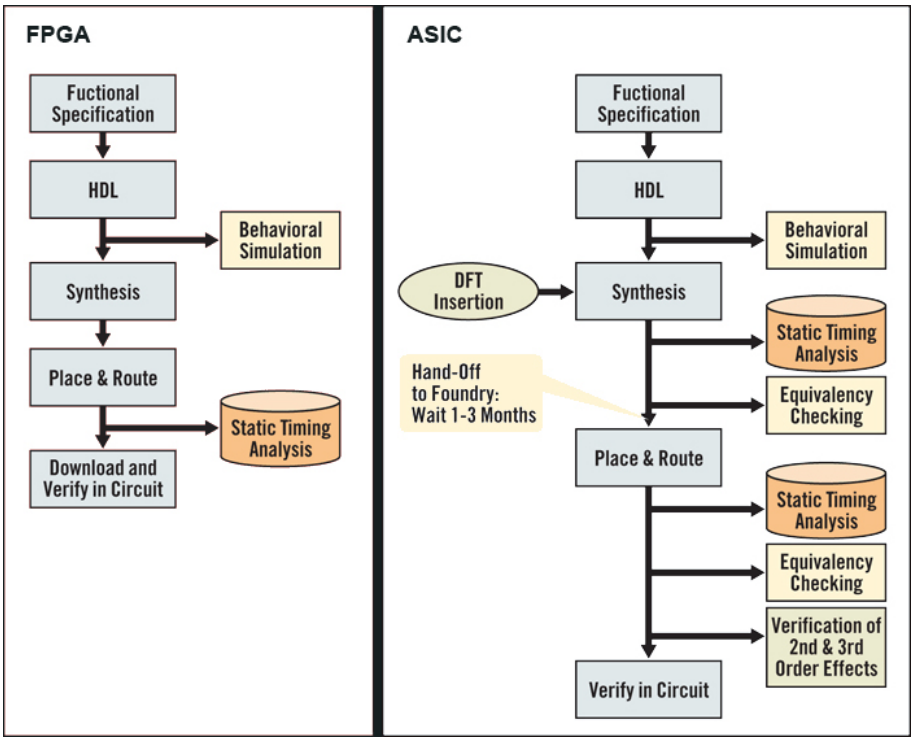

FIGURE 1.12: FPGA vs. ASIC Design Flow Comparison[13].

structures and the connections are constant and the operations which a processor can do (addition, multiplication, I/O control, etc.) are predefined.

Additionally CPUs are sequential processing devices. Although modern CPU designs are at least somewhat superscalar with emphasis in achieving high-ILP (Instruction-level parallelism), making them to behave less linearly and more in parallel, they mainly work by breaking an algorithm up into a sequence of operations and execute them one at a time. FPGAs on the other hand are parallel preprocessing devices. Bearing that in mind, FPGAs can outperform CPUs in certain tasks because they can achieve the same result in fewer clock cycles and they can process larger data at once whereas in CPUs the data flow is limited by the processor's bus (typically 32 bit, 64-bit, etc.). Of course the results are highly dependent on the algorithm. On the other hand, the CPUs are much more power efficient and cost less.

FPGAs vs. GPUs

A Graphics Processing Unit (GPU) is a single-chip processor primarily used to manage computer graphics. Besides that, their highly parallel structure makes them very

because they share a common bus. This is referred to as the Von Neumann bottleneck and often limits the performance of the system[17].

effective in algorithms where processing of large blocks of data is done in parallel, making them a popular alternative platform for compute-intensive applications. The main advantages of the GPU as an accelerator stem from its high memory bandwidth and a large number of programmable cores with thousands of hardware threads. Unlike FPGAs, GPUs excel in floating-point operations, making them a natural fit for floating-point intensive applications such as signal or image processing.

GPUs are flexible and easy to program using high level languages and APIs which abstract away hardware details. In addition, compared with hardware modification in FPGAs, changing functions is straightforward via rewriting and recompiling code, but this flexibility comes at a cost. Just like CPUs, GPUs have a fixed hardware architecture which, compared to the flexible hardware fabric of FPGAs, leaves less room to the developers. On the contrary, with FPGAs the developers can directly steer module-to-module hardware infrastructure and trade-off resources and performance by selecting the appropriate level of parallelism to implement an algorithm. The hardware fabric is used to approximate a custom chip, effectively eliminating the inefficiencies caused by the traditional von Neumann execution model and can achieve vastly improved performance and power efficiency[18]. Finally, GPUs execute programs in a single instruction, multiple data (SIMD) fashion, which can only run one routine at a time. Threads are executed in warps and within a warp, the hardware is not capable of executing if-else statements at the same time. There is what we call warp divergence in which no more than half the threads per warp are being executed per iteration. Unlike that, FPGAs can run several kernels at a time and we don't need to try to avoid branching. Above all that, GPUs have historically been very power demanding which is a huge liability for embedded computing.

To sum up, FPGAs provide the best expectation of performance and flexibility, while GPUs tend to be easier to program and require less hardware resources.

1.2 CAD Tools

1.2.1 Mapping Designs to FPGAs

Implementing a circuit in a modern FPGA requires that millions of programmable switches and configuration bits are set to the proper state, either on or off. Clearly it is impossible for a circuit designer to specify the state of each programmable bit by hand. Instead designers describe a circuit at a higher level of abstraction, typically using

a hardware description language (HDL)[5], such as VHDL and Verilog. Computer-Aided Design (CAD) programs then convert this high level description into a programming file specifying the state of every programmable switch in the FPGA. To keep the complexity of this procedure tractable, the problem of determining how to map a circuit into an FPGA is broken down into a series of sequential subproblems[5]. Those subproblems include synthesis, placement, routing and finally, the bitstream generation, as shown in Figure 1.13.

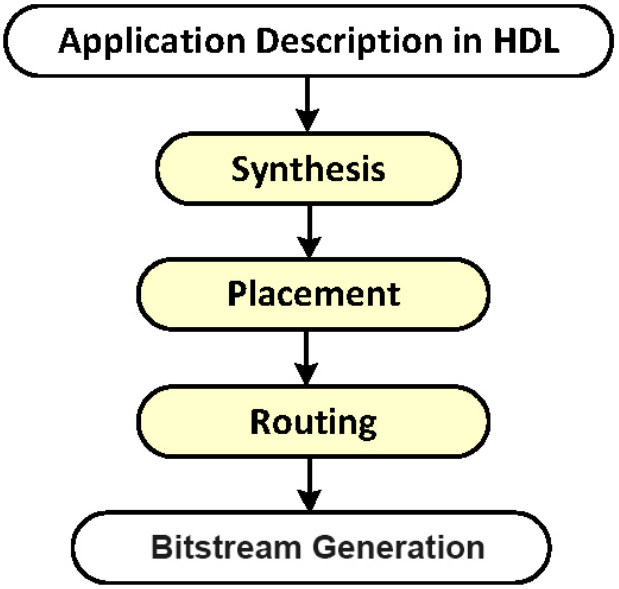

FIGURE 1.13: FPGA CAD Flow.

1.2.2 Synthesis

Logic synthesis flow typically consists of four steps (Figure 1.14). First, the initial network is optimized using technology-independent optimization techniques such as node extraction/substitution and don't-care optimization. Second, the optimized network is decomposed into one consisting of 2-input gates plus inverters (that is, the network becomes 2-bounded) to increase flexibility in mapping. Third, the actual mapping takes place, with the goal of covering the 2-bounded network with K-LUTs while optimizing one or more objectives. Finally in the packing stage, several LUTs and registers are packed into one logic block, respecting limitations such as the number of LUTs or the number of distinct input signals and clocks a logic block may contain[5, 7].

[5]HDLs were created to implement register-transfer level (RTL) abstraction. RTL is a design abstraction which models a synchronous digital circuit in terms of the flow of digital signals (data) between hardware registers, and the logical operations performed on those signals. RTL abstraction is used in HDLs to create high-level representations of a circuit, from which lower-level representations and ultimately actual wiring can be derived. Designing at the RTL level is a typical practice in modern digital circuit design[19–21].

Technology mapping is maybe the most essential step in the above process and has a significant impact on the quality of the final FPGA implementation. Many algorithms have been proposed for optimizing area[22–25], timing[24, 26, 27], power[28–30], and routability[31, 32]. They can be classified as structural or functional. A structural mapping algorithm does not modify the input network other than to duplicate logic. It reduces technology mapping to a covering problem in which the technology-independent logic gates in the input network are covered with logic cones so that each cone can be implemented using one logic cell — for example, a K-input lookup table (K-LUT) — for LUT-based FPGAs. Figure 1.15 is an example of structural mapping. A functional mapping algorithm, on the other hand, treats technology mapping in its general form as a problem of Boolean transformation/decomposition of the input network into a set of interconnected logic cells. It mixes Boolean optimization with covering. Functional mapping algorithms tend to be time consuming, which limits their use to small designs or to small portions of a design[7].

Recent advances in technology mapping try to combine mapping with other steps in the design flow. Such integrated mapping algorithms have the potential to explore a larger solution space compared to what is possible with just technology mapping and thus have the potential to arrive at mapping solutions with better quality. For example, algorithms have been proposed to combine logic synthesis with covering to overcome the limitations of pure structural mapping[7, 33, 34].

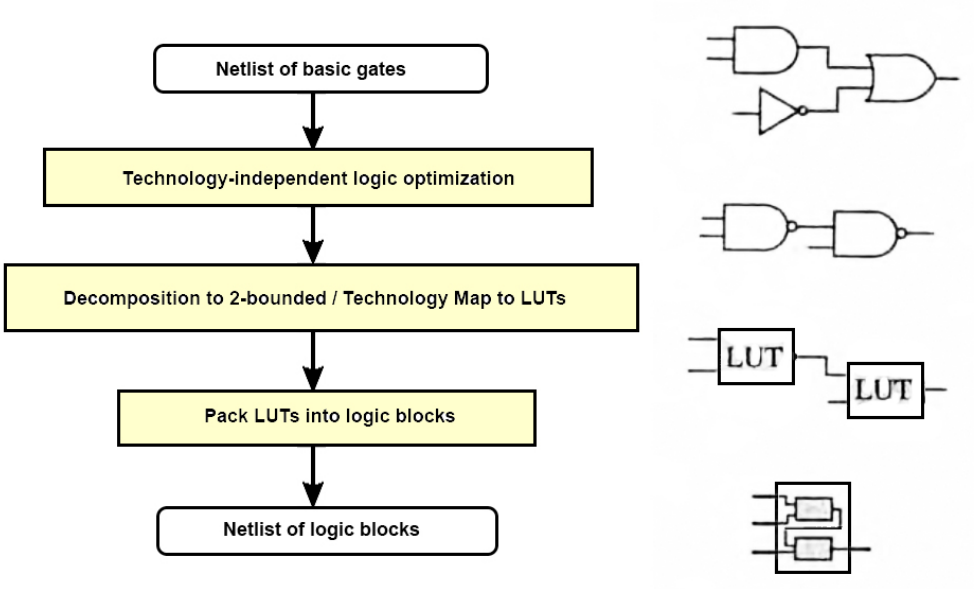

FIGURE 1.14: Logic Synthesis Flow.

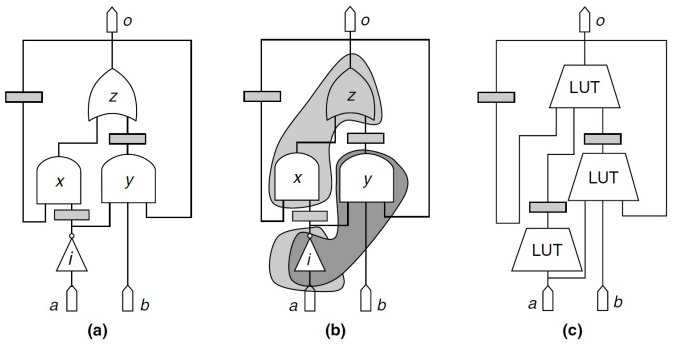

FIGURE 1.15: Structural Technology Mapping: (a) original network, (b) covering, (c) mapping solution[7].

1.2.3 Placement

Placement follows the synthesis process in FPGA CAD flow. It is when we choose a location for each block in the circuit. An FPGA placement algorithm takes two basic inputs: (1) a netlist specifying the functional blocks to be implemented and the connections between them, and (2) a device map indicating which functional unit can be placed at each location. The algorithm selects a legal location for each block according to the optimization goals and/or legality constraints. Usually the metrics we try to optimize are: critical path delay, power consumption and routability of the final result. Both the legality constraints and the optimization metrics depend on the FPGA architecture being targeted. Figure 1.16 illustrates the FPGA placement problem[7].

A good placement is extremely important for FPGA designs because, apart from the great impact it has to the overall design's speed and power consumption, without a high quality placement, a circuit generally cannot be successfully routed. At the same time, finding a good placement for a circuit is a challenging problem. A large commercial FPGA contains more than 500,000 functional blocks, leading to approximately 500,000! possible placements. Exhaustive evaluation of the placement solution space is therefore impossible. In addition, placement is an NP-complete problem, so there are no known polynomial algorithms that produce optimal results. As a result, the development of fast and effective heuristic placement algorithms is a very important research area.

Types of Placement Algorithms

There are three major classes of placers used today: partition-based (min-cut)[35, 36], analytic[37] and simulated annealing based placers[38–40].

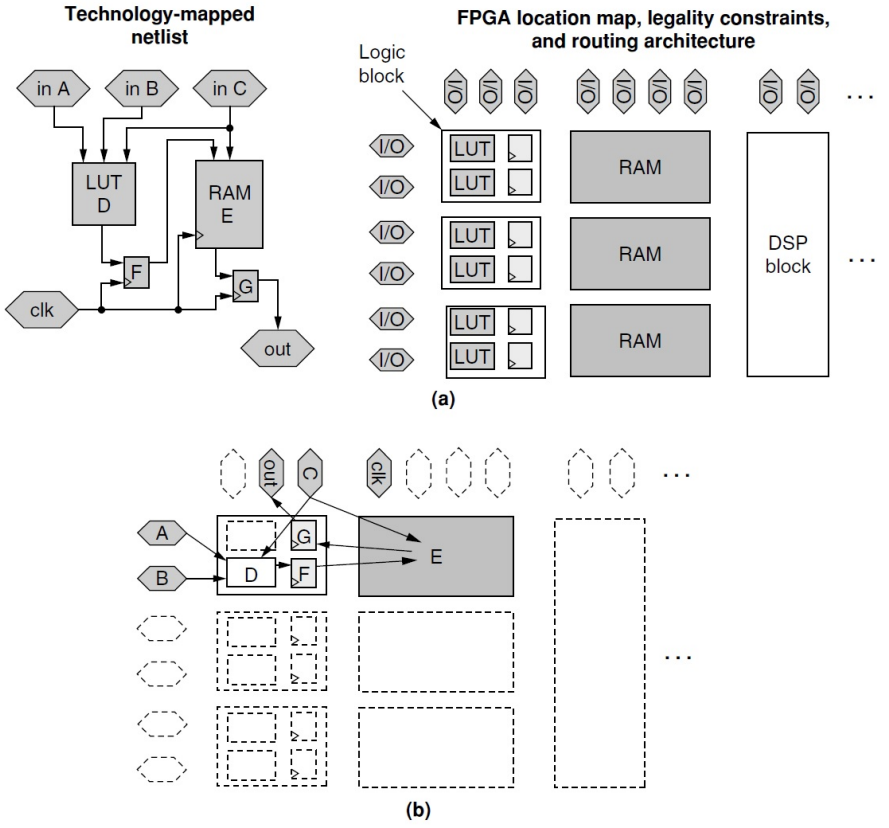

FIGURE 1.16: Placement Overview: (a) inputs to the placement algorithm, (b) placement algorithm output[7].

A partition-based placer works by recursively partitioning the circuit netlist and assigning each partition to a different physical region in the FPGA. Usually each partitioning step divides a previous (larger) partition into two pieces (bipartitions the component) although some algorithms perform multiway partitioning to produce a larger number of circuit partitions in each step. Partitioning algorithms attempt to minimize the number of nets that are cut, or that cross, between partitions. Since each partition of the circuit will be assigned to a different region of the FPGA, partition-based placement minimizes the number of nets leaving each region and hence indirectly optimizes the amount of wiring required by the design. Partition-based placement can leverage the availability of high-quality, CPU-efficient partitioning algorithms, making this approach scalable to large problems. However, for some FPGA architectures, partition-based placement suffers from the disadvantage that it does not directly optimize the circuit timing or the amount of routing required by the placement. Hierarchical FPGAs are good candidates for partition-based placement, since their routing architectures create natural partitioning cut lines. Recursive partitioning has also been used for placement in island-style FPGAs. In an island-style FPGA, blocks separated by a short Manhattan distance can be connected with a small amount of routing. Consequently, the cut lines

are designed to divide the FPGA into ever-shrinking squares. The fewer the signals that leave each square, the less interconnect is required[7].

Analytic algorithms are based on creating a smooth function of a placement that approximates routed wirelength. Efficient numerical techniques are used to find the global minimum of this function. If the function approximates wirelength well, this solution is a placement with good wirelength. However, this global minimum is usually an illegal placement, so constraints and heuristics must be applied to guide the algorithm to a legal solution. While analytic placement approaches are popular for ASICs, few exist for FPGAs, likely due to the more difficult FPGA placement legality constraints[7].

Simulated annealing is the most widely used placement algorithm for FPGAs. It mimics the annealing procedure by which strong metal alloys are created. Initially blocks can move fairly freely, but as the temperature drops they gradually freeze into a high-quality placement[41]. The basic flow of simulated annealing for placement is as follows: First an initial placement is generated. This initial placement is generally of low quality, and is often created simply by assigning each block to the first legal location found. The placement is then iteratively improved by proposing and evaluating placement perturbations, or moves. A placement perturbation is proposed by a move generator, generally by moving a small number of blocks to new locations. A cost function is used to evaluate the impact of each proposed move. Moves that reduce cost are always accepted, or committed to the placement, while those that increase cost are accepted with probability $e^{-\Delta_{Cost}/T}$, where T is the current temperature. This function ensures that moves that increase the cost by an amount that is small compared to the current temperature are likely to be accepted, while moves that increase the cost by an amount much larger than the current temperature are not. Accepting some moves that increase the cost helps escape local minima and produces a higher-quality final placement. At the start of the anneal temperature is high. Then it gradually decreases according to the annealing schedule. This schedule also controls how many moves are performed between temperature updates and when the placement is considered sufficiently optimized that the anneal should end[7]. VPR[38] is maybe the most popular simulated-annealing-based placement tool.

Optimization Goals

The basic goal of an FPGA placement algorithm is to minimize the interconnect required to route the signals between the logic blocks. The routing required to connect two blocks is a function not only of the distance between them but also of the FPGA architecture. In Figure 1.17 we see an example of how the FPGA's architecture influences the wirelength

for a given placement. In an island-style FPGA, the amount of wiring required to connect two functional blocks is roughly proportional to the Manhattan distance between them. But for hierarchical architectures the case is quite different. There, the amount of wiring required to connect two functional blocks is proportional to the number of levels of the routing hierarchy that must be traversed to connect them. Clearly FPGA placement algorithms must have a model of the routing architecture they target in order to achieve good results.

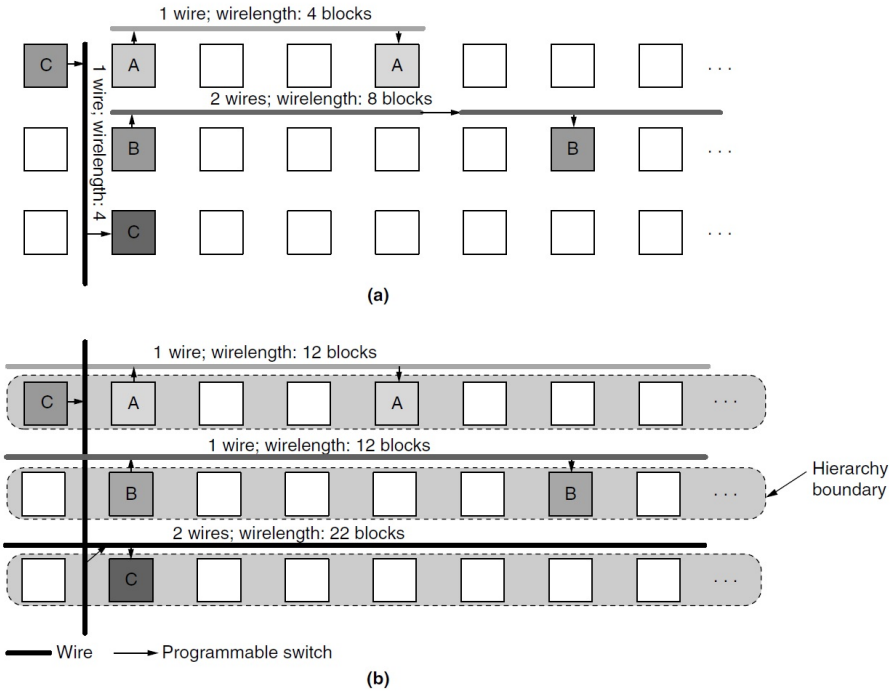

FIGURE 1.17: Influence of FPGA architecture on wirelength: (a) island-style FPGA, (b) hierarchical FPGA[7].

FPGA placement tools can broadly be divided into routability-driven and timing-driven algorithms. Routability-driven algorithms try to create a placement that minimizes the total interconnect required, as this increases the probability of successfully routing the design. Since FPGA interconnect is prefabricated, the amount of interconnect in each region of a device is fixed, and a placement that requires more interconnect in a device region than that region contains cannot be routed. Consequently, some routability-driven placement algorithms minimize not only the total wiring required by the design but also the amount of routing congestion. Routing congestion occurs when the interconnect demand approaches or exceeds the fabricated wiring capacity in some part of the FPGA. On the other hand, in addition to optimizing for routability, timing-driven algorithms use timing analysis to identify critical paths and/or connections and to optimize the delay of those connections. Since most delays in an FPGA are due to

the programmable interconnect, timing-driven placement can achieve a large improvement in circuit speed over routability-driven approaches. Finally, some recent FPGA placement algorithms attempt to minimize power consumption as well (power-driven placement)[7].

1.2.4 Routing

Once locations for all the logic blocks in the circuit have been chosen, we have to find a path to route the signals to all those resources. That consists of determining which programmable switches should be turned on to connect all the logic block input and output pins required by the circuit. Routing is a crucial step in the mapping of circuits to FPGAs. For large circuits that utilize many FPGA resources, it can be very difficult and time consuming to successfully route all of the signals. Just like placement, routing is also a NP-complete problem[42], so there are no known polynomial-time algorithms that produce good results. Additionally, the performance of the mapped circuit highly depends on routing critical and near-critical paths with minimum interconnect delays. One disadvantage of FPGAs is that they are slower than their ASIC counterparts, so it is important to squeeze out every possible nanosecond of delay in the routing[7].

In FPGA routing, one usually represents the routing architecture of the FPGA as a directed graph. Each wire and each logic block pin becomes a node in this routing-resource graph and potential connections become edges. Routing a connection corresponds to finding a path in this routing-resource graph between the nodes representing the logic block pins to be connected. To avoid using up too many of the limited number of wires in an FPGA, one requires this path to be as short as possible. Furthermore, it is important that the routing for one net does not use up routing resources another net needs, so most FPGA routers have some kind of congestion avoidance scheme to resolve contention for routing resources. An additional optimization goal is to make nets on or near the critical path fast by routing them using short paths and fast routing resources. Routers that attempt to optimize timing in this way are called timing-driven, whereas delay-oblivious routers are purely routability-driven. Since most of the delay in FPGAs is due to the programmable routing, timing-driven routing is crucial to obtain good circuit speeds[5].

However, the first and most important goal, is a complete routing of all signals, which is quite difficult to achieve in FPGAs because of the hard constraints on routing resources. Unlike ASICs, FPGAs have a fixed amount of interconnect. The usual approach in placement is to minimize the wiring resources anticipated for routing signals. Although this reduces the overall demand for resources, signals inevitably compete for

the same resources during routing. The challenge is to find a way to allocate resources so that all signals can be routed. The second goal, minimizing delay, requires the use of minimum-delay routes for signals, which can be expensive in terms of routing resources, especially for high-fanout signals. Thus, the solution to the entire routing problem requires the simultaneous solution of two interacting and often competing subproblems.

To make the FPGA routing problem tractable, nearly all of the routing schemes in the literature incorporate features of the underlying architecture. The problem is that new architectures become constrained by the restrictions of such existing routing algorithms.

1.2.5 Bitstream

The collection of binary data used to program the reconfigurable logic device is most commonly referred to as a "bitstream". The bitstream spatially represents the configuration data of a large collection of small, relatively simple hardware components. It is loaded into the device's internal units before the device is placed in its operating mode, and typically, no changes are made to the data while the device is operating. There are some significant exceptions to this rule: The configuration data may in fact be changed while a device is operational, but this is somewhat akin to self-modifying code in instruction set architectures. This is a very powerful technique, but carries with it significant challenges.

The software used to generate configuration bitstream data for FPGA devices is perhaps some of the most complex available. It usually consists of many layers of functionality and can run on the largest workstations for hours or even days to produce the output for a single design. FPGA configuration bitstream formats have almost always been proprietary and for that reason, the only tools available to perform bitstream generation tasks are those supplied by the device manufacturer. After the FPGA bitstream is created, it's typically stored externally in a nonvolatile memory such as an EPROM and then it's loaded into the device shortly after the initial power-up sequence, most often bit-serially[7].

1.3 Summary

Field-Programmable Gate Arrays (FPGAs) have become one of the key digital circuit implementation media. A crucial part of their success lies in their architecture, which governs the nature of their programmable logic functionality and their programmable

interconnect. FPGA architecture has a dramatic effect on the quality of the final device's speed performance, area efficiency, and power consumption[43]. Compared with other implementation platforms, FPGAs have several advantages for their users, including[8]:

- quick time-to-market,
- being a standard product,
- no non-recurring engineering costs for fabrication,
- pre-tested silicon for use by the designers
- re-programmability

CAD tools play an important role on the performance of an FPGA design. The greatest challenge that todays CAD tools face is the need to produce high quality placements and routings for ever-larger circuits. FPGA capacity doubles every two to three years, doubling the size of those problems at the same rate. In addition, uniprocessor speed is no longer increasing as quickly as it did in the past, which means that single processor speed will increase by less than two times in the same period. In order to maintain the fast time-to-market and ease of use historically provided by FPGAs, placement and routing algorithms cannot be allowed to take ever more CPU time. There is thus a compelling need for algorithms that are very scalable yet still produce high-quality results. The roadmap for future microprocessors indicates that the number of independent processors, or cores, on a single chip will increase rapidly in the coming years. Consequently, most engineers will have parallel computers on their desktops. Part of the solution to the problem of keeping FPGA placement times reasonable may be to find techniques and algorithms to exploit parallel processing without sacrificing result quality. Furthermore, new FPGA architectures (such as the three-dimensional chip stacking) can provide more flexibility and help break free from todays limitations.

Chapter 2

Three-dimensional Chip Stacking - A Whole New World!

Field-Programmable Gate Arrays (FPGAs) have become the implementation medium for the majority of digital circuits. The key to FPGAs' popularity is their feature to support application implementation by appropriately (re-)configuring the functionality of hardware resources. This allows FPGAs to provide higher flexibility, rapid product prototyping and significantly reduced non-recurring engineering (NRE) costs, as compared to ASIC (Application-Specific Integrated Circuit) devices. Additionally, this situation makes the FPGA paradigm to grow in importance, as there is a stronger demand for faster, smaller, cheaper, and lower-energy devices.

For decades, semiconductor manufacturers have been shrinking transistor size in Integrated Circuits (ICs) to achieve the yearly increase in performance described by Moore's Law, which exists only because the RC delay was negligible, as compared to the signal propagation delay. For sub-micron technology, however, the RC delay becomes a dominant factor. Furthermore, previous studies showed that at 130nm technology node, approximately 51% of the microprocessor's power is consumed by interconnect fabric[44]. This has generated many discussions concerning the end of device scaling as we know it, and has hastened the search for solutions beyond the perceived limits of current 2-D devices.

Three dimensional (3-D) chip stacking is considered by many as the silver bullet technology that will accommodate for all the aforementioned requirements[45]. Stacking multiple dies in the vertical axis and interconnecting them using very fine-pitch Through Silicon Vias (TSVs) enables the creation of chips with very diverse functionalities, implemented in different process technologies in a very small form factor[46]. Introducing locality along the z-axis enables on average shorter interconnections between system

components, which in turn leads to reduced signal propagation delay compared to conventional (i.e. 2-D) architectures[45, 46].

It is common for architecture designers to estimate that the longest interconnect is equal to twice the length of the die edge[47]. In order to show the potential gains of the new integration approach in this field, Figure 2.1 illustrates an example structure, where the interconnection length at 3-D platforms is significantly reduced compared to conventional 2-D architectures. More specifically, for a given total chip area A (both for 2-D and 3-D devices), as the number of device layers increases, the area per layer and consequently the corresponding longest interconnection path are reduced. For instance, the longest routing path for a 3-D architecture consisted of four layers is almost half compared to the 2-D device.

Additionally, by stacking smaller dies rather than manufacturing a large planar die also leads to yield significant cost improvements because the probability that a die is defective is positively correlated with its area. Consequently, the shift from horizontal to vertical stacking of circuits has the potential to rewrite the conventions of electronics design.

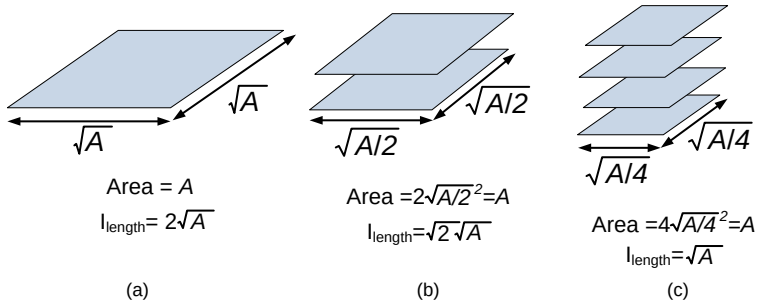

FIGURE 2.1: Variation on interconnection length for (a) a 2-D device, (b) a 3-D architecture with two layers, and (c) a 3-D architecture with four layers.

The benefits of using 3-D integration in logic chips are especially great for designing FPGAs, compared to other ICs, since these architectures suffer from data communication problems; interconnection delay and power/energy consumption are the main bottlenecks compared to alternative ASIC implementations. Hence, it is likely that the reconfigurable architectures will drive rapid adoption of 3-D integration, faster than any other device. However, in order such technology to be widely accepted, several challenges need to be satisfied. For instance, methodologies, algorithms and tools that facilitate the architecture-level exploration, as well as the application mapping onto 3-D platforms are essential.

This chapter summarizes a number of approaches related to the 3-D reconfigurable domain. Towards this direction, both architectural and algorithmic solutions are discussed. The rest of the chapter is organized as follows: Section 2.1 introduces the concept of 3-D reconfigurable architectures, while the algorithms and tools employed for the scopes of exploration phase and application mapping are discussed in Section 2.2. Section 2.3 summarizes the available academic toolsets for supporting these tasks and finally section 2.4 concludes the chapter.

2.1 3-D Reconfigurable Platforms

There are two integration approaches for constructing 3-D FPGAs. In the first approach, each physical layer is treated as a 2-D FPGA and the communication among layers is provided by 3-D Switch Boxes (SBs). An example of this fabrication approach has demonstrated the improved performance of a five-layer stack by decreasing the area×delay product of a 2-D FPGA by 36%, where an aggressive TSV pitch of 3μm is assumed[48].

Alternatively, each layer includes only one component of the FPGA architecture, such as memory, SBs, or logic gates (leading, in this case, to a three-layer 3-D FPGA). Experimental results show that this device achieves a 1.7× improvement in performance as compared to a 2-D FPGA[49]. There are two issues related to the evolution of this integration approach. Each component (e.g., logic, memory, SBs) does not necessarily scale with the same ratio as the FPGA size increases. This difference can result in dissimilar silicon area for each layer, which in turn, leads to wasting silicon (layers of the same area are preferred from a manufacturing perspective). In addition, each layer will require a different set of masks, increasing the manufacturing cost. Being able to utilize the same (full or partial) set of masks can lower the cost, which is a fundamental trait of the FPGA paradigm.

Figure 2.2 visualizes the differences between conventional SBs (found in 2-D FPGAs) and the 3-D SBs. A 2-D SB can be used where an incoming routing track is connected to wires on the same layer ($F_s = 3$). The SB flexibility F_s denotes the number of directions to which each incoming wire can be connected. Alternatively, 3-D SBs support connections to the third dimension (upper and lower layers, $F_s = 5$), except for the top and bottom layers of the 3-D stack where $F_s = 4$.

In addition to the envisioned integration approaches for 3-D FPGAs, these devices can also support different types of SBs due to the added design freedom that the third physical dimension introduces[50]. These different types of SBs can, in turn, be used to decrease the number of TSVs for the interlayer connections[48]. A comparison of

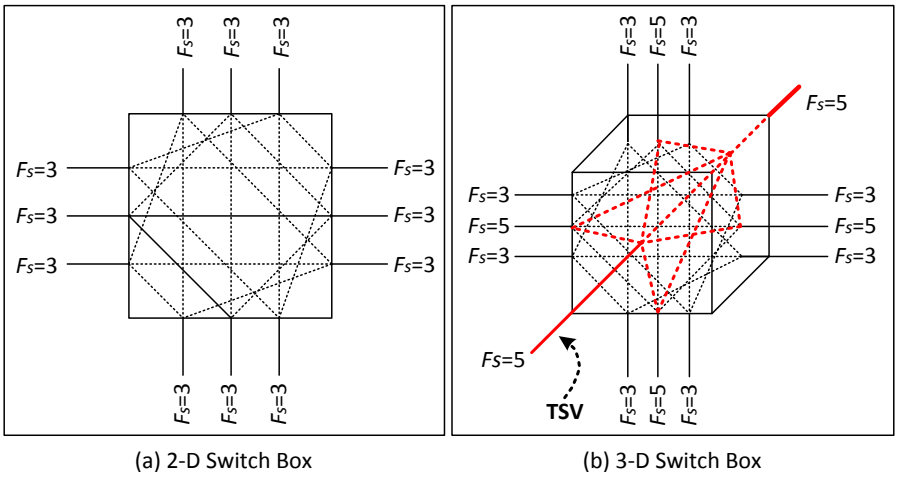

FIGURE 2.2: Template for different types of SBs: (a) 2-D SB and (b) 3-D SB.

these SB designs has shown that specific SB circuits require fewer TSVs to implement an application[51]. Furthermore, a prototype asynchronous 3-D FPGA has also demonstrated the feasibility of designing such devices[52], where the logic resources of this architecture are identical to the corresponding 2-D design, while the SBs are appropriately expanded with interlayer channels.

A different perspective in the design of 3-D FGPAs is discussed in[51]. Heterogeneity is introduced to the interconnect fabric rather than the distribution of the different circuit components composing an FPGA. The key idea of that work was to combine existing 2-D[50, 53] and 3-D[54, 55] SBs, so that the vertical interconnects are more efficiently utilized. By utilizing efficiently the vertical interconnections, such a 3-D FPGA, exhibits comparable or superior performance over other 3-D FPGA approaches[49, 52, 54–57]. From a manufacturing perspective, fewer interlayer connections within a 3-D FPGA, result to area savings. These savings, in turn, decrease the fabrication cost, while the additional silicon area within each layer can be used for logic blocks[58, 59].

The interest for designing 3-D FPGAs has been already addressed by the industry, since there are commercial approaches of the new design paradigm. Typical examples are the 3-D FPGAs provided by Tezzaron Corp.[60], as well as the devices from Xilinx (Virtex-7 & UltraScale FPGAs[61]) [1].

[1]These architectures differ in the way that layers are physically implemented. Specifically, the devices from Tezzaron employ a conventional 3-D process technology, wafer-level stacking, where the interlayer connectivity is provided through TSVs. This is a via that goes from the front side of the wafer (typically connecting to one of the lower metal layers) through the wafer and out the back[45]. On the other hand, the Virtex-7 devices from Xilinx, also known as 2.5D FPGAs, rely on the Stacked Silicon Interconnect Technology (SSIT)[61] to route signals between die slices of the same plane. The improvement in the number of logic elements of 2.5D FPGAs over conventional ones is very significant. For instance, the largest interposer-based FPGA, the Virtex-7 XC7V2000T, has 4 dies (which Xilinx calls Super Logic Regions) and 1.954 million logic elements, while the largest non-interposer Virtex-7 die (the XC7VX980T), has 979k logic elements and Altera's largest FPGA, the Stratix V 5SGXBB, has 952k

2.1.1 Design 3-D FPGAs with Heterogeneous Interconnect

The interconnection infrastructure highly affects the performance of applications implemented onto reconfigurable architectures. Due to the importance of this parameter, numerous architectural approaches have been proposed over the last years. This section discusses a methodology for designing 3-D FPGAs with heterogeneous interconnect network. This heterogeneity refers to the different types of SBs (either 2-D or 3-D) used in each layer, as compared to existing approaches[49, 52, 54–57] that employ only 3-D SBs. More specifically, the construction of the heterogeneous interconnect fabric depends on the statistical and spatial characteristics of the applications mapped on the 3-D FPGA, following the idea proposed initially by Betz and Rose[53], where 2-D FPGAs should be designed by considering the routing demands posed by the application placement and routing. The distinct difference of this approach compared to the rest of the implementations is that the objective is not the size of the routing channels but rather the distribution of the 2-D and 3-D SBs across the interconnect fabric.

The interlayer connectivity demand for a representative application implemented onto a 3-D FPGA with three layers is depicted in Figure 2.3, where the integration technology is TSV and only 3-D SBs are used for interconnections. Each point (i, j) of this graph depicts the number of utilized interlayer connections (i.e., TSVs) within the corresponding SB placed on spatial location (i, j). Based on this analysis, we conclude that the demand for interlayer communication varies between two arbitrary points (x_1, y_1, z_1) and (x_2, y_2, z_2) of the device, even for 3-D SBs placed on adjacent locations within the same layer. Additionally, the demand for utilizing interlayer connections between different application domains exhibits considerable but reasonably predictable variations. The higher percentage of utilized interlayer connections occurs in the middle of each layer, since placement and routing algorithms have higher flexibility to form connections in the middle of each layer as compared to its periphery.

This non-uniform utilization of the TSVs is due to the routing congestion that occurs in the center of an FPGA layer, and therefore, more routing resources have to be fabricated within this region[63]. Alternatively, board-level constraints can limit the placement of the I/Os at specific locations, resulting in increasing routing congestion close to the FPGA periphery. In this case, additional routing resources between the I/O pads and the logic block arrays are required.

The results shown in Figure 2.3 indicate that regions with approximately constant demand of vertical connectivity can be distinguished across each layer. We discretize, therefore, the hardware resources of each layer into three regions based on the number

logic elements. Even though all these FPGAs use a 28nm process, silicon interposer technology allows the creation of FPGAs with twice the resources possible on even an extremely large single die[62].

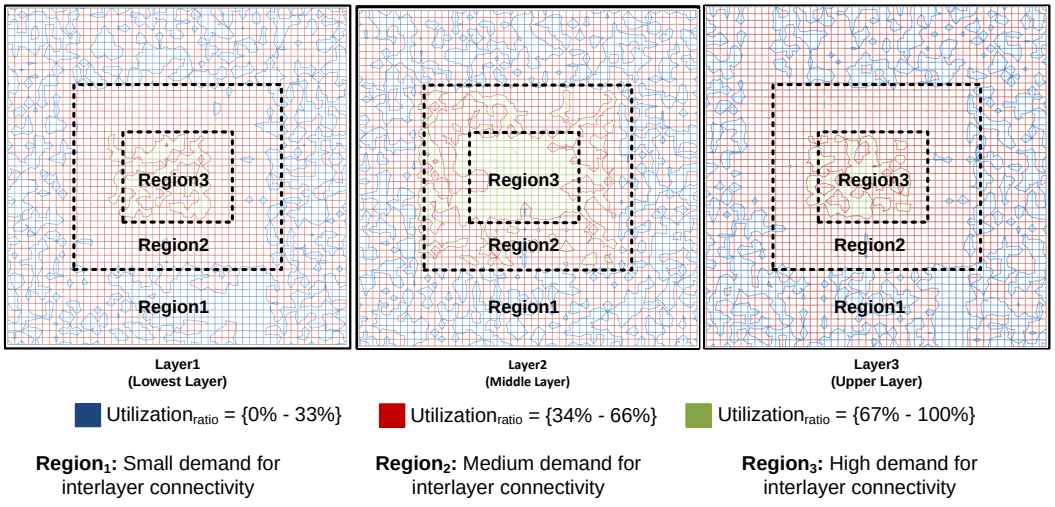

FIGURE 2.3: A classification example of the interlayer communication resources across a three-layer 3-D FPGA device[51].

of utilized vertical connections (i.e., TSVs) in the 3-D SBs of each region. More specifically, within $Region_1$ the percentage of the utilized TSVs in the 3-D SBs is less than 33%.Within $Region_2$ the density of the utilized TSVs in the 3-D SBs is between 33% and 66%. Finally, the percentage of the utilized TSVs in the 3-D SBs of $Region_3$ is greater than 66%.

Analogous approaches for selecting regions including different groups of hardware resources can also be applied. To best exploit this feature of 3-D FPGA architectures, it is feasible to employ a different density of interlayer connectivity at each (x, y, z) point of the 3-D architecture for each mapped application. This configuration, however, results in an application-specific (e.g., ASIC-like) design. Therefore, it is possible to introduce a piecewise homogeneous interconnection architecture, similar to the one depicted in Figure 2.4, consisting of regions with different interlayer interconnection densities. The authors in[51] depicted that 3-D FPGAs consisted of more than three regions do not provide additional gains in performance or power/energy consumption, whilst silicon area saturates rapidly with the number of these regions. As a result we conclude that the number of regions should be kept relatively small.

2.2 CAD Algorithms for 3-D Reconfigurable Architectures

Today's Computer-aided design (CAD) tools make it possible to automate many aspects of the design process. This has mainly been done by the use of effective and efficient algorithms and corresponding software structures. Still, the very large scale integration (VLSI) design process is extremely complex, and even after breaking the entire

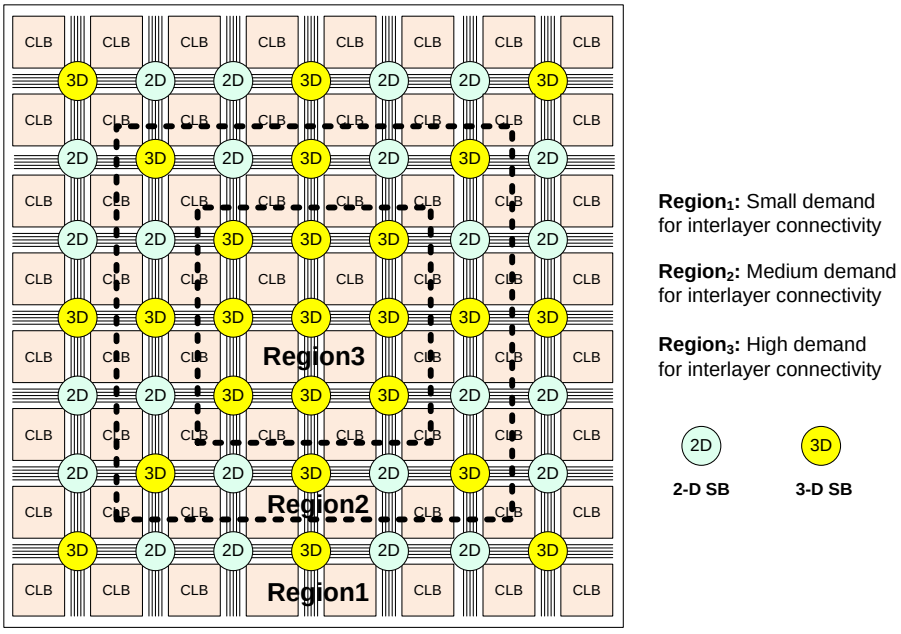

FIGURE 2.4: Example of architectural template for 3-D FPGAs with heterogeneous interconnect fabric.

process into several conceptually easier steps, it has been shown that each step is still computationally hard. Although, a number of EDA (Electronic Design Automation) tools that provide automated application implementation onto hardware platforms are available, their execution still imposes an increased run-time overhead. This problem becomes more evident by taking into account that the capacity, in term of logic resources, steadily increases at the rate anticipated by Moore's Law. Hence, the EDA tools must synthesize, place and route more logic blocks and interconnection networks for every new platform. However, given the increasing complexity of applications mapped onto FPGAs, it is expected that physical design tools will be extensively stressed to deliver highly optimized solutions within practical run-time budgets.

Compile time has recently been recognized as an important issue for FPGAs, while there are designers that are willing to afford a reduction in the quality of results (e.g. a penalty in performance) in exchange for a high-speed compilation[64]. Moreover, as the capacity of FPGA devices and the size of designs grow, there is a great interest for reducing the execution run-time of CAD tools that perform application's implementations onto reconfigurable platforms.

To keep run-time in check, the two main companies offering high-capacity FPGAS, Xilinx and Altera, have been continuously optimizing their CAD tools. Even though that motion alleviated the run-time pressure, it is unlikely that those algorithm engineering efforts can be sustained at the rate required by several more generations of Moore's

Law. Continuous technology scaling, without comparable scaling of execution run-time for application implementation onto FPGA devices, is expected to lead to a run-time crisis. This crisis, among others, manifests itself as a reduction in productivity and the corresponding increase in engineering costs. Based on relevant research approaches, there are three ways to reduce the execution run-time of CAD tools, which can be classified as follows:

- Discourage flat compilation of the entire design, and instead force users to compile partitions of their designs incrementally and assemble the partitions. Even though this approach mitigates execution run-time, it imposes an increased design complexity and it does not allow the application of optimizations between partitions.
- Introduce faster single-threaded algorithms, which can achieve mentionable execution speedup with little, or no, sacrifice at quality of derived application implementation. This selection leads to mentionable speedups, however, it is not widely accepted as it cannot follow the exponential growth in the number of FPGA logic cells.
- Develop novel parallel algorithms to take advantage of the existing and upcoming multi-core processors. With the current market trend of increasing the number, rather than designing faster CPU cores, the utilization of parallel CAD algorithms promises to alleviate the run-time crisis. These algorithms allow the capacity of FPGA platforms, as well as the number of working processor cores, to scale simultaneously. Towards this direction, both Xilinx and Altera have started to implement parallel flavors for their CAD algorithms that offer mentionable execution speedups.

In order to reduce the design process complexity, several intermediate levels of abstractions are introduced. A top-down design methodology divides the whole design process into a number of distinct phases, as depicted in Figure 2.5. Starting from an application's netlist after synthesis and technology mapping (3-D platform agnostic algorithms), we proceed to application implementation onto the target 3-D reconfigurable platform. This procedure consists of three consecutive tasks, namely (i) application partitioning, (ii) placement and (iii) routing.

The application partitioning is crucial for achieving both higher performance and resource utilization ratios. To support this task, a number of algorithms and tools have been proposed. Next, the derived partitions are assigned to the available physical layers of the 3-D stack. During this step, one (or more) partition(s) is (are) assigned

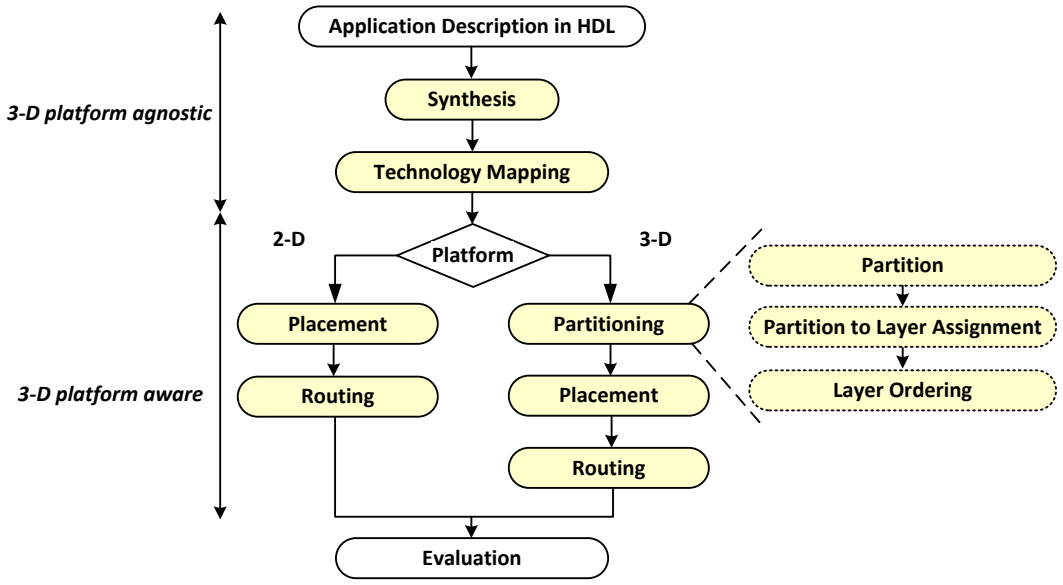

FIGURE 2.5: Toolflow for performing application mapping onto FPGA platforms.

to each of the 3-D FPGA layers. Usually, the primary objective in this step is to minimize the connections between already derived partitions. Then, the layers have to be ordered to form the 3-D stack. Similar to previous steps, this task aims to minimize the interlayer connections, and hence the TSVs, by appropriately ordering the available layers. Additionally, the layer ordering procedure can address a number of important design issues, such as improving the thermal distribution and shape of the 3-D stack, thereby enhancing its reliability. Even though hypergraph partitioning is a well-studied problem, these algorithms rarely lead to sufficient results because they do not take into consideration inherent limitations and constraints posed by the underline 3-D platform (presented in more detail in Section 2.2.1). Specifically, the main limitation of such approach lies on addressing solely the partition problem, while dismissing the partition-to-layer assignment and layer ordering.

In order to highlight the importance of partition, partition-to-layer assignment and layer ordering subproblems, Figure 2.6 gives an example of a digital circuit implemented onto a 3-D platform. Two alternative 3-D stacks (shown in Figures 2.6(b) and 2.6(c)) are evaluated after a successful application partitioning into four segments, namely A, B, C and D (as depicted in Figure 2.6(a)). The evaluation of the derived results for this analysis is based on the *net-cut* parameter. This term refers to the application's routing networks, running between different partitions (layers). Even though the 3-D stacks depicted in this figure are retrieved from the same application partitioning, the variations in layer assignment and layer ordering lead to different net-cuts. Furthermore, the total cuts between consecutive layers $\left(\sum_{i=1}^{i=3}(cut_i)\right)$, which correspond to the number of TSV connections, is also affected by the output of partitioning problem.

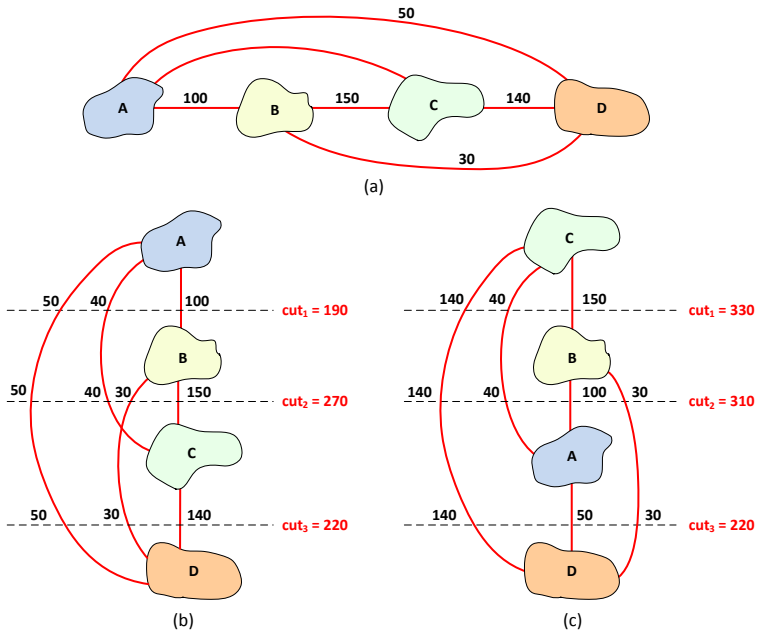

FIGURE 2.6: Task graph for application implementation onto 3-D architecture: (a) application partition and (b)-(c) alternative 3-D stacks derived with different selections at partition to layer assignment and layer ordering algorithms.

Finally, the last step deals with the application's placement and routing (P&R) onto the target 3-D FPGA. Each of these tasks is performed simultaneously for all of the device layers, such that the proper constraints are propagated among the layers. In order to achieve a solution that balances the utilization of hardware resources with the performance enhancement, the interlayer connections should be utilized only for timing critical routing paths.

Upcoming subsections provide additional details about the platform-aware (3-D) algorithms for performing application mapping onto target architectures.

2.2.1 Application Partitioning

For many existing and emerging applications in VLSI, producing efficient partitioning is of great importance. The problem consists of partitioning the vertices of a hypergraph into k roughly equal parts, such that the number of hyperedges connecting vertices on different parts is minimized[2].

[2] A hypergraph is a generalization of a graph, where the set of edges is replaced by a set of hyperedges. Specifically, a hyperedge extends the notion of an edge by allowing more than two vertices to be connected by a hyperedge. Formally, a hypergraph $H = (V, E^h)$ is defined as a set of vertices VE and a set of hyperedges E^h, where each hyperedge is a subset of the vertex set V and the size of a hyperedge is the cardinality of that subset.

Circuit partitioning is NP-hard problem[65]. That is, as the problem size grows linearly, the effort needed to find an optimal solution grows faster than any polynomial function. To date, there is no known polynomial-time, globally optimal algorithm for hypergraph partitioning. However, several efficient heuristics have been developed. These algorithms can retrieve high-quality circuit partitioning solutions in low-order polynomial time. Typical examples of such algorithms are the Fiduccia-Mattheyses (FM)[66], the Kernighan-Lin (KL)[67], the hMetis[65], the annealing/tabu[68], as well as their extensions.

Regarding the partitioning algorithms for 3-D FPGA platforms[54, 55, 65] the majority of them focuses on retrieving a min-cut solution[3]. Rather than utilizing many TSVs in order to achieve the maximum performance improvement, these min-cut partitioning algorithms exploit the potential reduction of utilized vertical connectivity. Even though this goal is acceptable for multi-chip devices, it's rarely an efficient approach in the context of 3-D architectures, since the electrical characteristics of TSVs are much better than the corresponding characteristics of the off-chip connections. A first order comparison between a routing wire in 45nm technology and a TSV can be found in[69].

On the other hand, new approaches such as the max-cut partitioning, targeting 3-D FPGA platforms, can lead to reductions in both delay and power consumption[54, 68], compared to the conventional min-cut approach. These gains occur mainly due to the higher and more efficient utilization of fabricated TSVs, which in turn leads to shorter wire-lengths for critical nets. Note that the emphasis of such approaches cannot be focused solely to the maximization of TSV utilization, because it might lead to unroutable designs (due to routing congestion problems).

2.2.2 Placement in 3-D Architectures

Upon completion of the partitioning phase, the application's netlist is assigned to a particular place on the FPGA through the placement task. Placement is an essential step in EDA, since it deals with the slot assignment problem (determine exact locations for various circuit components within the FPGA's area). Application's placement onto FPGA platforms can take hours, or even days, depending on the complexity of these designs, since placement is though to be the most time-consuming processes in physical implementation flows for reconfigurable architectures. This problem becomes even worst in the 3-D domain, since these platforms contain more resources compared to the 2-D FPGAs.

[3]In graph theory, a minimum cut of a graph is a cut (a partition of the vertices of a graph into two disjoint subsets that are joined by at least one edge) whose cut set has the smallest number of edges (unweighted case), or smallest sum of weights possible.

A good quality placement is essential to the overall design's quality, since it influences among others various design metrics, such as the interconnect delay, the congestion, the wirelength, as well as the power consumption. While there exists a lot of previous research on placement algorithms for improving these metrics, very few of them have pay equal importance to the minimization of the execution run-time.

The majority of existing techniques for 3-D circuit placement are summarized as follows:

- Partitioning-based algorithms, where the netlist and the FPGA's area are divided into smaller sub-netlists and sub-regions, respectively, according to cut-based cost functions. This process is repeated until each sub-netlist and sub-region is small enough to be handled efficiently. An example of this approach is min-cut placement[70].
- Analytic techniques model the placement problem using an objective (cost) function, which can be maximized or minimized via mathematical analysis. The objective can be quadratic or otherwise non-convex. Examples of analytic techniques include quadratic placement and force-directed placement[71].
- Stochastic algorithms, which perform randomized moves in order to optimize the cost function. A typical example of such an approach is the usage of simulated annealing algorithm[53].
- Techniques that are based on evolution algorithm. Such approaches aim to perform a more effective search space exploration, whereas their inherent parallelism can be exploited by the underlying multi-core architectures for reducing the execution run-time.

The functionality of these algorithms is identical to those proposed for the corresponding 2-D platforms, as examined in Section 1.2.3. The main difference for the 3-D domain concerns the customization of their cost functions (e.g. extend the bounding box to bounding cube for wire-length minimization) in order to take into consideration inherent constraints posed by the 3-D domain.

2.2.3 Routing in 3-D Architectures

Following placement, the routing algorithm determines a path to route the signals to all the resources. The most important objective of routing is to complete all the required connections. Other objectives, such as reducing the routing wirelength and ensuring each net satisfies its required timing budget, have become essential for modern routers.

Research concerning FPGA routing has received considerable attention in the literature. Routing is typically a very complex combinatorial problem. To make it manageable, we usually adopt a two-stage approach of global routing followed by detailed routing. Global routing partitions the routing region into tiles and decides tile-to-tile paths for all nets while attempting to optimize some given objective function (e.g., total wirelength and circuit timing). Subsequently, guided by the paths obtained in global routing, detailed routing assigns actual tracks and vias to the nets.

As we know, routing resources are prefabricated, and consequently quite limited, in the FPGA platforms. This problem becomes even more crucial in the 3-D domain concerning the availability of interlayer connections (TSVs). In order to reduce execution run-time, global routing tries to reduce the propagation delay for each net, while simultaneously balancing the channel congestion and minimizing the channel density. Note that in global routing the exact wire segments are not chosen yet. It is during the detailed routing that each net is assigned to a particular wire segment within a given channel.

A smaller FPGA with a narrow channel width is typically less expensive and exhibits better performance than a larger FPGA. Hence, detailed routing aims mainly to minimize the overall channel width required to route all nets. In addition, due to the large parasitic capacitance and resistance of programmable switches (found inside SBs), it takes significant amount of time to propagate a signal from the source of the net to its most distant sink. As a result, another parameter which significantly affects the performance of the design is the spread of the routing path over the device's area.

The main difference of routing algorithms targeting 3-D reconfigurable architectures, as compared to the rest of the 2-D implementations, is the benefit of exploiting the inherent flexibility provided by the third dimension. To do so, the interlayer connections must be employed specifically for realizing connections of critical or near critical paths. This can be enforced in the detailed routing, by constructing a routing graph with the vertical links embedded as high cost edges. In such graph, the input/output pins, as well as the logic blocks, are represented as vertices with a specific cost associated with them, while edges correspond to the connections between them, as depicted in Figure 2.7.

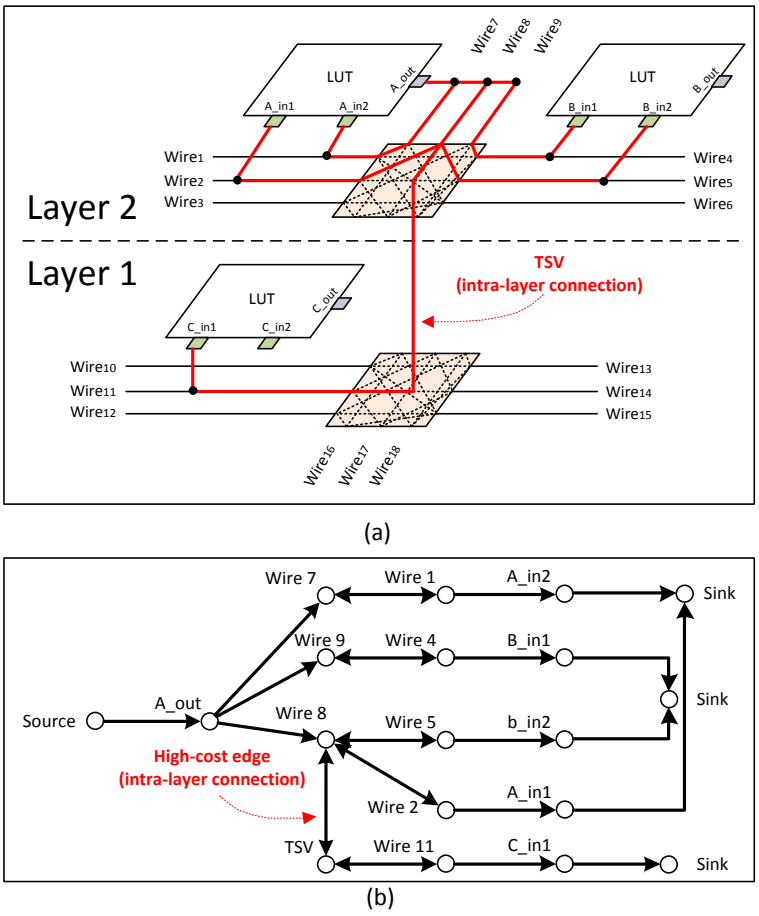

FIGURE 2.7: Illustration of the routing graph construction: (a) netlist routing and (b) corresponding routing graph.

2.3 Toolflows targeting 3-D FPGAs - Overview of our approach

2.3.1 Motivation

This section summarizes the main features found in available tools that perform application implementation onto 3-D FPGA platforms and introduces our approach to the subject. Table 2.1 provides a qualitative comparison among recently proposed academic toolflows for application implementation onto 3-D FPGAs. A number of conclusions can be derived based on this analysis. Among others, the majority of existing tools focus solely onto homogeneous 3-D devices. Although the concept of homogeneity is inherent at FPGAs, the opportunity to integrate either different process technologies (e.g., logic, memory, etc)[72] and/or to design domain-specific 3-D platforms[73] can achieve

mentionable gains, as discussed in Sections 2.1.1 and 2.2.1. Moreover, none of the existing CAD algorithms are tunable according to requirements posed either by target application, or the employed 3-D platform.

In addition, it is worthwhile to mention that the available algorithms[74–81] rely on straight-forward extensions of existing 2-D tools, which cannot fully exploit the benefits of 3-D technology. On the contrary, physical design in the 3-D realm requires fresh approaches (e.g. new algorithms and cost functions), that can benefit from the architectural features provided by these devices. For instance, the algorithms for netlist placement mainly rely on simulated annealing, which is a sequential algorithm that cannot benefit from the existing many-core CPUs. As presented in Section 2.2, there is a great need to develop novel parallel algorithms to take advantage of the current market trend of increasing the number of CPU cores. Hence, one might expect that by manipulating more advanced algorithms, we can achieve mentionable reduction at execution run-time and consequently alleviate the time-to-market pressure.

Furthermore, the availability of source code is another interesting property, since it provides engineers the opportunity to modify and/or extent appropriately the functionality of these tools in order to take into consideration additional features. For instance, more advanced 3-D reconfigurable architectures can be supported (e.g. consisted of mixed digital/analog circuits) if the additional architectural properties are appropriately modeled. Note that such a feature is very important especially for educational purposes in topics related to architecture design, as well as CAD algorithm development.

Application implementation onto FPGA platforms can take hours, or even days, depending on the complexity of these designs. One of the most time-consuming steps in the FPGA CAD flow is application's placement. As we have emphasized before, a good quality placement is essential to the overall design's quality, since it influences among others the interconnect delay, the congestion, the wirelength, as well as the power consumption [82]. While there exists a lot of previous research on placement algorithms for improving application's maximum operating frequency, power/energy dissipation and the wiring area occupied by a circuit, very few of them have as their primary goal the minimization of the tool's execution run-time. Compile time has recently been recognized as an important issue for FPGAs[83] and there are designers willing to afford a reduction in the quality of results (e.g. a penalty in performance) in exchange for a high-speed compilation[82]. As a result, as the capacity of FPGA devices and the size of designs grow, there is a great interest for performing fast application's implementations onto re-configurable platforms.

Our approach is to introduce a novel placer, based on Ant Colony Optimization (ACO), targeting heterogeneous 3-D reconfigurable architectures. For evaluation purposes, this algorithm was also implemented as a stand-alone open-source tool, which was integrated onto the 3-D MEANDER open-source design framework (see Table 2.1). The inherent parallelism found in our algorithm is exploited by today's multi-core architectures to reduce the execution run-time. This will allow the capacity of the FPGAs to scale, as the number of working processors scale as well, avoiding the run-time crisis. Also, by manipulating the cost function in our algorithm, one can easily enforce all the legality constrains and implement new FPGA architectures. For example, by taking into account the structure of a heterogeneous interconnect fabric of an FPGA's architecture, our approach can maximize the performance by a more efficient utilization of the fabricated TSVs, something that the existing state-of-the-art algorithms cannot achieve. Finally, our approach is able to combine partitioning with placement and thus has the potential to arrive at placement solutions with better quality. Even though the solution space from such an approach is huge, the inherent parallelism of our algorithm is the key that will allow for efficient exploration of ever growing solution spaces as the multi-core computational power of conventional processors increases.

Feature		MEVA-3D [74]	TPR [75]	TPR [76]	VPR3D [77]	3D-Tree [78]	3-D MEANDER (previous version [73] [79] [80])
Architecture	3-D technology	TSV	TSV	TSV	SSIT	TSV	wirebonding, TSV, SSIT
	Heterogeneous layers	yes	no	no	no	no	yes
	Interlayer routing	uniform	uniform	uniform	uniform	uniform	uniform, full-custom
Algorithm	CAD tuning	no	no	no	no	no	application-specific
	Partition engine	N/A	sim. anneal.	hMetis	hMetis	hMetis	tabu
	Partition objective	min-cut	min-cut	min-cut	min-cut	min-cut	constrained max-cut
	Placement engine	N/A	sim. anneal.	sim. anneal.	sim. anneal.	sim. anneal.	ACO
	Routing engine	N/A	pathfinder	pathfinder	pathfinder	pathfinder	pathfinder
Evaluation	Wirelength	yes	yes	yes	yes	yes	yes
	Delay	yes	yes	yes	yes	yes	yes
	Power	yes	no	no	yes	yes	yes
Other	Graphical interface	no	no	no	yes	yes	yes
	Public available	no	yes	no	no	no	yes

TABLE 2.1: Qualitative comparison among toolflows for 3-D reconfigurable platforms.

2.3.2 Architecture Template of targeted 3-D FPGA

In this section we introduce the proposed architectural template targeting to alleviate the impact of long wire-lengths. The concept of this architecture is depicted schematically in Figure 2.8, where the CLBs are assigned to different device layers. Note that the architectural template discussed in this paper is orthogonal to the rest architectural approaches for 3-D FPGAs.

The routing connectivity between layers is actually implemented through vertically aligned 3-D switch boxes (SBs). Previous studies have shown the efficiency of designing

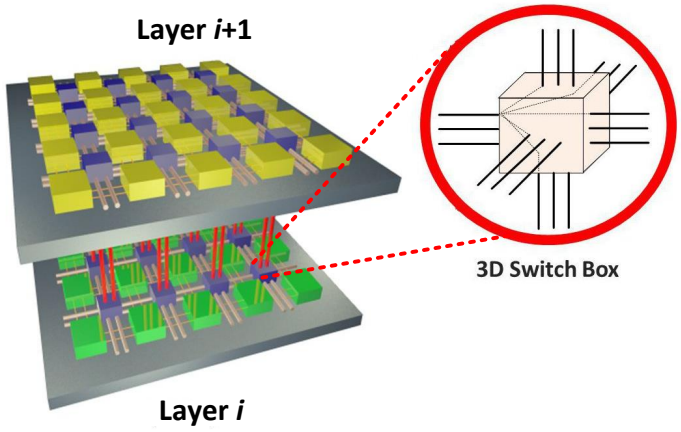

FIGURE 2.8: Abstract view of the proposed 3-D FPGA architecture.

such components[73]. However, since 3-D SBs impose the usage of TSVs, their careful spatial assignment is a crucial task for achieving performance and area efficient solutions. Figure 2.9 depicts schematically the template of the proposed architecture. The architectural organization of these layer might differ (i.e. assuming a heterogeneous 3-D FPGA).

Specifically, each layer of the introduced 3-D FPGA is based on island-style architecture, where the logic blocks are arranged in an array (square or rectangle) surrounded by horizontal and vertical routing channels. The communication between resources assigned to different layers is provided through vertically aligned TSVs. These TSVs are actually implemented inside the SBs, which are appropriately extended in order to be aware of the third dimension[73]. Such kind of connectivity provides routing paths (depicted with dotted lines in Figure 2.9) between SBs assigned to adjacent layers with the same (x, y) coordinates.

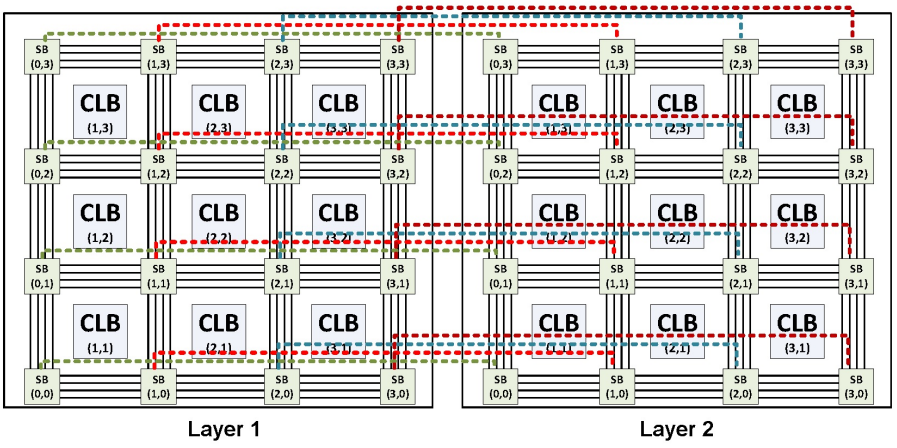

FIGURE 2.9: Architectural template of our proposed 3-D architecture with two layers interconnected through TSVs (dotted lines).

Table 2.2 summarizes the main properties of the employed TSV technology[84], whereas regarding the modeling of the remaining hardware resources (e.g. routing wires, transistors, etc), we follow a similar approach to the one found in relevant literature[85]. Note that the selection of the employed values for architectural components do not affect the generality of the introduced solution, which is also applicable to other flavors of 3-D integration (such as the 2.5-D provided by Xilinx[86]).

Property	Value
Length:	4-9 um
Diameter:	1.2 um
Minimum pitch (P_{TSV}):	4 um
Resistance:	0.35 Ω
Capacitance:	2.5 fF

TABLE 2.2: Properties of the employed TSVs[84].

2.4 Summary

Three dimensional (3-D) chip stacking is a revolutionary technology able to achieve improved performance at a reduced power consumption and smaller footprint than conventional two dimensional (2-D) implementations. The design of such 3-D platforms is a complex problem that demands architecture-level exploration and customization. As a result there is an ever growing need for more efficient and faster CAD tools addressing these aspects. Additionally, the application mapping onto 3-D platforms should be supported by algorithms that, apart from quality, are able to produce results in reasonable execution run-time, to keep up with the fast growing capacity of FPGA devices and the size of the designs. In this chapter we summarized a number of state-of-the-art solutions related to 3-D reconfigurable architectures and CAD algorithms and presented a brief overview of our approach.

Chapter 3

A Novel Placement Algorithm based on Ant Colony Optimization

Ants exhibit complex social behaviors that have long since attracted the attention of humans. Many biologists study these behaviors of ants in detail and have stumbled into some interesting results. One of the most surprising behavioral patterns exhibited by ants is the ability of certain ant species to find what computer scientists call shortest paths. Biologists have shown experimentally that this is possible by exploiting communication based only on pheromones, an odorous chemical substance that ants may deposit and smell. It is this behavioral pattern that inspired computer scientists to develop algorithms for the solution of optimization problems[87].

In this chapter we introduce our proposed algorithm for addressing the placement problem at 3-D reconfigurable architectures. The introduced approach relies on Ant Colony Optimization (ACO) algorithm and mimics the aforementioned foraging behavior of ants in order to find a high quality placement in regard to legality constrains and optimization goals. Ant colony optimization (ACO) is a population-based metaheuristic, classified as a Swarm intelligence (SI) method[1], that can be used to find approximate

[1]Swarm intelligence (SI) is the collective behavior of decentralized, self-organized systems, natural or artificial. The concept is employed in work on artificial intelligence. The expression was introduced by Gerardo Beni and Jing Wang in 1989, in the context of cellular robotic systems[88–90]. SI systems consist typically of a population of simple agents or boids interacting locally with one another and with their environment. The inspiration often comes from nature, especially biological systems. The agents follow very simple rules, and although there is no centralized control structure dictating how individual agents should behave, local, and to a certain degree random, interactions between such agents lead to the emergence of "intelligent" global behavior, unknown to the individual agents. Examples in natural systems of SI include ant colonies, bird flocking, animal herding, bacterial growth, fish schooling and microbial intelligence.

solutions to difficult optimization problems. In ACO, a set of software agents called artificial ants search for good solutions to a given optimization problem.

The rest of the chapter is organized as follows: Section 3.1 introduces the field of Ant Colony Optimization. Section 3.2 provides an overview of the existing ACO algorithms, while Section 3.3 presents our proposed implementation. Finally in section 4.3 we conclude the chapter.

3.1 Introduction to ACO - From Real to Artificial Ants

Ant colonies, and more generally social insect societies, are distributed systems that, in spite of the simplicity of their individuals, present a highly structured social organization and as a result can accomplish complex tasks using the collective intelligence of the group. The field of ant based algorithms derived from the observation of real ants' behavior, and uses these models as a source of inspiration for the design of novel algorithms for the solution of optimization and distributed control problems. The main idea is that the self-organizing principles which allow the highly coordinated behavior of real ants can be exploited to coordinate populations of artificial agents that collaborate to solve computational problems. Several different aspects of the behavior of ants have inspired different kinds of ant algorithms. Examples are foraging, division of labor, brood sorting, and cooperative transport. In all these examples, ants coordinate their activities via stigmergy, a form of indirect communication mediated by modifications of the environment. The idea behind ant algorithms is to use a form of artificial stigmergy to coordinate societies of artificial agents[87].

One of the most successful examples of ant based algorithms is known as Ant Colony Optimization (ACO). ACO is inspired by the foraging behavior of ants, and targets discrete optimization problems. As we mentioned earlier, ants communicate indirectly by modifying their environment. This form of communication is called *stigmergy*. In fact the visual perceptive faculty of many ant species is only rudimentarily developed and there are many ant species that are completely blind. As a result most of the communication among individuals is based on the use of chemicals exuded by the ants. Those chemicals are called *pheromones*.

3.1.1 The Double Bridge Experiment

The original inspiration for the ACO algorithms comes from an experiment performed by Jean-Louis Deneubourg and colleagues, called the double-bridge experiment. Although

ACO has grown to become a fully fledged algorithmic framework and now includes many components that are no longer related to real ants, we report here the double-bridge experiment for its historical value.

The foraging behavior of many ant species, as, for example, I. humilis (Deneubourg, Aron, Goss, & Pasteels, 1990; Goss et al., 1989), Linepithema humile, and Lasius niger (Bonabeau et al., 1997), is based on indirect communication mediated by pheromones. While walking from food sources to the nest and vice versa, ants deposit those pheromones on the ground, forming in this way a pheromone trail. Ants can smell the pheromone trails and they tend to choose, probabilistically, paths marked by strong pheromone concentrations. The pheromone trail-laying and -following behavior of some ant species was the topic of investigation on the double bridge experiment. The setup was as follows: They used a double bridge connecting a nest of ants of the Argentine ant species I. humilis and a food source. Then they ran experiments varying the ratio $r = l_l/l_s$ between the length of the two branches of the double bridge, where l_l was the length of the longer branch and l_s the length of the shorter one.

In the first experiment the bridge had two branches of equal length ($r = 1$, as shown in Figure 3.1 (a)). At the start, ants were left free to move between the nest and the food source and the percentage of ants that chose one or the other of the two branches were observed over time. The outcome was that (Figure 3.2 (a)), although in the initial phase random choices occurred, eventually all the ants used the same branch. This result can be explained as follows: When a trial starts there is no pheromone on the two branches. Hence, the ants do not have a reference and they select with the same probability any of the branches. Yet, because of random fluctuations, a few more ants will select one branch over the other. Because ants deposit pheromone while walking, a larger number of ants on a branch results in a larger amount of pheromone on that branch. This larger amount of pheromone in turn stimulates more ants to choose that branch again, and so on until finally the ants converge to one single path. This *autocatalytic* or *positive feedback* process is, in fact, a nice example of the self-organizing behavior of the ants. It is also an example of stigmergic communication. Ants coordinate their activities, exploiting indirect communication mediated by modifications of the environment in which they move.

In the second experiment, the length ratio between the two branches was set to $r = 2$, so that the long branch was twice as long as the short one (Figure 3.1 (b)). In this case, in most of the trials, after some time had past, all the ants chose to use only the short branch (Figure 3.2 (b)). As in the first experiment, ants leave the nest to explore the environment and arrive at a decision point where they have to choose one of the two branches. Because the two branches initially appear identical to the ants, they choose

randomly. Therefore, it can be expected that, on average, half of the ants choose the short branch and the other half the long branch, although stochastic oscillations may occasionally favor one branch over the other. However, this experimental setup presents a remarkable difference with respect to the previous one: because one branch is shorter than the other, the ants choosing the short branch are the first to reach the food and to start their return to the nest. But then, when they must make a decision between the short and the long branch, the higher level of pheromone on the short branch will bias their decision in its favor. Therefore, pheromone starts to accumulate faster on the short branch, which will eventually be used by all the ants because of the autocatalytic process described previously. When compared to the experiment with the two branches of equal length, the influence of initial random fluctuations is much reduced, and *stigmergy*, *autocatalysis*, and *differential path length* are the main mechanisms at work (see Figure 3.3). Interestingly, it can be observed that, even when the long branch is twice as long as the short one, not all the ants use the short branch, but a small percentage may take the longer one. This may be interpreted as a type of *path exploration*.

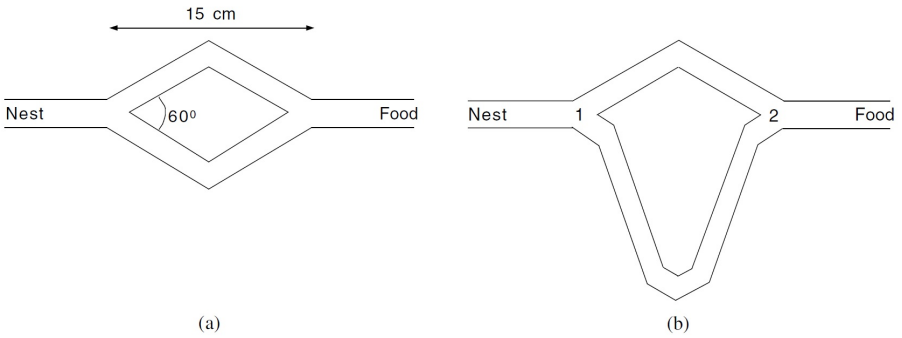

FIGURE 3.1: Experimental setup for the double bridge experiment. (a) Branches have equal length. (b) Branches have different length.[87].

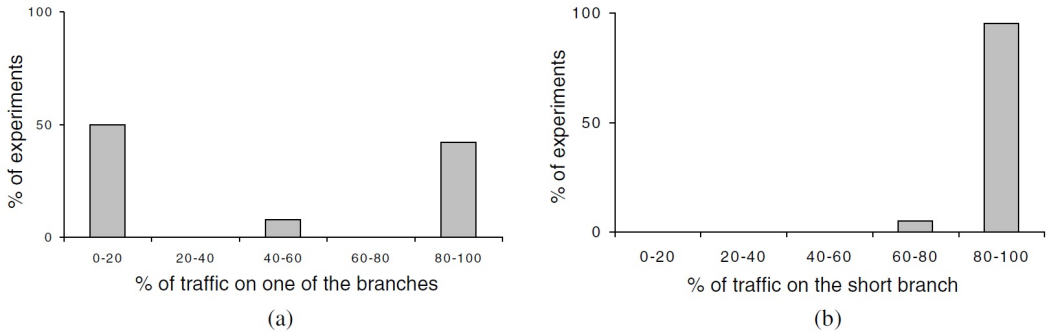

FIGURE 3.2: Results obtained in the double bridge experiment. (a) Results for the case in which the two branches have the same length (b) Results for the case in which one branch is twice as long as the other.[87].

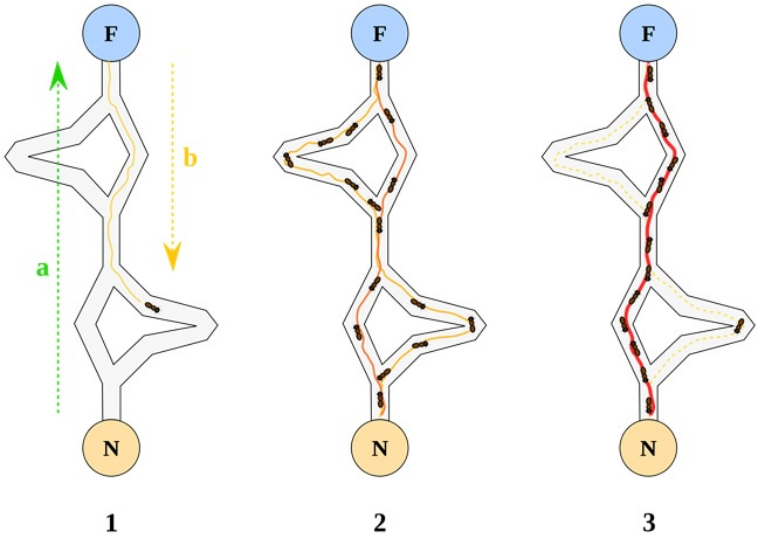

FIGURE 3.3: Stigmergy, autocatalysis and differential path length at work.

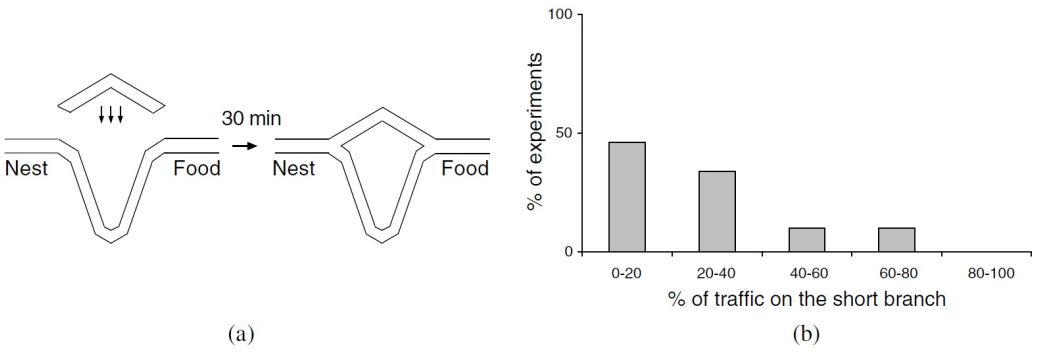

FIGURE 3.4: In this experiment initially only the long branch was offered to the colony. (a) The initial experimental setup and the new situation after 30 minutes, when the short branch was added. (b) In the great majority of the experiments, once the short branch is added the ants continue to use the long branch[87].

It is also interesting to see what happens when the ant colony is offered, after convergence, a new shorter connection between the nest and the food. This situation was studied in an additional experiment in which initially only the long branch was offered to the colony and after 30 minutes the short branch was added (Figure 3.4). In this case, the short branch was only selected sporadically and the colony was trapped on the long branch. This can be explained by the high pheromone concentration on the long branch and by the slow evaporation of pheromone. In fact, the great majority of ants choose the long branch because of its high pheromone concentration, and this autocatalytic behavior continues to reinforce the long branch, even if a shorter one appears. Pheromone evaporation, which could favor exploration of new paths, is too slow: the lifetime of the pheromone is comparable to the duration of a trial, which

means that the pheromone evaporates too slowly to allow the ant colony to "forget" the suboptimal path to which they converged so that the new and shorter one can be discovered and learned[87].

3.1.2 From the Natural Inspiration to the Artificial Model

The double bridge experiments show clearly that ant colonies have a built-in optimization capability. By the use of probabilistic rules based on local information they can find the shortest path between two points in their environment. Taking inspiration from the double bridge experiments, it is possible to design artificial ants that, by moving on a graph modeling the double bridge, can find the shortest path between two nodes corresponding to the nest and to the food source, as depicted in Figure 3.5.

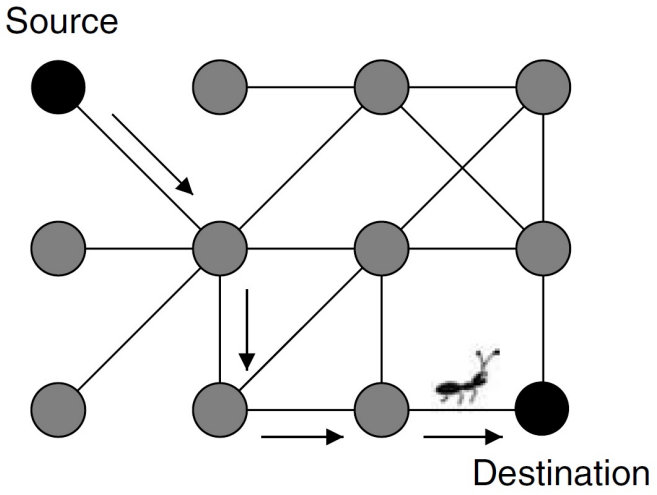

FIGURE 3.5: Ants build solutions, that is, paths from a source to a destination node.[87].

Ants initially wander randomly around their environment. Once food is located an ant will begin laying down pheromone in the environment. Numerous trips between the food and the colony are performed and if the same route is followed that leads to food then additional pheromone is laid down. Pheromone decays in the environment, so that older paths are less likely to be followed. Other ants may discover the same path to the food and in turn may follow it and also lay down pheromone. A positive feedback process routes more and more ants to productive paths that are in turn further refined through use[91].

The objective of the strategy is to exploit historic and heuristic information to construct candidate solutions and fold the information learned from constructing solutions into the history. Solutions are constructed one discrete piece at a time in a

probabilistic step-wise manner. The probability of selecting a component is determined by the heuristic contribution of the component to the overall cost of the solution and the quality of solutions from which the component has historically known to have been included. History is updated proportionally to the quality of the best known solution and is decreased proportionally to the usage[91].

ACO algorithms can be applied to any combinatorial optimization problem for which a solution construction procedure can be conceived.

Combinatorial Optimization Problems

Combinatorial optimization problems involve finding values for discrete variables such that the optimal solution with respect to a given objective function is found[87]. Many optimization problems of practical and theoretical importance are of combinatorial nature. Examples involve routing problems (Traveling Salesman Problem, Vehicle Routing, FPGA Routing), assignment problems (Quadratic Assignment, Graph Coloring), scheduling problems (Job Shop, Total tardiness), subset problems (Graph Coloring, Multiple Knapsack, Set Covering), machine learning problems and many more. A combinatorial optimization problem is either a maximization or a minimization problem which has associated a set of problem instances.

More formally, an instance of a combinatorial optimization problem Π is a triple (S, f, Ω) where S is the set of candidate solutions, f is the objective function which assigns an objective function value $f(s)$ to each candidate solution $s \in S$, and Ω is a set of constraints. The solutions belonging to the set $\widetilde{S} \subseteq S$ of candidate solutions that satisfy the constraints Ω are called feasible solutions. The goal is to find a globally optimal feasible solution s^*. For minimization problems this consists in finding a solution $s^* \in \widetilde{S}$ with minimum cost, that is, a solution such that $f(s^*) \leq f(s)$ for all $s \in \widetilde{S}$. For maximization problems one searches for a solution with maximum objective value, that is, a solution with $f(s^*) \geq f(s)$ for all $s \in \widetilde{S}$ [2].

Unfortunately most of the combinatorial optimization problems (or at least the interesting ones) are NP-complete[3]. As a result, exact algorithms for the solution of such problems need, in the worst case, exponential time to find the optimum, making them infeasible for practical applications. If optimal solutions cannot be efficiently obtained

[2]Note that maximizing a function over its argument is equivalent to minimizing that function over the same argument with a sign change. So maximization and minimization are computationally equivalent.

[3]NP-complete is the complexity class of decision problems for which answers can be checked for correctness, given a certificate, by an algorithm whose run time is polynomial in the size of the input (that is, it is NP) and no other NP problem is more than a polynomial factor harder (NP-hard). Informally, a problem is NP-complete if answers can be verified quickly, and a quick algorithm to solve this problem can be used to solve all other NP problems quickly[92].

in practice, the only possibility is to trade optimality for efficiency. In other words, the guarantee of finding optimal solutions can be sacrificed for the sake of getting very good solutions in polynomial time. Approximate algorithms, also loosely called heuristic methods or simply heuristics, seek to obtain good, that is, near-optimal solutions at relatively low computational cost without being able to guarantee the optimality of solutions. A metaheuristic is a set of algorithmic concepts that can be used to define heuristic methods applicable to a wide set of different problems. In other words, a metaheuristic can be seen as a general-purpose heuristic method designed to guide an underlying problem-specific heuristic (e.g., a local search algorithm or a construction heuristic) toward promising regions of the search space containing high-quality solutions. A metaheuristic is therefore a general algorithmic framework which can be applied to different optimization problems with relatively few modifications to make them adapted to a specific problem. The use of metaheuristics has significantly increased the ability of finding very high-quality solutions to hard, practically relevant combinatorial optimization problems at a reasonable time. This is particularly true for large and poorly understood problems. Ant colony optimization is a metaheuristic in which a colony of artificial ants cooperate in finding good solutions to difficult discrete optimization problems.

Applying Ant Colony Optimization

As we mentioned earlier, Ant Colony Optimization can be applied to any combinatorial optimization problem for which a solution construction procedure can be conceived. Artificial ants build solutions by performing randomized walks on a fully connected weighted graph $G_C = (C, L)$ whose nodes are the components C, and the set L fully connects the components C [4]. The graph G_C is called construction graph and elements of L are called connections. To build a solution each ant utilizes information stored at each component or connection. This information is the pheromone trail and the heuristic information, whose values are used by the algorithm's probabilistic decision rule to make decisions on how to move on the graph.

We associate with every component $c_i \in C$ or connection $l_{ij} \in L$ a pheromone trail τ (τ_i if associated with components, τ_{ij} if associated with connections), and a heuristic value η (η_i and η_{ij} , respectively). The pheromone trail is used to encode a long-term memory about the entire search process, and is updated by the ants themselves. Differently, the heuristic value, often called heuristic information, represents a priori information about the problem instance or run-time information provided by a source

[4] If the graph representing our problem is not fully connected we can convert it to a fully connected one simply by assigning a huge weight value to the non-existing edges.

different from the ants. In many cases η is the cost, or an estimate of the cost, of adding the component or connection to the solution under construction. The problem's constraints $\Omega(t)$ are built into the ants' constructive heuristic. In most applications, ants construct feasible solutions. However, sometimes it may be necessary or beneficial to also let them construct infeasible solutions.

Succeeding the construction of a solution, each ant evaluates the quality of its solution and based on that quality it decides on how much pheromone to deposit on the components or connections used. Subsequent ants use this information to guide their search towards promising solutions. Thus, good-quality solutions emerge as the result of the collective interaction among the ants. Summarizing:

- A colony of artificial ants move *concurrently*, *asynchronously* and *independently* on the construction graph, incrementally building solutions to the optimization problem.
- In order to move around the graph they use a *stochastic* local decision policy that makes use of *pheromone trails* and *heuristic information*.
- Based on the quality of the solution, each ant *deposits* different amounts of pheromone. In doing so they adaptively modify the way the problem is represented and perceived by other ants.

3.1.3 The Ant Colony Optimization Metaheuristic

In this section we present in more detail the aforementioned procedure. In Algorithm 1 we present the ACO metaheuristic in pseudo-code. Informally, an ACO algorithm can be imagined as the interplay of three procedures: ConstructAntsSolutions, UpdatePheromones, and DaemonActions[87].

Algorithm 1 ACO Metaheuristic pseudo-code.

```
1: procedure ACOMetaheuristic
2:     ConstructAntsSolution
3:     UpdatePheromones
4:     DaemonActions
5: end procedure
```

ConstructAntsSolutions manages a colony of ants that concurrently and asynchronously visit adjacent states of the considered problem by moving through neighbor nodes of the problem's construction graph G_C. They move by applying a stochastic local decision policy that makes use of pheromone trails and heuristic information. In this way, ants incrementally build solutions to the optimization problem. Once an ant has built

a solution, the ant evaluates the solution that will be used by the UpdatePheromones procedure to decide how much pheromone to deposit.

UpdatePheromones is the process by which the pheromone trails are modified. The trails value can either increase, as ants deposit pheromone on the components or connections they use, or decrease, due to pheromone evaporation. From a practical point of view, the deposit of new pheromone increases the probability that those components/-connections that were either used by many ants or that were used by at least one ant and which produced a very good solution will be used again by future ants. On the other hand, pheromone evaporation implements a useful form of "forgetting". It avoids a too rapid convergence of the algorithm toward a suboptimal region, therefore favoring the exploration of new areas of the search space.

Finally, the DaemonActions procedure is used to implement centralized actions which cannot be performed by single ants. Examples of daemon actions are the activation of a local optimization procedure, or the collection of global information that can be used to decide whether it is useful or not to deposit additional pheromone to bias the search process from a nonlocal perspective. As a practical example, the daemon can observe the path found by each ant in the colony and select one or a few ants (e.g., those that built the best solutions in the algorithm iteration) which are then allowed to deposit additional pheromone on the components/connections they used.

There are several possible termination criteria, which we may choose depending on the application:

- maximum CPU time elapsed,
- maximum number of solutions generated,
- percentage deviation from a lower/upper bound from the optimum value,
- maximum number of iterations without improvement in solution quality,
- other metaheuristic-dependent rules (e.g. empty neighborhood),
- or any other implementation-specific criterion that may be suitable.

3.2 Overview of ACO Algorithms

There are numerous variations of ACO algorithms, many of which have been successfully utilized to solve various real world problems[87, 93–97]. The first ACO algorithm was Ant System which provided the inspiration for a number of extensions that significantly

improved performance and are currently among the most successful ACO algorithms. In this section we will make a brief introduction to the main members of the ACO family:

3.2.1 Ant System

The Ant System (AS) algorithm works in two main phases: the ants' solution construction and the pheromone update.

Solution Construction

In AS, m artificial ants concurrently build a solution to the given problem. At each construction step, ant k applies a probabilistic action choice rule, called *random proportional rule*, to decide which connection (or component) to select next. The probabilistic step-wise construction of solution makes use of both history (pheromone) and problem-specific heuristic information to incrementally construct a solution piece-by-piece. Each connection can only be selected if it has not already been chosen (for most combinatorial problems). For those components, j, that can be selected from a given current component, i, their probability for selection by ant k is defined as[87, 91]:

$$p_{ij}^{k} = \frac{[\tau_{ij}]^{\alpha} \times [\eta_{ij}]^{\beta}}{\sum_{l \in N_i^k} [\tau_{il}]^{\alpha} \times [\eta_{il}]^{\beta}} \tag{3.1}$$

where τ_{ij} is the pheromone trail, η_{ij} is the heuristic information that is available a priori, α and β are two parameters which determine the relative influence of the pheromone trail and the heuristic information, and N_i^k is the feasible neighborhood of ant k (the components j that can be selected from a given current component, i). By this probabilistic rule, the probability of choosing a particular component increases with the value of the associated pheromone trail τ_{ij} and heuristic information η_{ij}. The role of the parameters α and β is the following: If $\alpha = 0$, then we only use the a priory heuristic information and our algorithm degrades to a stochastic greedy algorithm. On the other hand if $\beta = 0$ we only use the pheromone trails without any heuristic bias. This usually leads to rather poor results because there is no information to guide the initial constructions.

This random proportional rule presented in Equation 3.1 is analogous to the "Roulette Wheel" selection method[98] used in genetic algorithms in the sense that fittest individuals (the ones who in the past have produced hight quality solutions and as a result have higher pheromone values) have a larger share of the roulette wheel and as a result greater probability to be chosen again, where weakest individuals occupy smaller share

of the roulette wheel and have less probability to be chosen. A visual representation of the "Roulette Wheel" selection method is shown in Figure 3.6.

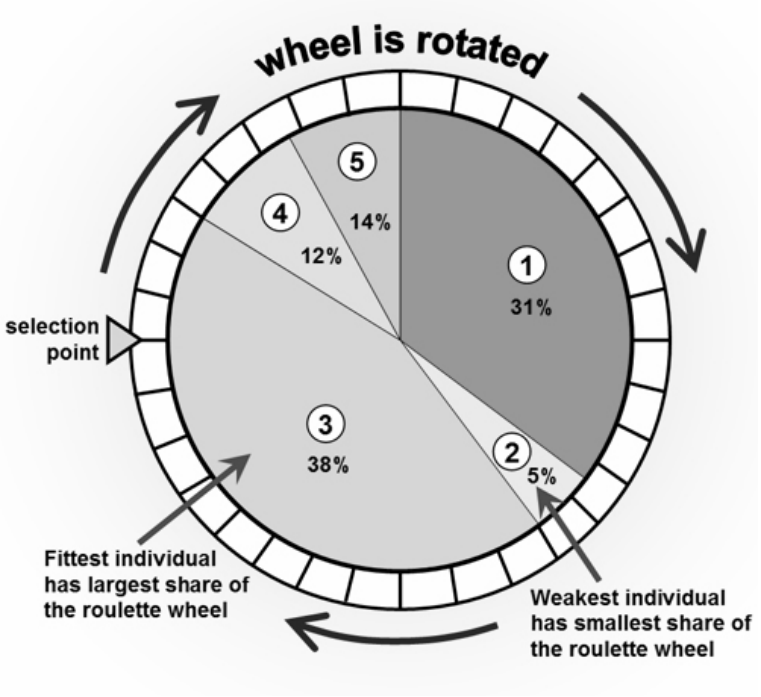

FIGURE 3.6: "Roulette Wheel" selection method.

Pheromone Trails Update

At the end of each iteration, after all ants have constructed their solutions, the pheromone trails are updated. This is done by first lowering the pheromone value on all the connections by a constant factor, and then adding pheromone on the connections the ants have used in their solutions. Pheromone evaporation is implemented by Equation 3.2:

$$\tau_{ij} \leftarrow (1 - \rho) \times \tau_{ij} \tag{3.2}$$

where ρ is the pheromone evaporation rate. Evaporation is used to avoid unlimited accumulation of the pheromone trails and it enables the algorithm to "forget" bad decisions previously taken. In fact, if a connection is not chosen by the ants, its associated pheromone value decreases exponentially in the number of iterations. After evaporation, all ants deposit pheromone on the connections they have selected in their solution using Equation 3.3:

$$\tau_{ij} \leftarrow \tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^{k} \tag{3.3}$$

where $\Delta\tau_{ij}^{k}$ is the amount of pheromone ant k deposits on the connections it has selected in his solution, and it's defined as follows:

$$\Delta\tau_{ij}^{k} = \begin{cases} \frac{1}{C^k} & \text{if connection } (i,j) \text{ belongs to } T^k \\ 0 & \text{otherwise} \end{cases} \tag{3.4}$$

where C^k is the cost of the solution T^k built by the k-th ant. By means of Equation 3.4, the better an ant's solution is, the more pheromone the connections belonging to this solution receive. In general, connections that are used by many ants and which are part of high quality solutions, receive more pheromone and are therefore more likely to be chosen by ants in future iterations of the algorithm[87].

3.2.2 Elitist Ant System

The Elitist Ant System (EAS) is a first improvement on the initial AS. The idea is to provide strong additional reinforcement to the connections belonging to best solution found since the start of the algorithm. This solution is denoted as T^{bs} (best-so-far solution) in the following. Note that this additional feedback to the best-so-far solution (which can be viewed as additional pheromone deposited by an additional ant called best-so-far ant) is another example of a daemon action of the ACO metaheuristic[87].

Pheromone Trails Update

The additional reinforcement of solution T^{bs} is achieved by adding a quantity $e = C^{bs}$ to its connections, where e is a parameter that defines the weight given to the best-so-far solutions T^{bs}, and C^{bs} is its cost. Thus, Equation 3.3 for the pheromone deposit becomes:

$$\tau_{ij} \leftarrow \tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^{k} + e\Delta\tau_{ij}^{bs} \tag{3.5}$$

where $\Delta\tau_{ij}^{k}$ is defined as in Equation 3.4 and $\Delta\tau_{ij}^{bs}$ is defined as:

$$\Delta\tau_{ij}^{bs} = \begin{cases} \frac{1}{C^{bs}} & \text{if connection } (i,j) \text{ belongs to } T^{bs} \\ 0 & \text{otherwise} \end{cases} \tag{3.6}$$

3.2.3 Rank-Based Ant System

Another improvement over AS is the rank-based version of AS (AS_{rank}). In AS_{rank} each ant deposits an amount of pheromone that decreases with its rank. Additionally, as in EAS, the best-so-far ant always deposits the largest amount of pheromone in each iteration[87].

Pheromone Trails Update

Before updating the pheromone trails, the ants are sorted by decreasing solution quality and the quantity of pheromone an ant deposits is weighted according to the rank r of the ant. In each iteration only the $(w-1)$ best ranked ants and the ant that produced the best-so-far solution (this ant does not necessarily belong to the set of ants of the current algorithm iteration) are allowed to deposit pheromone. The best-so-far solution gives the strongest feedback, with weight w, while the r-th best ant of the current iteration contributes to pheromone updating with the value $\frac{1}{C^r}$ multiplied by a weight given by $max\{0, w-r\}$. Thus, the AS_{rank} pheromone update rule is:

$$\tau_{ij} \leftarrow \tau_{ij} + \sum_{r=1}^{w-1} (w-r)\Delta\tau_{ij}^{r} + w\Delta\tau_{ij}^{bs} \tag{3.7}$$

where $\Delta\tau_{ij}^{r} = \frac{1}{C^r}$ and $\Delta\tau_{ij}^{bs} = \frac{1}{C^{bs}}$

3.2.4 $MAX - MIN$ Ant System

$MAX - MIN$ Ant System (MMAS) introduces four main modifications with respect to AS. First, it strongly exploits the best solution found: only either the iteration-best ant, that is, the ant that produced the best solution in the current iteration, or the best-so-far ant is allowed to deposit pheromone. Unfortunately, such a strategy may lead to a stagnation situation in which all the ants construct the same solution, because of the excessive growth of pheromone trails on connections of a good, although suboptimal, solution. To counteract this effect, a second modification introduced by MMAS is that it limits the possible range of pheromone trail values to the interval $[\tau_{min}, \tau_{max}]$. Third, the pheromone trails are initialized to the upper pheromone trail limit, which, together

with a small pheromone evaporation rate, increases the exploration of solutions at the start of the search. Finally, in MMAS, pheromone trails are reinitialized each time the system approaches stagnation or when no improved solution has been generated for a certain number of consecutive iterations[87].

Pheromone Trails Update

After all ants have constructed a solution, pheromones are updated by applying evaporation as in AS (Equation 3.2), followed by the deposit of new pheromone as follows:

$$\tau_{ij} \leftarrow \tau_{ij} + \Delta\tau_{ij}^{best} \tag{3.8}$$

where $\Delta\tau_{ij}^{best} = \frac{1}{C^{best}}$. The ant which is allowed to add pheromone may be either the best-so-far or the iteration-best. In general, in MMAS implementations both the iteration-best and the best-so-far update rules are used, in an alternate way.

3.2.5 Ant Colony System

Ant Colony System (ACS) differs from AS in three main points. First, it exploits the search experience accumulated by the ants more strongly than AS does through the use of a more aggressive action choice rule. Second, pheromone evaporation and deposition take place only on the connections belonging to the best-so-far solution. Third, each time an ant uses a connection (i, j), it removes some pheromone from the connection to increase the exploration of alternative paths[87].

Solution Construction

In ACS, ants choose the components to built their solutions according to a pseudorandom proportional rule, presented in Equation 3.9:

$$p_{ij}^{k} = \begin{cases} argmax_{l \in N_i^k} \left\{ \tau_{il} \times [\eta_{il}]^{\beta} \right\} & \text{if } q \leq q_0 \\ \frac{\tau_{ij} \times [\eta_{ij}]^{\beta}}{\sum_{l \in N_i^k} [\tau_{il}]^{\alpha} \times [\eta_{il}]^{\beta}} & \text{otherwise} \end{cases} \tag{3.9}$$

where q is a random variable uniformly distributed in $[0, 1]$ and q_0 $(0 \leq q_0 \leq 1)$ is a parameter. As a result with probability q_0 the ant makes the best possible move according to the information learned so far while with probability $(1 - q_0)$ it performs a biased exploration of the connections.

Global Pheromone Trails Update

In ACS only one ant (the best-so-far ant) is allowed to add pheromone after each iteration. Thus, the update in ACS is implemented by the following equation:

$$\tau_{ij} \leftarrow (1-\rho)\tau_{ij} + \rho\Delta\tau_{ij}^{bs}, \quad \forall(i,j) \in T^{bs} \tag{3.10}$$

It's noteworthy to mention that in ACS the pheromone trail update, both evaporation and new pheromone deposit, only applies to the connections of T^{bs}, not to all the connections as in AS. This is important, because in this way the computational complexity of the pheromone update at each iteration is reduced from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$. Also note that, in Equation 3.10 the deposited pheromone is discounted by a factor ρ. This results in the new pheromone trail being a weighted average between the old pheromone value and the amount of pheromone deposited.

Local Pheromone Trails Update

In addition to the global pheromone trail updating rule, in ACS the ants use a local pheromone update rule that they apply immediately after having chosen a connection during the construction of their solution. The effect of that rule is that each time an ant uses a connection, its pheromone trail is reduced, so that the connection becomes less desirable for the following ants, encouraging the exploration of new solutions. This rule is presented in Equation 3.11:

$$\tau_{ij} \leftarrow (1-\xi)\tau_{ij} + \xi\tau_0 \tag{3.11}$$

where ξ, $0 < \xi < 1$, is the local pheromone evaporation rate and τ_0 is the initial value for the pheromone trails.

It is important to note that, while for the previously discussed AS variants it does not matter whether the ants construct the solutions in parallel or sequentially, in ACS this makes a difference, because of the local pheromone update rule. In most ACS implementations the choice has been to let all the ants move in parallel, although there is, at the moment, no experimental evidence in favor of one choice or the other.

3.2.6 Approximate Nondeterministic Tree Search (ANTS)

Approximate nondeterministic tree search (ANTS) is an ACO algorithm that exploits ideas from mathematical programming. In particular, ANTS computes lower bounds on the completion of a partial solution to define the heuristic information that is used by each ant during the solution construction. The name ANTS derives from the fact that the proposed algorithm can be interpreted as an approximate nondeterministic tree search since it can be extended in a straightforward way to a branch & bound procedure[5].

Apart from the use of lower bounds, ANTS also introduces two additional modifications with respect to AS: the use of a novel action choice rule and a modified pheromone trail update rule[87].

Use of Lower Bounds

In ANTS, lower bounds on the completion cost of a partial solution are used to compute heuristic information on the attractiveness of adding a connection (i, j) to the solution. This is achieved by tentatively adding the connection to the current partial solution and by estimating the cost of a complete solution containing this connection by means of a lower bound. This estimate is then used to compute the value η_{ij} that influences the probabilistic decisions taken by the ant during the solution construction. The lower the estimate the more attractive the addition of a specific connection.

The use of lower bounds to compute the heuristic information has the advantage in that otherwise feasible moves can be discarded if they lead to partial solutions whose estimated costs are larger than the best-so-far solution. A disadvantage is that the lower bound has to be computed at each single construction step of an ant and therefore a significant computational overhead might be incurred. To avoid this as much as possible, it is important that the lower bound is computed efficiently.

Solution Construction

The rule used by ANTS to compute the probabilities during the ants' solution construction has a different form than that used in most other ACO algorithms. In ANTS, an

[5]Branch and bound (BB or B&B) is an algorithm design paradigm for discrete and combinatorial optimization problems. A branch-and-bound algorithm consists of a systematic enumeration of candidate solutions by means of state space search. The set of candidate solutions is thought of as forming a rooted tree with the full set at the root. The algorithm explores branches of this tree, which represent subsets of the solution set. Before enumerating the candidate solutions of a branch, the branch is checked against upper and lower estimated bounds on the optimal solution, and is discarded if it cannot produce a better solution than the best one found so far by the algorithm[99].

ant k chooses components, j, that can be selected from a given current component, i, with probability:

$$p_{ij}^k = \frac{\zeta\tau_{ij} + (1-\zeta)\eta_{ij}}{\sum_{l\in N_i^k} \zeta\tau_{il} + (1-\zeta)\eta_{il}}, \quad \text{if } j \in N_i^k \tag{3.12}$$

where ζ, $0 \leq \zeta \leq 1$ is a parameter. An advantage of Equation 3.12 is that, when compared to Equation 3.1, only one parameter is used. Additionally, simpler operations that are faster to compute, like sums instead of multiplications for combining the pheromone trail and the heuristic information, are applied.

Pheromone Trails Update

Another particularity of ANTS is that it has no explicit pheromone evaporation. Pheromone updates are implemented as follows:

$$\tau_{ij} \leftarrow \tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^k \tag{3.13}$$

where $\Delta\tau_{ij}^k$ is given by:

$$\Delta\tau_{ij}^k = \begin{cases} \theta\left(1 - \frac{C^k - LB}{L_{avg} - LB}\right) & \text{if connection } (i,j) \text{ belongs to } T^k \\ 0 & \text{otherwise} \end{cases} \tag{3.14}$$

where θ is a parameter, LB is the value of a lower bound on the optimal solution value computed at the start of the algorithm and we have $LB \leq C^*$, where C^* is the cost of the optimal solution, and L_{avg} is the moving average of the last l solutions generated by the ants, that is, the average length of the l most recent solutions generated by the algorithm (with l being a parameter of the algorithm). If an ant's solution is worse than the current moving average, the pheromone trail of the connections used by the ant is decreased. Otherwise, if the ant's solution is better, the pheromone trail is increased. The additional effect of using Equation 3.14 is a dynamic scaling of the objective function differences which may be advantageous if in later stages of the search the absolute difference between the ant's solution qualities becomes smaller and, consequently, C^k moves closer to L_{avg}. (Note that once a solution with objective function value equal to LB is found, the algorithm can be stopped, because this means that an optimal solution is found.)

3.2.7 Hyper-Cube Framework for ACO

The hyper-cube framework for ACO was introduced to automatically rescale the pheromone values in order for them to lie always in the interval $[0, 1]$. This choice was inspired by the mathematical programming formulation of many combinatorial optimization problems, in which solutions can be represented by binary vectors. In such a formulation, the decision variables, which can assume the values $\{0, 1\}$, typically correspond to the solution components as they are used by the ants for solution construction. A solution to a problem then corresponds to one corner of the n-dimensional hyper-cube, where n is the number of decision variables. The set of all feasible solutions consists of all vectors $\overrightarrow{u} \in \mathcal{R}^n$ that are convex combinations of binary vectors $\overrightarrow{x} \in \mathcal{B}^n$[87].

Pheromone Trails Update

In the hyper-cube framework the pheromone trails are forced to stay in the interval $[0, 1]$. This is achieved by adapting the standard pheromone update rule of ACO algorithms. The modified rule is given by:

$$\tau_{ij} \leftarrow (1 - \rho)\tau_{ij} + \rho \sum_{k=1}^{m} \Delta\tau_{ij}^{k} \tag{3.15}$$

where,

$$\Delta\tau_{ij}^{k} = \begin{cases} \frac{1/C^k}{\sum_{h=1}^{m} 1/C^h} & \text{if connection } (i,j) \text{ is used by ant } k \\ 0 & \text{otherwise} \end{cases} \tag{3.16}$$

This pheromone trail update rule guarantees that the pheromone trails remain smaller than 1. The new pheromone vector can be interpreted as a shift of the old pheromone vector toward the vector given by the weighted average of the solutions used in the pheromone update.

3.2.8 Convergence of ACO Algorithms

The brief history of the ant colony optimization metaheuristic is mainly a history of experimental research. The first theoretical problem considered is the one concerning convergence: will the metaheuristic find the optimal solution if given enough resources? There are two types of convergence, convergence in value and convergence in solution.

Proving convergence in value intuitively means proving that the algorithm generates at least once the optimal solution. Proving convergence in solution can be interpreted as proving that the algorithm reaches a situation in which it generates over and over the same optimal solution. It has been proven that particular subsets of ACO algorithms converge asymptotically[87]. In particular, asymptotic convergence in value was proved for ACS and MMAS, two of the experimentally best-performing ACO algorithms[6]. Unfortunately, convergence proofs tell us that the bias introduced in the stochastic algorithm does not rule out the possibility of generating an optimal solution, but they do not say anything about the speed of convergence, that is, the computational time required to find an optimal solution.

3.2.9 Stagnation Detection

Artificial ants iteratively sample solutions through a loop that includes a solution construction, biased by the artificial pheromone trails and the heuristic information. The main mechanism at work in ACO algorithms that triggers the discovery of high quality solutions is the positive feedback given through the pheromone update by the ants. The higher the quality of the ant's solution, the higher the amount of pheromone the ant deposits on the connections (or components) of its solution. This in turn leads to the fact that these connections have a higher probability of being selected in the subsequent iterations of the algorithm. The emergence of connections with high pheromone values is further reinforced by the pheromone trail evaporation that avoids an unlimited accumulation of pheromones and quickly decreases the pheromone level on connections that only very rarely, or never, receive additional pheromone.

With good parameter settings, the long-term effect of the pheromone trails is to progressively reduce the size of the explored search space so that the search concentrates on a small number of promising connections. Yet, this behavior may become undesirable, if the concentration is so strong that it results in an early stagnation of the search[7]. In such an undesirable situation the system has ceased to explore new possibilities and no better solution is likely to be found anymore. Several measures may be used to describe the amount of exploration an ACO algorithm still performs and to detect stagnation situations, such as the standard deviation σ_L of the quality of the solutions the ants construct after every iteration, the λ-branching factor, or the average $\overline{\varepsilon} = \sum_{i=1}^{n} \varepsilon_i / n$ of the entropies ε_i of the selection probabilities at each node[87]. The stagnation behavior of

[6] Convergence in solution has also been proven for some uncommon members of ACO family, in particular GBAS and $\text{ACO}_{bs,\tau_{min}(\theta)}$.

[7] Search stagnation is defined as the situation in which all the ants follow the same path and construct the same solution.

the ACO algorithms is beyond the scope of this thesis so we will not make any further analysis.

3.2.10 Parallelization of ACO Algorithms

The very nature of ACO algorithms makes them inherently parallelizable. The solution construction phase can be easily parallelized by letting the ants move in parallel and build their solutions concurrently. There are many different exchange strategies[100] and also most of the parallel models that are used in other population based algorithms can be easily adapted to ACO. Most parallelization strategies can be classified into fine-grained and coarse-grained strategies. Characteristic of fine-grained parallelization is that very few individuals are assigned to single processors and that frequent information exchange among the processors takes place. In coarse-grained approaches, on the contrary, larger subpopulations or even full populations are assigned to single processors and information exchange is rather rare. Finally many researchers have investigated the type of information that should be exchanged among the colonies and how this information should be used to update the colonies' pheromone trail information and have also considered the case in which information among the colonies is exchanged at certain intervals[8]. Their main observation was that the best results were obtained by limiting the information exchange to the locally best solutions[87].

3.3 A Novel Placement Algorithm based on ACO

In this section we introduce our proposed algorithm for addressing the placement problem at 3-D reconfigurable architectures. Our approach relies on Ant Colony Optimization (ACO). It is a novel, swarm-intelligence based algorithm, that mimics the foraging behavior of ants in order to find a high quality placement in regard to legality constrains and optimization goals. The inherent parallelism of ACO algorithms, the flexibility they provide in regard of integrating different cost functions or FPGA architectures and the fact that they combine a positive feedback mechanism and stochasticity decision policy which account for rapid discovery of good solutions are just some of the strengths that make ACO algorithms a good fit for the problem of FPGA placement. The pseudo-code for our ACO-based placer is presented in Algorithm 2.

Several ACO algorithms have been proposed in the literature [87]. Our implementation incorporates concepts from the $MAX-MIN$ Ant System (MMAS) and the Ant

[8]For example, Bullnheimer et al.[101] proposed the partially asynchronous parallel implementation (PAPI). In PAPI, pheromone information was exchanged among the colonies every fixed number of iterations and a high speedup was experimentally observed.

Algorithm 2 Pseudo-code of our ACO-based placer.

```
1: set_aco_parameters();
2: init_heuristic_matrix();
3: init_pheromone_matrix();
4:
5: iteration = 1;
6: while (!termination_condition()) {
7:    for (n = 1; n ≤ n_ants; n++) {
8:       construct_solution(ant[n]);
9:       compute_placement_quality(ant[n]);
10:      compare_best_so_far_ant(ant[n]);
11:      local_pheromone_update(ant[n]);
12:   }
13:   global_pheromone_update();
14:   iteration++;
15: }
```

Colony System (ACS). The functionality of our introduced algorithm can be summarized as follows: In every iteration, each ant in the colony constructs a solution from scratch. To do so, every $ant[n]$ assigns block i to a physical location j on layer k using a pseudorandom proportional rule. As a result, all the blocks in the circuit netlist will be mapped to a specific location on the FPGA. The order in which the blocks are examined is random. These steps are repeated until a complete assignment is obtained. At that point we evaluate the quality of the solution and compare it to the best solution obtained so far ($best\_so\_far\_ant$), keeping the best of the two solutions. Every time an ant completes the construction phase, we perform a local pheromone update, so that the placement becomes less desirable for the following ants. At the end of each iteration, a global pheromone update rule is used for both evaporation and new pheromone deposition. The process continues until one of the termination conditions is met.

In the following subsections we give additional technical details about different parameters of the employed ACO algorithm.

3.3.1 Algorithm Initialization

To begin with, we initialize the various parameters used in ACO algorithms (i.e. ρ, α, β, ξ, q_0, τ_0). Proper initialization of these parameters is crucial as the performance of the algorithm is extremely affected by them. As is the common practice we evaluated these parameters through theory and preliminary experiments. Then, comes the initialization of the heuristic and pheromone matrices.

Heuristic Information

The heuristic information is used to guide the ants in the early stages of the algorithm towards promising regions of the search space. As we have stated before, even though convergence is guaranteed, the time to convergence is uncertain. That is why we need a good heuristic metric to speedup the process in the initial construction steps. In this implementation we introduce two types of heuristic information, a *static* and a *dynamic* one.

Static Heuristic Information

The concept we used to compute the static heuristic values for each netlist block and physical location is that, intuitively, the greater connection a block has in the circuit netlist, the more centric it should be implemented[102]. Hence, we define the heuristic information as shown in Equation 3.17:

$$\eta_{ijk} = cross(i) \times connect(j) \quad \forall k \tag{3.17}$$

where $cross(i)$ denotes the connective extent between block i and other blocks (Equation 3.18) and $connect(j)$ is defined as the total distance degree between $location(j)$ and other locations (Equation 3.19)

$$cross(i) = \sum_{j=1}^{\#blocks} graph(i,j) \tag{3.18}$$

$$connect(j) = \frac{1}{\sum_{i \in CLB} distance(i,j)} \tag{3.19}$$

where:

$$graph(i,j) = \begin{cases} 1 & \text{if } \exists \text{ connection (i, j)} \vee \text{(j, i)} \\ 0 & \text{otherwise} \end{cases} \tag{3.20}$$

The problem with the above static heuristic is that we observed that many of the benchmarks used in both academia and industry[9] have small fanin/fanout values. As

[9] A Survey of FPGA Benchmarks can be found here [103].

a result the values for $cross(i)$ were really small for most of the blocks. To counteract that effect we introduced a really robust dynamic heuristic function.

Dynamic Heuristic Information

The idea behind the dynamic heuristic information is that we want blocks of the same net (or path) to be close together in order to minimize the delay and total wire-length used. Setting aside hierarchical FPGA architectures with different lengths of wire, a good metric to evaluate a partial placement of a net is the sum Manhattan distance[10] of the blocks belonging to the net. With that in mind, each time an ant chooses a physical location for a netlist block, the dynamic heuristic information guides it towards physical locations closer to other already placed blocks of the same net. As you can image, this heuristic information can not be a priori defined. Instead it changes during the progressions of the solution construction.

For complexity reasons we do not compute heuristic values between every block of a net. Instead we modify the construction process to accommodate the on-the-fly computation of these values. To be more specific, instead of placing netlist blocks in random, we start by placing the blocks for the $\chi\%$ of the larger nets. We place the blocks of each of those nets sequentially, starting from the larger net, and we compute the dynamic heuristic information for every physical location of the FPGA as the Manhattan distance between the previously placed block of the net with the current block, as shown in Equation 3.21:

$$\eta_{ijk} = \frac{1}{manh\_dist(location(i), location(previous(i))} \tag{3.21}$$

where $location(i)$ is a possible physical location for block i (location j on layer k, for which we are currently computing its heuristic value) and $location(previous(i))$ is the physical location assigned to the previously placed block of the same net as i. This way the computational complexity drops to $\mathcal{O}(g \times \kappa)$ where g is the FPGA's grid size and κ the number of layers, or to $\mathcal{O}(g)$ if we perform the partition step of the 3-D CAD flow independently.

Pheromone Trails

After some iterations, the collective knowledge of the ants is incorporated into the pheromone information. Subsequent ants use the pheromone information as a guide

[10] The Manhattan distance is the sum of the absolute differences of their Cartesian coordinates.

toward promising regions of the search space. As a result, the initialization of the pheromone matrix plays a determining role in the performance of the algorithm. If the initial value (τ_0) for the pheromone trails is too low, then the search is quickly biased by the first solutions generated by the ants. On the other hand, too high initial values will require many iterations until pheromone evaporation reduces the pheromone values enough, so that pheromone added by ants can start to bias the search[87]. The formula we used to initialize the pheromone trails is $\tau_0 = 1/\rho C^*$, where C^* is the cost of the best placement we forecast (approximately 40% of the initial cost), since we estimate that in the long run, the upper pheromone trail limit on any component is bounded by $1/\rho C^*$. To compute the initial cost we use either a random placement, or a greedy placement based only on the heuristic information.

Like the heuristic information, a pheromone value is associated with each possible connection of the construction graph. That is, for each possible assignment of a netlist block i in a physical location j on a layer k, we have a value η_{ijk} and τ_{ijk}. Again, the size of these arrays (and consequently the time complexity of the initialization phase) can be considerably reduced by performing the partition step of the 3-D CAD flow independently and thus assigning beforehand the layers that each block will be placed.

3.3.2 Solution Construction

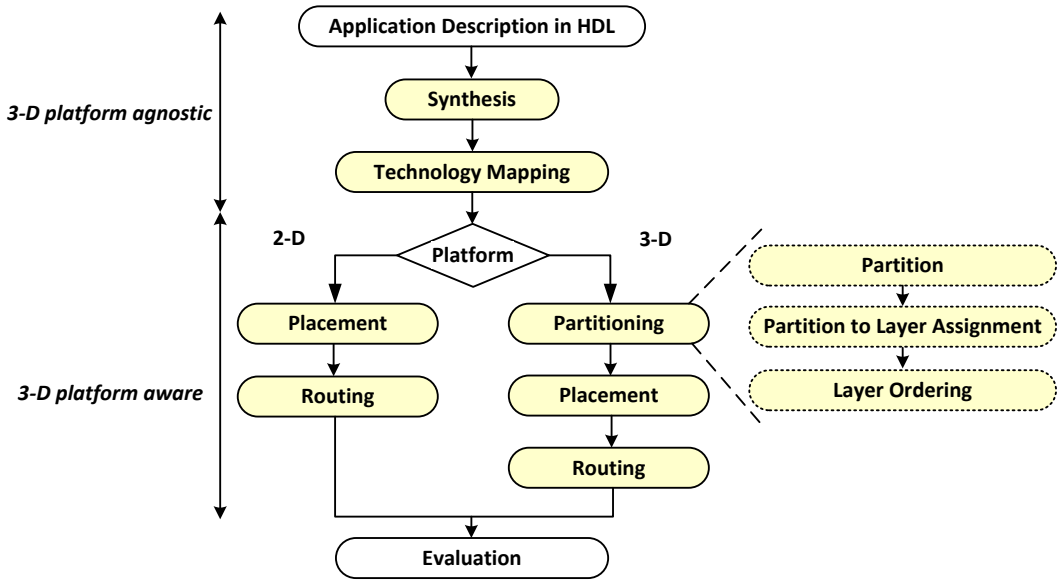

FIGURE 3.7: Toolflow for performing application mapping onto FPGA platforms.

In Figure 3.7 we remind the design process for application mapping onto 3-D Reconfigurable Architectures that we examined in Section 2.2. One of the key advantages of our implementation is the ability to incorporate in the solution construction phase both the

stages of partitioning and placement and as a result has the potential to arrive at superior placement solutions. In the solution construction phase we use the pseudorandom proportional rule from ACS. In particular, the probability with which ant m places the netlist block i to the physical spot j on the layer k is given by Equation 3.22:

$$p_{ijk}^{m} = \begin{cases} argmax_{(y,z)\in N^m}\left\{[\tau_{iyz}]^{\alpha} \times [\eta_{iyz}]^{\beta}\right\} & \text{if } q \leq q_0 \\ \frac{[\tau_{ijk}]^{\alpha}\times[\eta_{ijk}]^{\beta}}{\sum_{(y,z)\in N^m}[\tau_{iyz}]^{\alpha}\times[\eta_{iyz}]^{\beta}} & \text{otherwise} \end{cases} \tag{3.22}$$

where N^m is the set of available positions that ant m has in its disposition, q is a random variable uniformly distributed in $[0, 1]$ and $q_0, 0 \leq q_0 \leq 1$ is a parameter. So with probability q_0 the ant makes the best possible move as indicated by the learned pheromone and heuristic information (exploitation), while with probability $(1 - q_0)$ it performs a biased exploration [87]. Tuning the parameter q_0 allows modulation of the degree of exploration and the aggressiveness of the system, in other words how much the new placement will deviate from the $best\_so\_far$ placement.

The drawback of using the algorithm for both partitioning and placement is the huge search space that the ants have to face in order to find a good solution. However the inherent parallelism of our algorithm is the key that will allow for efficient exploration of ever growing search spaces as the multi-core computational power of conventional processors increases. To use as a pure placement algorithm, you just keep the layer k fixed (to the value given by the placer) for each netlist block. Then Equation 3.22 becomes:

$$p_{ij}^{m} = \begin{cases} argmax_{(y,z)\in N^m}\left\{[\tau_{iy}]^{\alpha} \times [\eta_{iy}]^{\beta}\right\} & \text{if } q \leq q_0 \\ \frac{[\tau_{ij}]^{\alpha}\times[\eta_{ij}]^{\beta}}{\sum_{(y,z)\in N^m}[\tau_{iy}]^{\alpha}\times[\eta_{iy}]^{\beta}} & \text{otherwise} \end{cases} \tag{3.23}$$

with the equivalent changes to the equations for pheromone trails and heuristic values.

3.3.3 Cost Function

After every block of the netlist has been successfully placed by an ant, it is time to evaluate the placement's quality. Here the optimization goals that we used are the minimization of the total wire-length and the minimization of the delay of the circuit. The cost function is presented in Equation 3.24:

$$Cost = \lambda \times timingCost + (1 - \lambda) \times wiringCost \tag{3.24}$$

The previously mentioned cost function is widely accepted and it is proved to be an effective way to evaluate the quality of the placement[85]. In this function, *wiringCost* attempts to minimize the total amount of interconnect required to successfully route the circuit by placing netlist blocks of the same net close together. It uses the following bounding box based function (Equation 3.25):

$$wiringCost = \sum_{i=1}^{\#nets} q(i) \times [bb_x(i) + bb_y(i) + bb_z(i)] \tag{3.25}$$

where bb_x, bb_y and bb_z are the spans across the x, y and z axes accordingly and $q(i)$ is a factor that compensates for the underestimation of the required wire by the above model.

The $TimingCost$ is used to reduce the critical path delay. To compute the $timingCost$, first we perform a timing analysis to the circuit in order to compute the delay of all of the paths[85]. Then, we compute the arrival time $T_{arrival}$, the required time $T_{required}$, as well as the slack and the criticality of each connection as follows:

$$T_{arrival}(i) = max_{\forall j \in fanin(i)} \{T_{arrival}(j) + delay(j,i)\} \tag{3.26}$$

$$T_{required}(i) = min_{\forall j \in fanout(i)} \{T_{required}(j) - delay(i,j)\} \tag{3.27}$$

$$Slack(i,j) = T_{required}(j) - T_{arrival}(i) - delay(i,j) \tag{3.28}$$

$$Criticality(i,j) = 1 - \frac{Slack(i,j)}{D_{max}} \tag{3.29}$$

where D_{max} is the critical path delay (maximum arrival time of all the sinks in the circuit). Then, we compute the timing cost of a connection (i, j) and the total $timingCost$ of the circuit (as the sum of all the timing costs) based on Equations 3.30 and 3.31, respectively.

$$timing\_cost(i,j) = delay(i,j) \times Criticality(i,j)^{Criticality\_Exp} \tag{3.30}$$

$$timingCost = \sum_{\forall i,j \subset circuit} timing\_cost(i,j) \tag{3.31}$$

For experimental purposes we also implemented two more cost functions, one computing the quadratic estimate for the k-point net using $k(k-1)/2$ 2-point nets[11], and one counting the number of hops.

3.3.4 Pheromone Update

Finally, we have the stage of the pheromone update. At this stage, the knowledge accumulated by the ants is incorporated into the pheromone trails to guide the following ants towards promising solutions. The aim of this stage is to increase the pheromone trails associated with good placement solutions while, using the evaporation mechanism, decrease the ones that produced poor solutions. In the proposed algorithm we adopt concepts from both the MMAS and the ACS. Specifically, just like in the ACS, we have two stages of pheromone update: a local one and a global one:

Global Pheromone Trail Update

The global pheromone update stage is performed at the end of every iteration. The role of this update is to decrease all the pheromone values through pheromone evaporation and increase the values associated with a good or promising solution, which is either the $best\_so\_far$ solution, or the $iteration\_best$ solution. Just like the MMAS, we alternate between the two. The relative frequency with which we choose to update the trails based on either the $best\_so\_far$ solution, or the $iteration\_best$ solution has an influence on how greedy the search is [87]. When pheromone updates are performed using the the $best\_so\_far$ solution, the search quickly focuses around that, whereas when we use the $iteration\_best$ solution, then the number of trails that receive pheromone is larger and the search is less directed. Experimental results [87] show that for large testcases the best performance is obtained by giving an increasingly stronger emphasis to the $best\_so\_far$ solution.

As for the pheromone update function per se, in our system we have implemented both the MMAS' and ACS' functions (as presented in Equations 3.2, 3.8 and 3.10). The default global pheromone update function used is given in Equation 3.32:

$$\tau'_{ijk} = \begin{cases} \tau_{min} & \text{if } \tau'_{ijk} < \tau_{min} \\ (1-\rho)\cdot\tau_{ijk} + \Delta\tau^{best}_{ijk} & \\ \tau_{max} & \text{if } \tau'_{ijk} > \tau_{max} \end{cases} \tag{3.32}$$

[11] More on Quadratic Placement in VLSI in [104]

$$\Delta\tau_{ijk}^{best} = \begin{cases} \frac{1}{Cost^{best}} & \text{if } ant^{best} \text{ maps } Block_i \text{ to } Spot_j,\ Layer_k \\ 0 & \text{otherwise} \end{cases} \tag{3.33}$$

where lower and upper trail limits τ_{min} and τ_{min} are defined as:

$$\tau_{max} = \tau_0 = 1/\rho C^* \tag{3.34}$$

$$\tau_{min} = \tau_{max}/c \tag{3.35}$$

where c is some constant. Lower and upper limits are imposed in order to avoid search stagnation (of course in the case of ACS' rule, the use of trail limits is obsolete since the pheromone update rule implicitly limits the pheromone trails in the range $[\tau_0, 1/C^{bs}]$[87]). Previous analysis on this topic[87] has shown that in order to avoid stagnation the lower pheromone trail limit play a more important role than upper trail limit.

Local Pheromone Trail Update

In addition to the global pheromone rule, we have adopted the local pheromone rule used in ACS. This rule is applied after an ant constructs a complete placement to discourage subsequent ants of the same iteration from constructing a similar placement, effectively increasing the exploration space. The function used for the local pheromone update is given by Equation 3.36:

$$\tau'_{ijk} = \begin{cases} \tau_{min} & \text{if } \tau'_{ijk} < \tau_{min} \\ (1-\xi)\cdot\tau_{ijk} & \forall(i,j,k) \in Placement^{best} \end{cases} \tag{3.36}$$

3.3.5 Heterogeneous Architectures

One key strength of the ACO algorithms is the ability to encode all the legality constrains and directly incorporate the targeted FPGA's architecture straight to the cost function. This can greatly facilitate the architecture-level exploration since it is fairly easy for the designers to address architecture-specific issues and expand to new architectures. As mentioned earlier in Section 3.1.2, it is possible to let the ants build low quality or even infeasible solutions. Those solutions simply "won't make the cut", and probably

no pheromone will be added to their trails. On the other hand, good solutions emerging from the use of the stochastic rule of ACO algorithms will be rewarded, increasing the probability of even more good solutions. Based on that, incorporating a new FPGA architecture is as simple as including the legality constrains into the cost function and letting the positive feedback rule of ACO algorithms do all the work.

In our implementation, the placer is aware of a possible heterogeneity in the interconnect fabric of an FPGA's architecture making it one of the few 3-D CAD tools supporting heterogeneous reconfigurable architectures. To incorporate this heterogeneity awareness, we simply tuned the *delay*() function used in Equations 3.26,3.27,3.28 and 3.30 to take into account the heterogeneity of the TSVs.

3.3.6 Parallel Implementation

ACO algorithms are inherently parallelizable both in the data and population domains[87]. Although the use of the local update rule of the ACS can lead to communication overhead. To overcome this, we implemented a coarse-grained approach with rather rare information exchange. In other words, for the parallel implementation we disregarded the local pheromone update rule and let all the ants move in parallel, exchanging information only at the end of every iteration. Due to the much larger colony, there were no considerable disadvantages from the omission of the local update rule, since the much larger number of ants can effectively search a much greater solution space. That led to a very simple implementation with considerable benefits in runtime. In fact it is so simple to parallelize an ACO algorithm, that by using the OpenMP API[12], you just need to insert a "parallel for" directive in line 7 of Algorithm 2.

3.3.7 Calibration

As is the standard practice, we calibrated our algorithm through a series of experiments, adjusting each time a different set of variables. We used a set of twenty commonly used MCNC circuits as benchmarks throughout the calibration process. See Appendix A for the default values for every parameter of our algorithm.

[12]OpenMP (Open Multi-Processing) is an API that supports multi-platform shared memory multiprocessing programming.

3.4 Summary

Ant colony optimization is a population-based metaheuristic that can be used to find approximate solutions to difficult optimization problems. In ACO, a set of agents called artificial ants search for good solutions to a given optimization problem. The artificial ants move through the graph, incrementally building solutions by employing a stochastic rule, biased by a pheromone model, that is, a set of parameters associated with graph components (either nodes or edges) whose values are modified at runtime by the ants.

Our proposed algorithm is based on ACO and uses the collective power of the artificial ants to find good quality placements onto 3-D reconfigurable architectures. It possesses many features that make it well suited to FPGA placement, such as:

- Inherent parallelism,
- direct enforcement of legality constrains into the cost function,
- support of heterogeneous architectures,
- open-source code.

Chapter 4

Experimental Results

This chapter quantifies the efficiency of the introduced algorithm as compared to the state-of-the-art relevant frameworks. The rest of the chapter is organized as follows: Section 4.1 presents our experimental setup. Section 4.2 presents the experimental results, and finally Section 4.3 summarizes the chapter.

4.1 Experimental Setup

The experimental setup is summarized as follows: The targeted 3-D FPGA consists of two to four layers, with identical logic and routing resources among these layers. Each of the layers follows the well-established island-style architecture discussed in Section 1.1.4, similar to the majority of commercial FPGA devices. The interlayer connections (TSVs) are implemented inside the 3-D switch boxes (SBs) and six TSVs per 3-D SB are considered.

For evaluation purposes, the introduced ACO-based placer was integrated as part of the open-source toolflow 3-D MEANDER[73]. Unfortunately, we cannot provide comparisons against the rest of the flows discussed in Table 2.1 since these tools are either not publicly available, or their implementation imposes constraints that cannot be addressed by existing 3-D technology (e.g. excessive amount of TSVs).

The measurements were taken using a set of the 20 biggest MCNC benchmarks. For the reference solution we employed the TPR tool[105]. Note that both the above tools perform netlist routing using the negotiated pathfinder algorithm. Consequently, we expect that performance improvement is based only to the different placement algorithms.

Four different colonies were utilized that concurrently searched for a good placement solution. In the end the best one was chosen.

4.2 Experimental Results

The TPR tool (as well as the other tools presented in Table 2.1) suffer from luck of flexibility. In addition they rely on straightforward extensions of existing 2-D tools which cannot fully exploit the benefits of 3-D technology. Therefore the aim of our project was to develop a state-of-the-art placer for modern 3-D FPGA architectures, able to be tuned according to requirements posed either by targeted application or by the employed 3-D platform.

The effectiveness of our proposed algorithm is depicted in Figure 4.1 which gives the critical path delay, normalized to the reference solution[1], for varying number of layers. As you can see, our algorithm achieves on average 10% reduction of the critical path delay. Additionally we have 32% reduction of the maximum net length and 29% reduction of the maximum segments used by a net (as depicted in Figures 4.2 and 4.3 respectively). Hence, besides the increase of the maximum operating frequency, we expect to have an improvement in power consumption.

The corresponding average values (average net length and average wire segments per net) are increased by 37% and 35% respectively (Figures 4.4 and 4.5). Still that does not constitute a drawback though, since the critical path delay is determined by the maximum values (the largest nets and paths are in the critical and near critical paths). Based on that, our algorithm takes a more aggressive approach in optimizing larger nets (and paths), which have the greatest impact on the critical path delay, allowing in that process the smaller ones to be placed in less than optimal way. For that reason, average net wire-length is increased as well (Figure 4.6).

At the same time Figure 4.7 shows a 30% improvement in the maximum number of bends. That leads to fewer transistors inside the switch boxes (SBs) and as a result lower cost.

Figure 4.8 shows the utilization of the interlayer connections (TSVs). As illustrated, our algorithm has on average a 10% lower TSV utilization, even though it achieves smaller critical path delay. That means it has the potential for even better placement solutions, by further exploiting the faster interlayer connections.

[1] All of the graphs presented in this section, except the one regarding the multi-CPU speedup, are normalized to the reference solution.

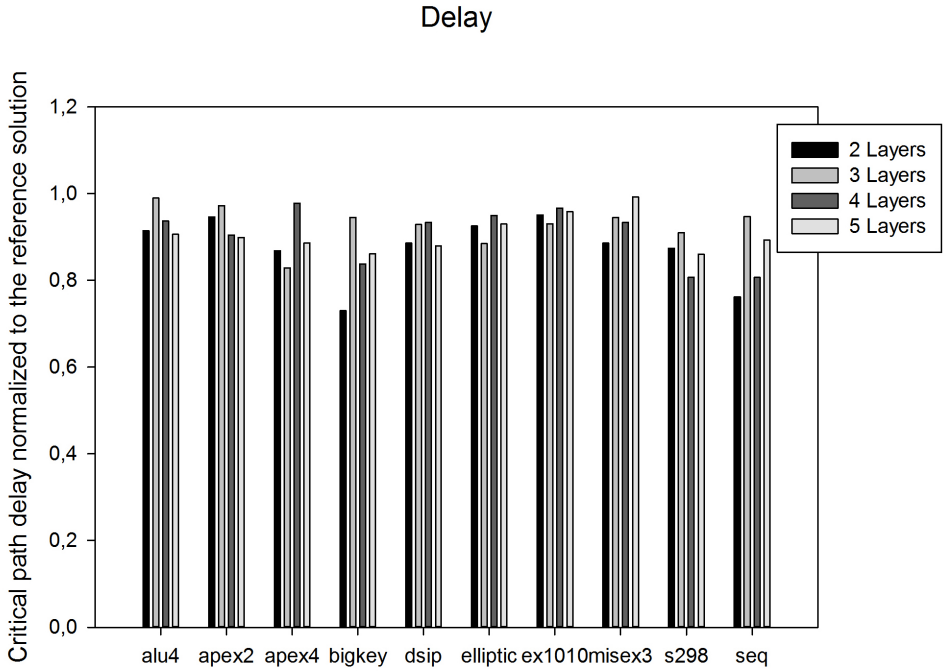

FIGURE 4.1: Critical Path Delay.

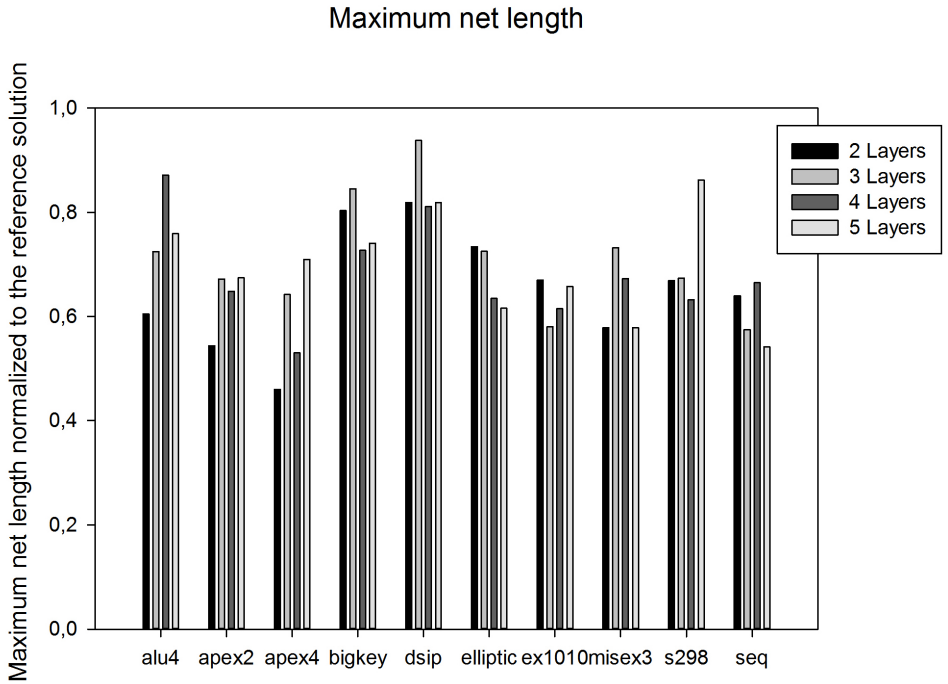

FIGURE 4.2: Maximum net length.

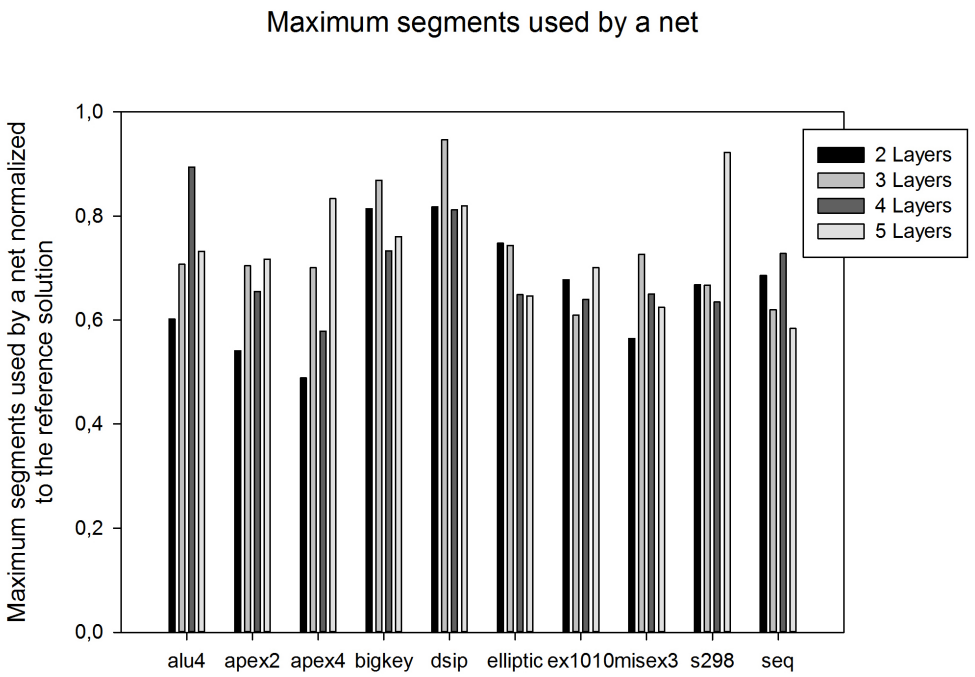

FIGURE 4.3: Maximum segments used by a net.

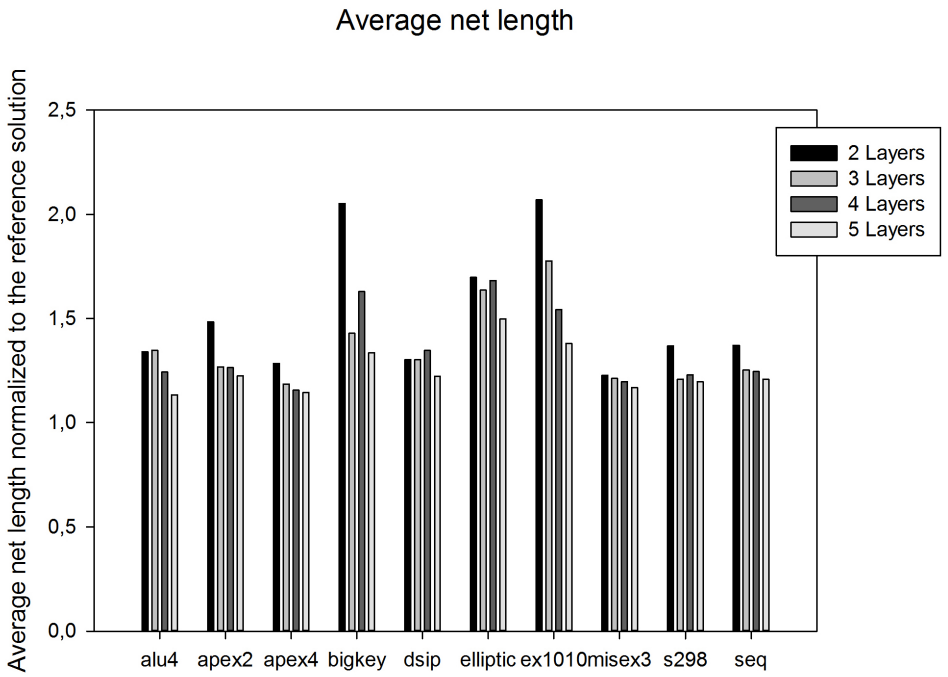

FIGURE 4.4: Average net length.

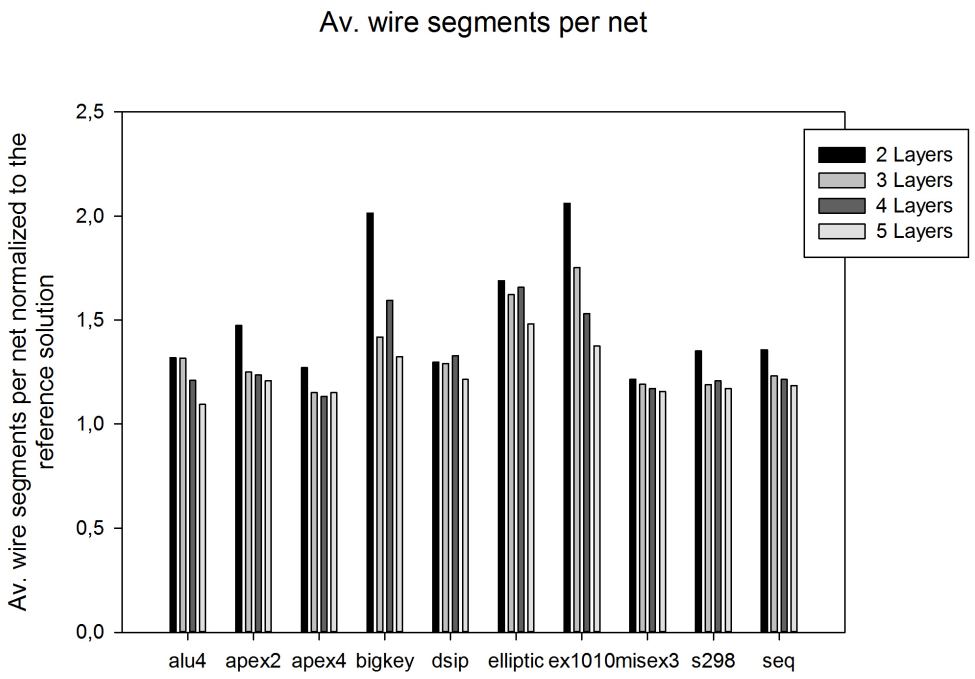

FIGURE 4.5: Average wire segments per net.

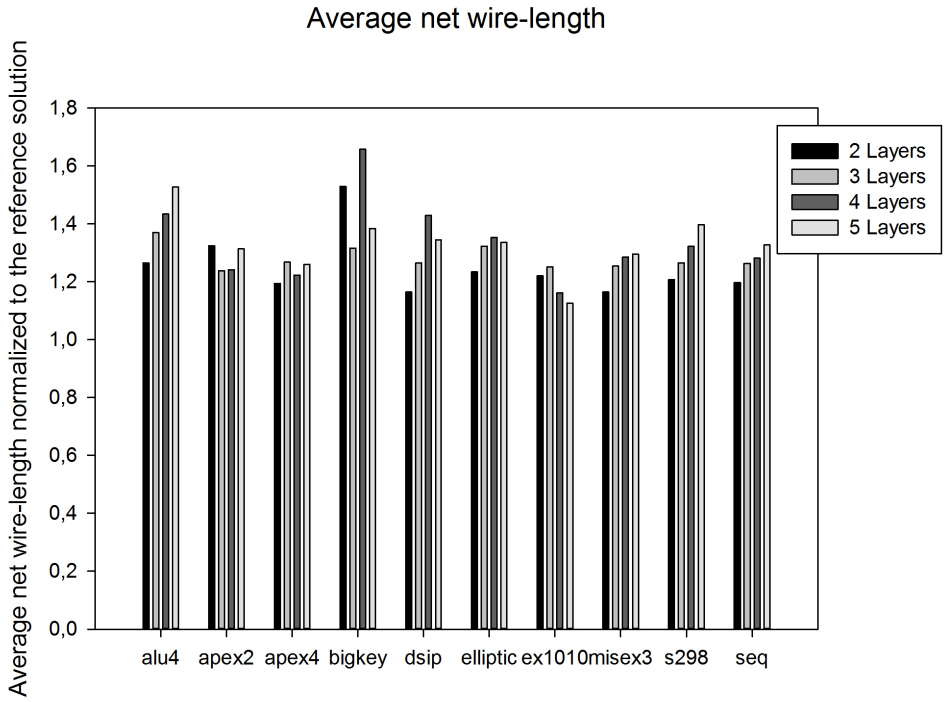

FIGURE 4.6: Average net wire-length.

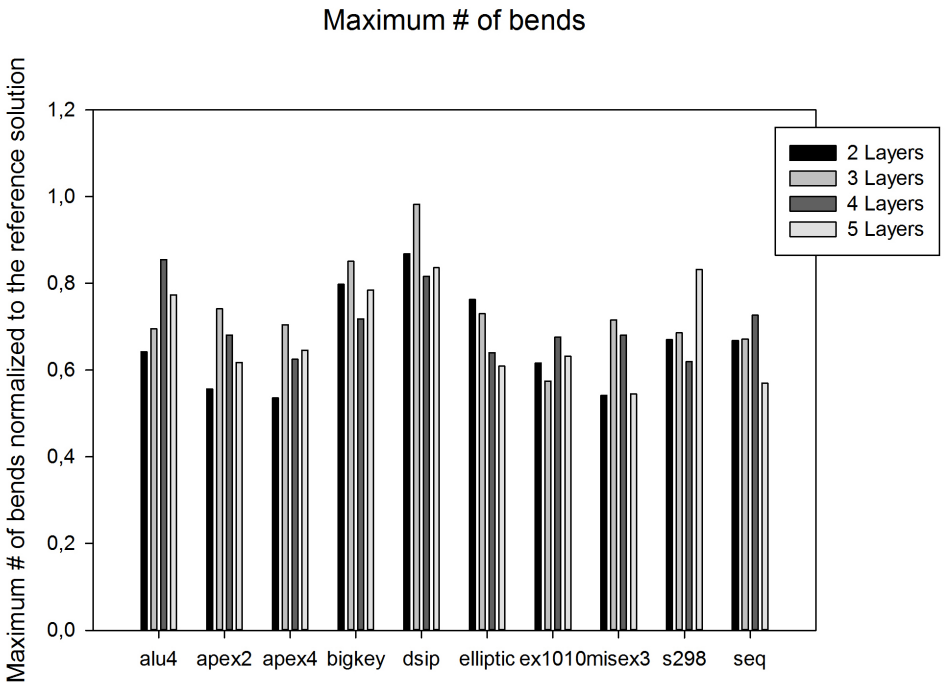

FIGURE 4.7: Maximum number of bends.

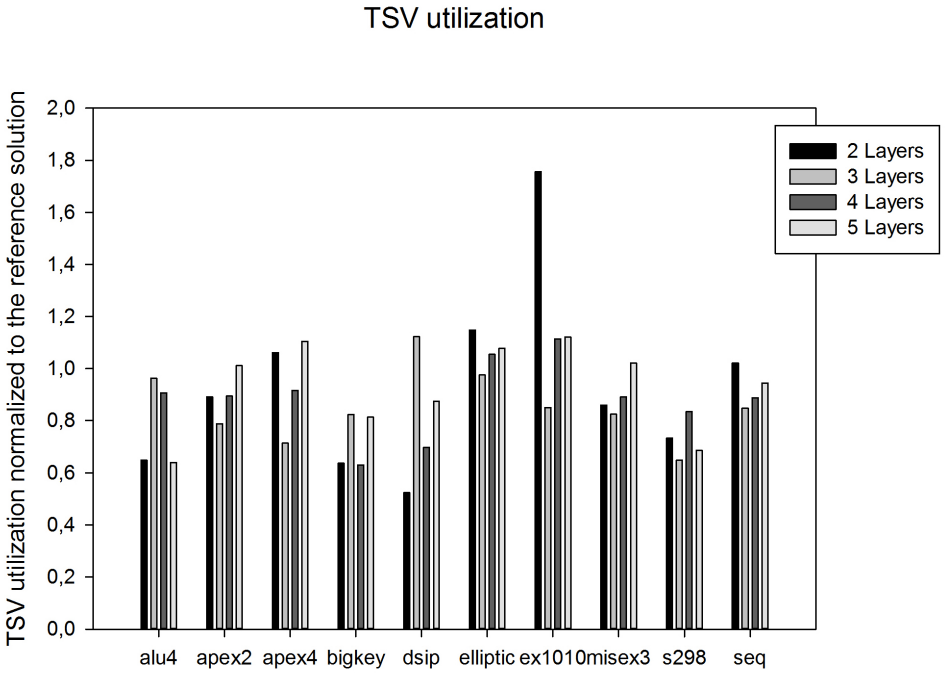

FIGURE 4.8: TSV utilization.

The trade-off to the aforementioned improvements is the fact that due to the increased average net length and average wire segments per net, we made routing even more complicated. As depicted in Figure 4.9, the route-run-time more than doubled. In fact, we have on average 151% increase in route-run-time. Nevertheless this does not constitute a problem. By taking advantage of the inherent parallelism of our algorithm and the existing multi-core platforms we can considerably reduce the execution run-time. Figure 4.10 depicts the speedup achieved in a 2-core and a 4-core processor with identical clock frequency. The results are normalized over the corresponding single-core and single-thread execution. As illustrated, our proposed algorithm achieves almost a 3× speedup on a 4-core CPU, with this speedup been very close to the theoretical threshold as estimated using the Amdahl's model[106].

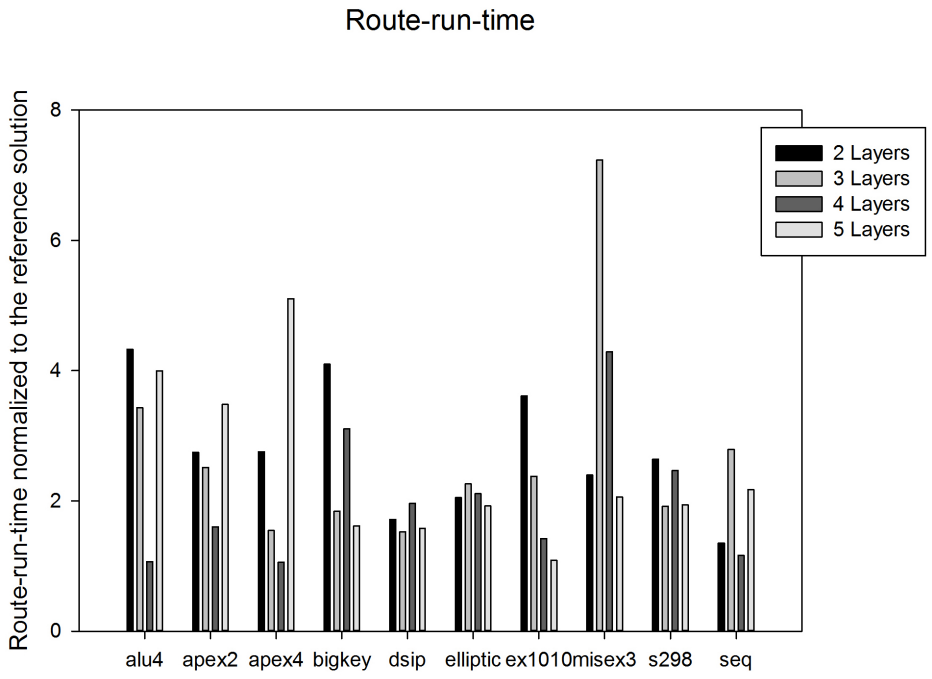

FIGURE 4.9: Route-run-time.

4.3 Summary

In this chapter we presented the experimental results for our proposed placer. As mentioned, our algorithm achieves on average 10% reduction on the critical path delay. Consequently this results to designs with increased maximum operating frequency and reduced power consumption. The trade-off for this improvement is an increase in route-run-time. Still this increased route-run-time can be eliminated by exploiting the inherent

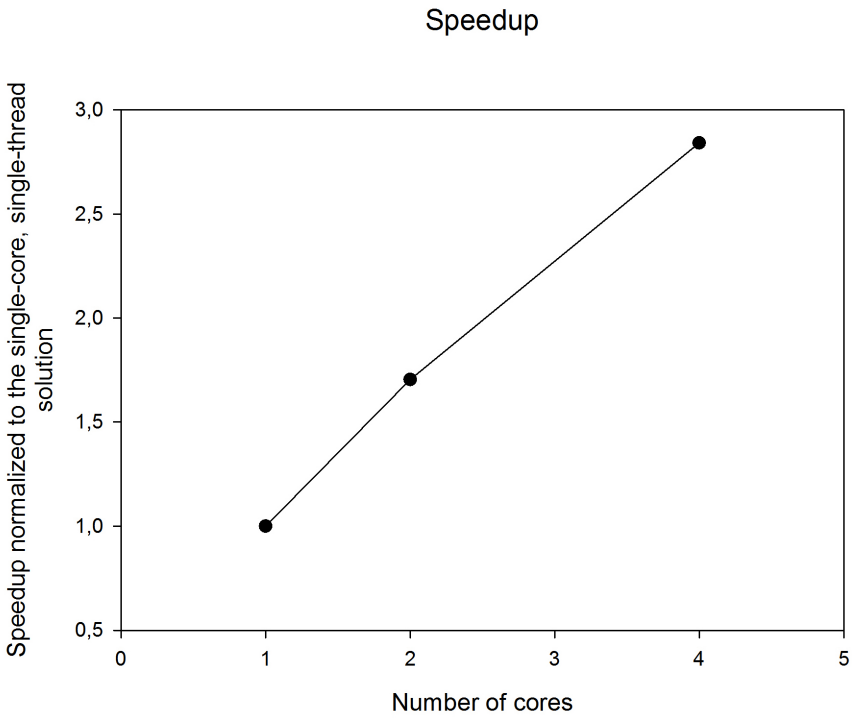

FIGURE 4.10: Multi-core speedup.

parallelism that characterizes ACO algorithms. As illustrated, our placer can achieve speedup very close to the theoretical threshold. This means that our proposed algorithm can take full advantage of todays multi-core architectures and has the ability to further decrease the execution run-time as multi-core CPUs scale according to today's market trends.

Chapter 5

Conclusion

5.1 Summary

Field-Programmable Gate Arrays (FPGAs) have received a lot of attention in the past few years. Their unique architecture has made them a popular implementation media for a wide range of applications. They combine the high performance of application-specific integrated circuits (ASICs) with the flexibility of microprocessors and offer many advantages such as quick time-to-market, no non-recurring engineering costs for fabrication, pre-tested silicon for use by the designers and re-programmability.

Over the last few years, in an effort to cope with the limitations of conventional 2-D circuit integration, designers have introduced a revolutionary new technique for three dimensional (3-D) chip stacking. This technique offers improved performance, reduced power consumption and lower cost compared to conventional two-dimensional integration methods. However, designing in such three dimensional platforms poses additional difficulties, making the already demanding process of mapping an application onto an FPGA even more challenging. Thus there is a compelling need for faster and more efficient Computer-Aided Design (CAD) tools to support application mapping in three dimension architectures.

Placement is considered one of the most arduous and time-consuming processes in physical implementation flows for reconfigurable architectures and at the same time it highly affects the quality of derived application implementation, as it has impact on the maximum operating frequency. This problem becomes even more harsh in 3-D architectures. To tackle this problem we introduced a novel placement algorithm, targeting three-dimensional reconfigurable architectures, based on Ant Colony Metaheuristics.

Ant colonies are distributed systems that, in spite of the simplicity of their individuals, present a highly structured social organization and as a result can accomplish complex tasks using the collective intelligence of the group. One of the most successful examples of ant based algorithms is known as Ant Colony Optimization (ACO). ACO is inspired by the foraging behavior of ants. The main idea is that the self-organizing principles which allow the highly coordinated behavior of real ants can be exploited to coordinate populations of artificial agents that collaborate to solve computational problems.

Our proposed algorithm incorporates concepts from both $MAX-MIN$ Ant System (MMAS) and Ant Colony System (ACS), the two best performing algorithms of the ACO family. It exhibits numerous advantages, such as inherent parallelism, direct enforcement of legality constrains into the cost function and support of heterogeneous architectures. For evaluation purposes, the introduced ACO-based placer was integrated as part of the open-source tool flow 3-D MEANDER. Experimental results validate the effectiveness of our algorithm since it achieves on average 10% reduction of the critical path delay. This results to designs with increased maximum operating frequency and reduced power consumption. Additionally our placer can achieve speedup in multi-core architectures very close to the theoretical one. This means that our proposed algorithm can take full advantage of todays multi-core CPUs, further decreasing the execution run-time.

5.2 Future Work

Many thing are still to be done for further development of our proposed tool. First and foremost, a more sophisticated timing-driven cost function must be integrated to the algorithm. The best performing tools for FPGA placement are timing-driven[40] and thus our tool should focus on that direction. Additionally our algorithm can benefit from a more fine-tuned parallel implementation since the inherent parallelism and its ability to exploit today's multi-core architectures is one of it's key strengths. Finally, based on the positive experimental results, we would encourage the adoption of ACO algorithms to tackle other parts of the FPGA CAD flow such as routing.

Appendix A

Manual

A.1 ACO-Placement3D

This is the manual for *ACO-Placement3D v1.0*, a novel algorithm for 3-D FPGA placement based on Ant Colony Optimization.

A.2 Copyright & Licensing

If you use ACO-Placement3D in your research, I would appreciate a citation in your publication(s). Please cite it as:

Panayiotis Danassis. ACO-Placement3D, Version 1.0.
Available from `https://github.com/panayiotisd/acoPlacement3D`, 2015.

The software is licensed under the MIT License:

A.3 Authors

This software was developed by *Panayiotis Danassis* with the valuable contribution of *Kostas Siozios*.

Panayiotis Danassis
url: http://panosd.eu/
e-mail: panos_dan@hotmail.com

Kostas Siozios
url: http://proteas.microlab.ntua.gr/ksiop/
e-mail: ksiop@microlab.ntua.gr

A.4 Manifest

Main control routines:

- acoPlacement.c
- acoPlacement.h

Implementation of ants' procedures:

- ants.c
- ants.h

Input / output routines:

- inOut.c
- inOut.h

Time measurement:

- timer.c

- timer.h

Auxiliary routines:

- utilities.c
- utilities.h

Other:

- Makefile
- README.md

A.5 Description

ACO-Placement3D is a novel placer, based on ant colony optimization (ACO), targeting 3-D FPGAs. It is a distributed algorithm, based on indirect communication (stigmergy) between artificial ants which work asynchronously to build a feasible placement. As so it is characterized by inherent parallelism and can greatly benefit from multi-core processors. It combines a positive feedback mechanism with stochastic decision policy and swarm intelligence.

The software was developed as part of my diploma thesis for the National Technical University of Athens (NTUA), school of Electrical and Computer Engineering. For more information please visit the following link (section: "publications"): `http://panosd.eu/`

A.6 Installation

Use the provided *Makefile* to compile the program under Linux. Executable *acoPlacement3D* is produced. There are four different compilation modes, which are presented in detail as follows:

A.6.1 Mode 1 (Default installation):

This is the default installation mode. Optimization flag $-O3$ is used. To compile in Default Mode just type:

make

A.6.2 Mode 2 (Print Mode):

In "Print Mode" the program displays, using a simple text based graphical representation, the best placement found so far, in order to visualize and give a better perspective of the solution found. It also informs you every time a better solution is found. Optimization flag $-O3$ is used. To compile in Print Mode type:

make print

A.6.3 Mode 3 (Debug Mode):

In "Debug Mode" the software prints a ton of debugging info. More specifically:

- The values of various key variables,
- The input netlist,
- The input/output pins,
- The global signals,
- The fanin and fanout table for every block,
- Exports in a file called "parse_netlist.out" the input netlist with the same format as the input file (with the exception of global signals),
- The hypernets (paths), if given,
- The nets,
- The positions of the TSVs (in heterogeneous architectures),
- States explicitly almost every function call and return in order to track the progress of the program.

No optimization flags are used. Also the software is compiled with the $-g$ flag in order to generate debugging information to be used by a debugger such as GDB. To compile in Debug Mode type:

make debug

A.6.4 Mode 4 (Parallel Mode):

In this mode the appropriate compiler flag is used ('-fopenmp') to "turn on" OpenMP, thus effectively parallelizing the application. Other than that, the "Parallel Mode" is similar to the "Print Mode". Optimization flag $-O3$ is used here as well. To compile in Parallel Mode type:

make parallel

A.6.5 Cleanup:

Use the following command to clean the directory from the executable and all the already compiled object files:

make clean

A.7 Usage

A.7.1 Options

To display the help text with the usage of the executable, type:

./acoPlacement3D

The following text will be displayed:

```
Usage: ./acoPlacement3D -g grid_size -c numberOfLayers -i input_netlist -h hypernets -n nets -p placement -e layers -v TSVs
[-r rho -a alpha -b beta -q q_0 -x xi -l lambda -f costFunction -w pheromoneUpd (mmas/acs) -z dynamic_heuristic (y/n)]
[-m maxIterations -t numberOfThreads -u initial_placement(random/heuristic) -s iteration_best_step -d tau_min_divisor]
```

Breaking down the above list, the command line options that the executable *acoPlacement3D* provides, are the following:

Mandatory Options:

- -g grid size.
- -i (input) netlist file name.
- -n (input) nets file name.
- -p (output) placement file name.

- -e (input) layers file (file that contains the layer that each block will be placed). Mandatory only if number of layers >1. The file must be in the same format as the placement file. (see Section A.7.4 for the format of the placement file)

Other Options:

- -m maximum number of iterations to perform (termination condition).
- -c number of layers (for 3D placement).
- -h (input) hypernets (paths) file name.
- -v (input) TSVs' locations (file containing the locations of the TSVs in a heterogeneous fabric. (see Section A.7.5 for the format of the file)
- -t number of threads (for parallel - OMP version).
- -u initial placement(*random*, *heuristic*). Chooses between a random initial placement or one based only on heuristic information.
- -f changes cost function (available choices: *wire_timing*, *quadratic_estimate*, *hops*).
- -l lambda, changes the relative importance between wire cost (bounding box) and timing cost in the *wire_timing* cost function.
  cost = lambda * timing_cost + (1 - lambda) * wire_cost;
- -w pheromoneUpd, changes between the two implemented pheromone update routines (*mmas*, *acs*). Type *mmas* to use the pheromone update rule of the $MAX - MIN$ Ant System (MMAS) or *acs* to use the rule of Ant Colony System (ACS).
- -z dynamic heuristic (y/n). Apply a dynamic heuristic criterion for the largest nets in order to improve the quality of the solution. A drawback is that runtime increases drastically.

ACO Related Options:

- -r rho, pheromone evaporation rate.
- -a alpha, pheromone trail influence.
- -b beta, heuristic information influence.

- -q q_0, Ant Colony System's (ACS) pseudorandom proportional rule's parameter.
- -x xi, Ant Colony System's (ACS) local pheromone update rule's evaporation rate.
- -s iteration_best_step, defines the relative frequency with which we choose to update the trails based on either the $best\_so\_far\_solution$, or the $iteration\_best\_solution$. Must be a positive non-zero value. E.g. if -s 1, then we only use $iteration\_best\_solution$, while if -s 999999 >max number of iterations, then we only use $best\_so\_far\_solution$. If -s 3, then we use $iteration\_best\_solution$ every 3 iterations.
- -d tau_min_divisor, defines the lower pheromone trail limit for the $MAX-MIN$ Ant System (MMAS), and as a result changes the stagnation behavior of the algorithm. tau_min = tau_max / tau_min_divisor.

Default Values:

As default, the pheromone update rules of MAX-MIN Ant System and the pseudorandom proportional rule of Ant Colony System (ACS) are used. The cost function that the algorithm tries to minimize is the total number of hops. The default values for all the above parameters are the following:

- -m 10
- -c 1
- -h (null)
- -t 1
- -u *random*
- -f *hops*
- -l 0
- -w *mmas*
- -z *y*
- -r 0.1
- -a 1
- -b 2

- -q 0.95
- -x 0
- -s 3
- -d 15

Pre-Compilation Options:

There are some parameters of the algorithm that are defined as constants. Those can be found at the .h files and are the following:

- n_ants number of ants in the colony (value: 256 ants)
- restart reinitialize pheromone matrix to avoid stagnation (value: INT_MAX iterations = disabled)
- printStep print results periodically (value: 5 iterations)
- exportPlacementStep export placement file (value: 5 iterations)

Note that options -c, -r, -a and -m can not take zero value.

A.7.2 Examples for running a benchmark:

./acoPlacement3D -i apex4.net -n apex4_net.echo -g 36 -p placement.p -m 10

or

./acoPlacement3D -i s38417.net -n s38417_net.echo -g 81 -p placement.p -t random -q 0.9 -b 2 -m 10 -z n

A.7.3 Output

Every run of the algorithm produces the following two files:
{your placement's file name}.p
{input netlist's name}.heur

The first one is the output placement file.

The second one is an auxiliary file with the values of the heuristic information. You can ignore this file. It's only used in subsequent runs of the algorithm to avoid the re-computation of the heuristic information and thus save some time.

A.7.4 Placement File Format

The first line of the placement file lists the netlist file and the architecture description file. The second line of the placement file gives the size of the logic block array (e.g. 20 x 20 logic blocks).

All the following lines have the format:

block_name x y z *subblock_number*

The block name is the name of this block, as given in the input netlist. x and y are the row and column in which the block is placed, respectively. z is the layer (vertical axis) of the block and it has to do only with the 3-D placement (it is not present in 2-D placement files). The subblock number is meaningful only for pads. Since we can place two pads in a row or column the subblock number specifies which of the possible pad locations (either location 0 or location 1) in row x and column y contains this pad. Note that the first pad occupied at some (x, y) location is always that with subblock number 0. For logic blocks (.clbs), the subblock number is always zero.

The placement files also include a fifth field as a comment. You can ignore this field.

A.7.5 TSVs File Format

This file is used in heterogeneous fabric architectures and contains the locations for all the TSVs (Through-Silicon Via). The file is just an array of $grid\_size \times grid\_size$ values of $\{0, 1\}$ (1 if a TSV exists in that location, 0 otherwise). An example of such file is presented below:

```
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 1 0 0 0
0 0 0 1 1 1 1 0 0 0
0 0 0 1 1 1 1 0 0 0
0 0 0 1 1 1 1 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
```

A.8 Known Bugs (& Future Work)

- Our netlist parser does not take into account comments. **If your netlist file includes comments, the program is going to crash!** To solve this, use under Linux the following command to get rid of the comments:

  ```
  sed -e 's/#.*$//' bench.net > bench_noComments.net
  ```

- For some reason OpenMP implementation doesn't work in some systems. Specifically the software works as it should if compiled under gcc version 4.7.2 (Debian 4.7.2-5) but if compiled under gcc version 4.8.2 20140120 (Red Hat 4.8.2-16) the program crashes.

- Parallel implementation works only with *BoundingBox* as cost function, due to dependencies in the usage of global timing matrices.

A.9 Contribute

- Source Code: `https://github.com/panayiotisd/acoPlacement3D`

A.10 Support

If you are having issues, contact us at: panos_dan@hotmail.com

Bibliography

[1] xilinx. What is programmable logic?, 2015. URL http://www.xilinx.com/company/about/programmable.html. [Online; accessed 15-May-2015].

[2] Wikipedia. Non-recurring engineering — wikipedia, the free encyclopedia, 2015. URL http://en.wikipedia.org/w/index.php?title=Non-recurring_engineering&oldid=659104398. [Online; accessed 15-May-2015].

[3] Wikipedia. Programmable logic device — wikipedia, the free encyclopedia, 2015. URL http://en.wikipedia.org/w/index.php?title=Programmable_logic_device&oldid=659545442. [Online; accessed 15-May-2015].

[4] Dave Vandenbout. Fpgas!? now what?, 2014. URL http://www.xess.com/static/media/appnotes/FpgasNowWhatBook.pdf. [Online; accessed 15-May-2015].

[5] Vaughn Betz, Jonathan Rose, and Alexander Marquardt, editors. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, Norwell, MA, USA, 1999. ISBN 0792384601.

[6] Stephen D. Brown, Robert J. Francis, Jonathan Rose, and Zvonko G. Vranesic. *Field-programmable Gate Arrays*. Kluwer Academic Publishers, Norwell, MA, USA, 1992. ISBN 0-7923-9248-5.

[7] S. Hauck and A. DeHon. *Reconfigurable Computing: The Theory and Practice of FPGA-based Computation*. Systems on Silicon Series. Morgan Kaufmann, 2008. ISBN 9780123705228. URL http://books.google.gr/books?id=vYgweLqkRzMC.

[8] Maya B. Gokhale and Paul S. Graham. *Reconfigurable Computing: Accelerating Computation with Field-Programmable Gate Arrays*. Springer Publishing Company, Incorporated, 1st edition, 2010. ISBN 1441938656, 9781441938657.

[9] xilinx. What is a fpga?, 2015. URL http://www.xilinx.com/fpga/. [Online; accessed 15-May-2015].

[10] M. Morris Mano and Michael D. Ciletti. *Digital Design (4th Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006. ISBN 0131989243.

[11] W.K. Chen. *The VLSI Handbook*. Electrical Engineering Handbook. Taylor & Francis, 2010. ISBN 9781420049671. URL https://books.google.gr/books?id=0r5LihlMogkC.

[12] xilinx. Cpld, 2015. URL http://www.xilinx.com/cpld/. [Online; accessed 15-May-2015].

[13] xilinx. Fpga vs. asic, 2015. URL http://www.xilinx.com/fpga/asic.htm. [Online; accessed 15-May-2015].

[14] Wikipedia. Central processing unit — wikipedia,, 2015. URL http://en.wikipedia.org/w/index.php?title=Central_processing_unit&oldid=661394727. [Online; accessed 18-May-2015].

[15] fpgacenter. Fpga or cpu?, 2015. URL http://fpgacenter.com/fpga/fpga_or_cpu.php. [Online; accessed 18-May-2015].

[16] J. Von Neumann. *First Draft of a Report on the EDVAC*. Moore School of Electrical Engineering, University of Pennsylvania, 1945. URL https://books.google.ch/books?id=t3zpygAACAAJ.

[17] Wikipedia. Von neumann architecture — wikipedia, the free encyclopedia, 2015. URL http://en.wikipedia.org/w/index.php?title=Von_Neumann_architecture&oldid=655865005. [Online; accessed 18-May-2015].

[18] Shuai Che, Jie Li, Jeremy W. Sheaffer, Kevin Skadron, and John Lach. Accelerating compute-intensive applications with gpus and fpgas. In *Proceedings of the 2008 Symposium on Application Specific Processors*, SASP '08, pages 101–107, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-1-4244-2333-0. doi: 10.1109/SASP.2008.4570793. URL http://dx.doi.org/10.1109/SASP.2008.4570793.

[19] F. Vahid. *Digital Design with RTL Design, Verilog and VHDL*. John Wiley & Sons, 2010. ISBN 9780470531082. URL https://books.google.gr/books?id=-YayRpmjc20C.

[20] Wikipedia. Hardware description language — wikipedia, the free encyclopedia, 2015. URL http://en.wikipedia.org/w/index.php?title=Hardware_description_language&oldid=662855044. [Online; accessed 19-May-2015].

[21] Wikipedia. Register-transfer level — wikipedia, the free encyclopedia, 2015. URL http://en.wikipedia.org/w/index.php?title=Register-transfer_level&oldid=662102220. [Online; accessed 19-May-2015].

[22] Jason Cong, Chang Wu, and Yuzheng Ding. Cut ranking and pruning: Enabling a general and efficient fpga mapping solution. In *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays*, FPGA '99, pages 29–35, New York, NY, USA, 1999. ACM. ISBN 1-58113-088-0. doi: 10.1145/296399.296425. URL http://doi.acm.org/10.1145/296399.296425.

[23] Robert Francis, Jonathan Rose, and Zvonko Vranesic. Chortle-crf: Fast technology mapping for lookup table-based fpgas. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, DAC '91, pages 227–233, New York, NY, USA, 1991. ACM. ISBN 0-89791-395-7. doi: 10.1145/127601.127670. URL http://doi.acm.org/10.1145/127601.127670.

[24] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM. ISBN 1-58113-297-2. doi: 10.1145/378239.379017. URL http://doi.acm.org/10.1145/378239.379017.

[25] Sean Safarpour, Andreas Veneris, Gregg Baeckler, and Richard Yuan. Efficient sat-based boolean matching for fpga technology mapping. In *Proceedings of the 43rd Annual Design Automation Conference*, DAC '06, pages 466–471, New York, NY, USA, 2006. ACM. ISBN 1-59593-381-6. doi: 10.1145/1146909.1147034. URL http://doi.acm.org/10.1145/1146909.1147034.

[26] D. Chen and J. Cong. Daomap: A depth-optimal area optimization mapping algorithm for fpga designs. In *Proceedings of the 2004 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '04, pages 752–759, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7803-8702-3. doi: 10.1109/ICCAD.2004.1382677. URL http://dx.doi.org/10.1109/ICCAD.2004.1382677.

[27] Jason Cong and Yean-Yow Hwang. Simultaneous depth and area minimization in lut-based fpga mapping. In *Proceedings of the 1995 ACM Third International Symposium on Field-programmable Gate Arrays*, FPGA '95, pages 68–74, New York, NY, USA, 1995. ACM. ISBN 0-89791-743-X. doi: 10.1145/201310.201322. URL http://doi.acm.org/10.1145/201310.201322.

[28] Julien Lamoureux and Steven J. E. Wilton. On the interaction between power-aware fpga cad algorithms. In *Proceedings of the 2003 IEEE/ACM International*

Conference on Computer-aided Design, ICCAD '03, pages 701–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 1-58113-762-1. doi: 10.1109/ICCAD.2003.106. URL http://dx.doi.org/10.1109/ICCAD.2003.106.

[29] Hao Li, Srinivas Katkoori, and Wai-Kei Mak. Power minimization algorithms for lut-based fpga technology mapping. *ACM Trans. Des. Autom. Electron. Syst.*, 9(1):33–51, January 2004. ISSN 1084-4309. doi: 10.1145/966137.966139. URL http://doi.acm.org/10.1145/966137.966139.

[30] Zhi-Hong Wang, En-Cheng Liu, Jianbang Lai, and Ting-Chi Wang. Power minization in lut-based fpga technology mapping. In *Proceedings of the 2001 Asia and South Pacific Design Automation Conference*, ASP-DAC '01, pages 635–640, New York, NY, USA, 2001. ACM. ISBN 0-7803-6634-4. doi: 10.1145/370155.370569. URL http://doi.acm.org/10.1145/370155.370569.

[31] Narasimha B. Bhat and Dwight D. Hill. Routable technologie mapping for lut fpgas. In *Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer &Amp; Processors*, ICCD '92, pages 95–98, Washington, DC, USA, 1992. IEEE Computer Society. ISBN 0-8186-3110-4. URL http://dl.acm.org/citation.cfm?id=645461.654583.

[32] Martine D. F. Schlag, Jackson Kong, and Pak K. Chan. Routability-driven technology mapping for lookup table-based fpga's. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 13(1):13–26, 1994. doi: 10.1109/43.273753. URL http://doi.ieeecomputersociety.org/10.1109/43.273753.

[33] Gang Chen and Jason Cong. Simultaneous logic decomposition with technology mapping in fpga designs. In *Proceedings of the 2001 ACM/SIGDA Ninth International Symposium on Field Programmable Gate Arrays*, FPGA '01, pages 48–55, New York, NY, USA, 2001. ACM. ISBN 1-58113-341-3. doi: 10.1145/360276.360298. URL http://doi.acm.org/10.1145/360276.360298.

[34] Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. Improvements to technology mapping for lut-based fpgas. In *Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays*, FPGA '06, pages 41–49, New York, NY, USA, 2006. ACM. ISBN 1-59593-292-5. doi: 10.1145/1117201.1117208. URL http://doi.acm.org/10.1145/1117201.1117208.

[35] Michael Hutton, Khosrow Adibsamii, and Andrew Leaver. Adaptive delay estimation for partitioning-driven pld placement. *IEEE Trans. Very Large Scale Integr. Syst.*, 11(1):60–63, February 2003. ISSN 1063-8210. doi: 10.1109/TVLSI.2002.808424. URL http://dx.doi.org/10.1109/TVLSI.2002.808424.

[36] Pongstorn Maidee, Cristinel Ababei, and Kia Bazargan. Fast timing-driven partitioning-based placement for island style fpgas. In *Proceedings of the 40th Annual Design Automation Conference*, DAC '03, pages 598–603, New York, NY, USA, 2003. ACM. ISBN 1-58113-688-9. doi: 10.1145/775832.775984. URL http://doi.acm.org/10.1145/775832.775984.

[37] Pak K. Chan and Martine D. F. Schlag. Parallel placement for field-programmable gate arrays. In *Proceedings of the 2003 ACM/SIGDA Eleventh International Symposium on Field Programmable Gate Arrays*, FPGA '03, pages 43–50, New York, NY, USA, 2003. ACM. ISBN 1-58113-651-X. doi: 10.1145/611817.611825. URL http://doi.acm.org/10.1145/611817.611825.

[38] Vaughn Betz and Jonathan Rose. Vpr: A new packing, placement and routing tool for fpga research, 1997.

[39] Jason Luu, Ian Kuon, Peter Jamieson, Ted Campbell, Andy Ye, Wei Mark Fang, and Jonathan Rose. Vpr 5.0: Fpga cad and architecture exploration tools with single-driver routing, heterogeneity and process scaling. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '09, pages 133–142, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-410-2. doi: 10.1145/1508128.1508150. URL http://doi.acm.org/10.1145/1508128.1508150.

[40] Alexander Marquardt, Vaughn Betz, and Jonathan Rose. Timing-driven placement for fpgas. In *Proceedings of the 2000 ACM/SIGDA Eighth International Symposium on Field Programmable Gate Arrays*, FPGA '00, pages 203–213, New York, NY, USA, 2000. ACM. ISBN 1-58113-193-3. doi: 10.1145/329166.329208. URL http://doi.acm.org/10.1145/329166.329208.

[41] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *SCIENCE*, 220(4598):671–680, 1983.

[42] Yu-Liang Wu and Douglas Chang. On the np-completeness of regular 2-d fpga routing architectures and a novel solution. In *Proceedings of the 1994 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '94, pages 362–366, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press. ISBN 0-89791-690-5. URL http://dl.acm.org/citation.cfm?id=191326.191492.

[43] Ian Kuon, Russell Tessier, and Jonathan Rose. Fpga architecture: Survey and challenges. *Found. Trends Electron. Des. Autom.*, 2(2):135–253, February 2008. ISSN 1551-3939. doi: 10.1561/1000000005. URL http://dx.doi.org/10.1561/1000000005.

[44] Nir Magen, Avinoam Kolodny, Uri C. Weiser, and Nachum Shamir. Interconnect-power dissipation in a microprocessor. In Louis Scheffer and Igor L. Markov, editors, *SLIP*, pages 7–13. ACM, 2004. ISBN 1-58113-818-0.

[45] Vasilis F. Pavlidis and Eby G. Friedman. *Three-dimensional Integrated Circuit Design*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2009. ISBN 9780080921860, 9780123743435.

[46] A. Papanikolaou, D. Soudris, and R. Radojcic. *Three Dimensional System Integration: IC Stacking Process and Design*. SpringerLink : Bücher. Springer, 2010. ISBN 9781441909626.

[47] Jan Rabaey. *Low Power Design Essentials*. Springer Publishing Company, Incorporated, 1st edition, 2009. ISBN 0387717129, 9780387717128.

[48] Aman Gayasen, Narayanan Vijaykrishnan, Mahmut T. Kandemir, and Arifur Rahman. Designing a 3-d fpga: Switch box architecture and thermal issues. *IEEE Trans. VLSI Syst.*, 16(7):882–893, 2008. URL `http://dblp.uni-trier.de/db/journals/tvlsi/tvlsi16.html#GayasenVKR08`.

[49] Mingjie Lin, Abbas El Gamal, Yi-Chang Lu, and S. Simon Wong. Performance benefits of monolithically stacked 3-d fpga. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 26(2):216–229, 2007. URL `http://dblp.uni-trier.de/db/journals/tcad/tcad26.html#LinGLW07`.

[50] Guy Lemieux and David A. Lewis. *Design of interconnection networks for programmable logic.* Kluwer, 2004. ISBN 978-1-4020-7700-5.

[51] Kostas Siozios, Vasilis F. Pavlidis, and Dimitrios Soudris. A novel framework for exploring 3-d fpgas with heterogeneous interconnect fabric. *ACM Trans. Reconfigurable Technol. Syst.*, 5(1):4:1–4:23, March 2012. ISSN 1936-7406. doi: 10.1145/2133352.2133356. URL `http://doi.acm.org/10.1145/2133352.2133356`.

[52] David Fang, Song Peng, Chris LaFrieda, and Rajit Manohar. A three-tier asynchronous fpga. In *Proceedings of the International VLSI/ULSI Multilevel Interconnection Conference*. Citeseer, 2006.

[53] Vaughn Betz, Jonathan Rose, and Alexander Marquardt, editors. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, Norwell, MA, USA, 1999. ISBN 0792384601.

[54] Cristinel Ababei, Yan Feng, Brent Goplen, Hushrav Mogal, Tianpei Zhang, Kia Bazargan, and Sachin S. Sapatnekar. Placement and routing in 3d integrated

circuits. *IEEE Design & Test of Computers*, 22(6):520–531, 2005. URL `http://dblp.uni-trier.de/db/journals/dt/dt22.html#AbabeiFGMZBS05`.

[55] Shamik Das, Andy Fan, Kuan-Neng Chen, Chuan Seng Tan, Nisha Checka, and Rafael Reif. Technology, performance, and computer-aided design of three-dimensional integrated circuits. In Charles J. Alpert and Patrick Groeneveld, editors, *ISPD*, pages 108–115. ACM, 2004. ISBN 1-58113-817-2. URL `http://dblp.uni-trier.de/db/conf/ispd/ispd2004.html#DasFCTCR04`.

[56] Chen Dong, Deming Chen, S. Haruehanroengra, and Wei Wang 0003. 3-d nfpga: A reconfigurable architecture for 3-d cmos/nanomaterial hybrid digital circuits. *IEEE Trans. on Circuits and Systems*, 54-I(11):2489–2501, 2007. URL `http://dblp.uni-trier.de/db/journals/tcas/tcasI54.html#DongCH007`.

[57] Roto Le, Sherief Reda, and R. Iris Bahar. High-performance, cost-effective heterogeneous 3d fpga architectures. In Paul Chow and Peter Y. K. Cheung, editors, *FPGA*, page 286. ACM, 2009. ISBN 978-1-60558-410-2. URL `http://dblp.uni-trier.de/db/conf/fpga/fpga2009.html#LeRB09`.

[58] S. Gupta, M. Hilbert, S. Hong, and R. Patti. Techniques for producing 3d ics with high-density interconnect. In *Proceedings of the 21st Intl. VLSI Multilevel Interconnection Conf.*, 2004.

[59] A.W. Topol, D.C.La Tulipe, L. Shi, D.J. Frank, K. Bernstein, S.E. Steen, A. Kumar, G.U. Singco, A.M. Young, K.W. Guarini, and M. Ieong. Three-dimensional integrated circuits. *IBM Journal of Research and Development*, 50(4.5):491–506, July 2006. ISSN 0018-8646. doi: 10.1147/rd.504.0491.

[60] Tezzaron. 3-d fpga from tezzaron `http://www.tezzaron.com/about/PhotoAlbum/Products/3DFPGA.html`.

[61] Xilinx. Stacked silicon interconnect technology delivers breakthrough FPGA capacity, bandwidth, and power efficiency.

[62] Andre Hahn Pereira and Vaughn Betz. Cad and routing architecture for interposer-based multi-fpga systems. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, FPGA '14, pages 75–84, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2671-1. doi: 10.1145/2554688.2554776. URL `http://doi.acm.org/10.1145/2554688.2554776`.

[63] B.K. Britton, Y.T. Oh, W. Oswald, H.T. Nguyen, S. Singh, G. Lee, W.-B. Leung, C. Spivak, J. Steward, and C-T Chen. Second generation orca architecture utilizing 0.5μm process enhances the speed and usable gate capacity of fpgas. In *ASIC*

Conference and Exhibit, 1994. Proceedings., Seventh Annual IEEE International, pages 474–478, Sep 1994. doi: 10.1109/ASIC.1994.404516.

[64] Harry Sidiropoulos, Kostas Siozios, Peter Figuli, Dimitrios Soudris, Michael Hübner, and Jürgen Becker. Jitpr: A framework for supporting fast application's implementation onto fpgas. *ACM Trans. Reconfigurable Technol. Syst.*, 6(2):7:1–7:12, August 2013. ISSN 1936-7406. doi: 10.1145/2492185. URL http://doi.acm.org/10.1145/2492185.

[65] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multilevel hypergraph partitioning: Application in vlsi domain. In *Proceedings of the 34th Annual Design Automation Conference*, DAC '97, pages 526–529, New York, NY, USA, 1997. ACM. ISBN 0-89791-920-3. doi: 10.1145/266021.266273. URL http://doi.acm.org/10.1145/266021.266273.

[66] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th Design Automation Conference*, DAC '82, pages 175–181, Piscataway, NJ, USA, 1982. IEEE Press. ISBN 0-89791-020-6. URL http://dl.acm.org/citation.cfm?id=800263.809204.

[67] B. W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell system technical journal*, 49(2):291—308, 1970.

[68] Kostas Siozios and Dimitrios Soudris. A tabu-based partitioning and layer assignment algorithm for 3-d fpgas. *Embedded Systems Letters*, 3(3):97–100, 2011. doi: 10.1109/LES.2011.2161571. URL http://doi.ieeecomputersociety.org/10.1109/LES.2011.2161571.

[69] Dae Hyun Kim and Sung Kyu Lim. Through-silicon-via-aware delay and power prediction model for buffered interconnects in 3d ics. In *Proceedings of the 12th ACM/IEEE International Workshop on System Level Interconnect Prediction*, SLIP '10, pages 25–32, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0037-7. doi: 10.1145/1811100.1811108. URL http://doi.acm.org/10.1145/1811100.1811108.

[70] Melvin A. Breuer. A class of min-cut placement algorithms. In *Proceedings of the 14th Design Automation Conference, DAC '77, New Orleans, Louisiana, USA, June 20-22, 1977*, pages 284–290, 1977. URL http://dl.acm.org/citation.cfm?id=809144.

[71] Hans Eisenmann and F.M. Johannes. Generic global placement and floorplanning. In *Design Automation Conference, 1998. Proceedings*, pages 269–274, June 1998.

[72] Mingjie Lin, A. El Gamal, Yi-Chang Lu, and Simon Wong. Performance benefits of monolithically stacked 3-d fpga. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(2):216–229, Feb 2007. ISSN 0278-0070. doi: 10.1109/TCAD.2006.887920.

[73] Kostas Siozios, Vasilis F. Pavlidis, and Dimitrios Soudris. A novel framework for exploring 3-d fpgas with heterogeneous interconnect fabric. *ACM Trans. Reconfigurable Technol. Syst.*, 5(1):4:1–4:23, March 2012. ISSN 1936-7406. doi: 10.1145/2133352.2133356.

[74] MEVA-3D, 2014. URL `http://cadlab.cs.ucla.edu/three_d/3dic.html`.

[75] C. Ababei, Y. Feng, B. Goplen, Hushrav Mogal, Tianpei Zhang, K. Bazargan, and S. Sapatnekar. Placement and routing in 3d integrated circuits. *Design Test of Computers, IEEE*, 22(6):520–531, Nov 2005. ISSN 0740-7475. doi: 10.1109/MDT.2005.150.

[76] C. Ababei, H. Mogal, and K. Bazargan. Three-dimensional place and route for fpgas. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Trans. on*, 25(6):1132–1140, June 2006. ISSN 0278-0070. doi: 10.1109/TCAD.2005.855945.

[77] Andre Hahn Pereira and Vaughn Betz. Cad and routing architecture for interposer-based multi-fpga systems. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, FPGA '14, pages 75–84, NY, USA, 2014. ACM. ISBN 978-1-4503-2671-1. doi: 10.1145/2554688.2554776.

[78] Vinod Pangracious, Emna Amouri, Zied Marakchi, and Habib Mehrez. Architecture level optimization of 3-dimensional tree-based FPGA. *Microelectronics Journal*, 45(4):355–366, 2014. ISSN 0026-2692. doi: 10.1016/j.mejo.2013.12.011.

[79] Kostas Siozios and Dimitrios Soudris. A tabu-based partitioning and layer assignment algorithm for 3-d fpgas. *IEEE Embed. Syst. Lett.*, 3(3):97–100, September 2011. ISSN 1943-0663. doi: 10.1109/LES.2011.2161571. URL `http://dx.doi.org/10.1109/LES.2011.2161571`.

[80] Kostas Siozios and Dimitrios Soudris. A power-aware placement and routing algorithm targeting 3d fpgas. *J. Low Power Electronics*, 4(3):275–289, 2008. doi: 10.1166/jolpe.2008.184. URL `http://dx.doi.org/10.1166/jolpe.2008.184`.

[81] N. Selvakkumaran and G. Karypis. Multiobjective hypergraph-partitioning algorithms for cut and maximum subdomain-degree minimization. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 25(3):504–517, November 2006. ISSN 0278-0070. doi: 10.1109/TCAD.2005.854637.

[82] Harry Sidiropoulos, Kostas Siozios, Peter Figuli, Dimitrios Soudris, Michael Hübner, and Jürgen Becker. JITPR: A framework for supporting fast application's implementation onto fpgas. *TRETS*, 6(2):7, 2013. doi: 10.1145/2492185. URL `http://doi.acm.org/10.1145/2492185`.

[83] Russell Tessier. Fast placement approaches for FPGAs. *ACM Trans. Design Autom. Electr. Syst.*, 7(2):284–305, 2002.

[84] Subhash Gupta, Mark Hilbert, Sangki Hong, and Robert Patti. Techniques for producing 3d ics with high-density interconnect.

[85] Vaughn Betz, Jonathan Rose, and Alexander Marquardt, editors. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, Norwell, MA, USA, 1999. ISBN 0792384601.

[86] Xilinx stacked silicon interconnect technology delivers breakthrough fpga capacity, bandwidth, and power efficiency, Oct. 2010.

[87] Marco Dorigo and Thomas Stützle. *Ant Colony Optimization*. Bradford Company, Scituate, MA, USA, 2004. ISBN 0262042193.

[88] G. Beni and J. Wang. Swarm intelligence in cellular robotic systems. In *NATO Advanced Workshop on Robotics and Biological Systems*, June 1989.

[89] Gerardo Beni. From swarm intelligence to swarm robotics. In *Proceedings of the 2004 International Conference on Swarm Robotics*, SAB'04, pages 1–9, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3-540-24296-1, 978-3-540-24296-3. doi: 10.1007/978-3-540-30552-1_1. URL `http://dx.doi.org/10.1007/978-3-540-30552-1_1`.

[90] Wikipedia. Swarm intelligence — wikipedia, the free encyclopedia, 2015. URL `http://en.wikipedia.org/w/index.php?title=Swarm_intelligence&oldid=662220353`. [Online; accessed 21-May-2015].

[91] J. Brownlee. *Clever Algorithms: Nature-inspired Programming Recipes*. LULU Press, 2011. ISBN 9781446785065. URL `https://books.google.co.uk/books?id=SESWXQphCUkC`.

[92] NIST. Np-complete, 2015. URL `http://xlinux.nist.gov/dads/HTML/npcomplete.html`. [Online; accessed 15-May-2015].

[93] Marco Dorigo and Gianni Di Caro. New ideas in optimization. chapter The Ant Colony Optimization Meta-heuristic, pages 11–32. McGraw-Hill Ltd., UK, Maidenhead, UK, England, 1999. ISBN 0-07-709506-5. URL `http://dl.acm.org/citation.cfm?id=329055.329062`.

[94] Marco Dorigo, Mauro Birattari, and Thomas Stützle. Ant colony optimization – artificial ants as a computational intelligence technique. *IEEE COMPUT. INTELL. MAG*, 1:28–39, 2006.

[95] Vinay Chopra and Amardeep Singh. Ant colony based approach for solving fpga routing.

[96] M. Dorigo and L. M. Gambardella. Ant colony system: A cooperative learning approach to the traveling salesman problem. *Trans. Evol. Comp*, 1(1):53–66, April 1997. ISSN 1089-778X. doi: 10.1109/4235.585892. URL `http://dx.doi.org/10.1109/4235.585892`.

[97] Setareh Shafaghi1 Fardad Farokhi and Reza Sabbaghi-Nadooshan. New ant colony algorithm method based on mutation for fpga placement problem. 2013.

[98] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, USA, 1998. ISBN 0262631857.

[99] Wikipedia. Branch and bound — wikipedia, the free encyclopedia, 2015. URL `http://en.wikipedia.org/w/index.php?title=Branch_and_bound&oldid=660739988`. [Online; accessed 22-May-2015].

[100] Issmail Ellabib, Paul Calamai, and Otman Basir. Exchange strategies for multiple ant colony system. *Inf. Sci.*, 177(5):1248–1264, March 2007. ISSN 0020-0255. doi: 10.1016/j.ins.2006.09.016. URL `http://dx.doi.org/10.1016/j.ins.2006.09.016`.

[101] Bullnheimer Bernd, Kotsis Gabriele, and Strau Christine. Parallelization strategies for the ant system, 1997.

[102] Wenyao Xu, Kejun Xu, and Xinmin Xu. A novel placement algorithm for symmetrical fpga. In *ASIC, 2007. ASICON '07. 7th International Conference on*, pages 1281–1284, Oct 2007. doi: 10.1109/ICASIC.2007.4415870.

[103] Raphael Njuguna. A survey of fpga benchmarks, 2015. URL `http://www.cse.wustl.edu/~jain/cse567-08/ftp/fpga/`. [Online; accessed 15-May-2015].

[104] Rob A. Rutenbar. Vlsi cad: Logic to layout, page 54, 2015. URL `https://github.com/blackmatt37/coursera/raw/master/VLSI%20CAD/Week%205/9-vlsicad-placer.pdf`. [Online; accessed 15-May-2015].

[105] Cristinel Ababei. Tpr: Three-d place and route for fpgas. In Jürgen Becker, Marco Platzner, and Serge Vernalde, editors, *FPL*, volume 3203 of *Lecture Notes in Computer Science*, page 1172. Springer, 2004. ISBN 3-540-22989-2. URL `http://dblp.uni-trier.de/db/conf/fpl/fpl2004.html#Ababei04`.

[106] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM. doi: 10.1145/1465482.1465560. URL `http://doi.acm.org/10.1145/1465482.1465560`.