Ultra-Fast FPGA Placement Using Steepest Gradient Descent Movement

Seppe Lenders

Supervisor: Prof. dr. ir. Dirk Stroobandt
Counsellors: Ir. Elias Vansteenkiste, Amit Kulkarni

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Electrical Engineering

Department of Electronics and Information Systems
Chair: Prof. dr. ir. Rik Van de Walle
Faculty of Engineering and Architecture
Academic year 2015-2016

UNIVERSITEIT GENT

Ultra-Fast FPGA Placement Using Steepest Gradient Descent Movement

Seppe Lenders

Supervisor: Prof. dr. ir. Dirk Stroobandt
Counsellors: Ir. Elias Vansteenkiste, Amit Kulkarni

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Electrical Engineering

Department of Electronics and Information Systems
Chair: Prof. dr. ir. Rik Van de Walle
Faculty of Engineering and Architecture
Academic year 2015-2016

UNIVERSITEIT GENT

Acknowledgements

I would like to thank the following people, without whom you would not be reading this master's dissertation right now. Prof. dr. ir. Dirk Stroobandt for making this research possible as leader of the HES research group within the ELIS department at Ghent University. Ir. Elias Vansteenkiste for the regular counseling and in-depth brainstorms and feedback, and for proofreading this text. My parents, for giving me all the space and time I needed to finish my studies and my thesis. And finally my girlfriend, for helping me define my priorities in a busy year.

Permission for use on loan

The author gives permission to make this master dissertation available for consultation and to copy parts of this master dissertation for personal use.

In the case of any other use, the copyright terms have to be respected, in particular with regard to the obligation to state expressly the source when quoting results from this master dissertation.

Seppe Lenders, 31 May 2016

Ultra-Fast FPGA Placement Using Steepest Gradient Descent Movement

Seppe Lenders

Supervisor: Prof. dr. ir. Dirk Stroobandt
Counsellors: Ir. Elias Vansteenkiste

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Electrical Engineering

Department of Electronics and Information Systems
Chair: Prof. dr. ir. Rik Van de Walle
Faculty of Engineering and Architecture
Academic year 2015-2016

Abstract

We propose a new heuristic for FPGA placement called gradient descent placement (GDP). It is based on the analytical placement technique HeAP and replaces the linear solving step found in analytical placers with a faster gradient-descent mechanism. GDP features both a wirelength-driven and a timing-driven mode. In wirelength-driven mode the performance and QoR is on par with analytical placement techniques. In timing-driven mode the wirelength and timing cost are about 5% lower and 8% higher respectively for equal runtimes. GDP is suited for massive parallelization on a GPGPU. We have released our full source code as part of a placement framework that is compatible with the VPR packing and routing steps. This framework can serve as a tool for future research into FPGA placement heuristics.

Keywords

FPGA, placement, analytical, gradient-descent, fast, timing-driven

Ultra-Fast FPGA Placement Using Steepest Gradient Descent Movement

Seppe Lenders

Promotor: Dirk Stroobandt, supervisor(s): Elias Vansteenkiste

Abstract— **We propose a new heuristic for FPGA placement called gradient descent placement (GDP). It is based on the analytical placement technique HeAP and replaces the linear solving step found in analytical placers with a faster gradient-descent mechanism. GDP features both a wirelength-driven and a timing-driven mode. In wirelength-driven mode the performance and QoR is on par with analytical placement techniques. In timing-driven mode the wirelength and timing cost are about 5% lower and 8% higher respectively for equal runtimes. GDP is suited for massive parallelization on a GPGPU. We have released our full source code as part of a placement framework that is compatible with the VPR packing and routing steps. This framework can serve as a tool for future research into FPGA placement heuristics.**

Keywords— **FPGA, placement, analytical, gradient-descent, fast, timing-driven**

I. Introduction

A field-programmable gate array (FPGA) is a reconfigurable integrated circuit, consisting of a grid of *blocks*. Every block contains configurable lookup tables (LUTs), flip-flops and an interconnection network. A configurable routing network connects block outputs to inputs. A detailed description of the architectures used in this work can be found in [3] and [4].

An FPGA circuit designer programs designs in a hardware description language (HDL). Through a synthesis process consisting of multiple steps this HDL description is converted to an FPGA configuration. The most time-consuming step of the synthesis is placement [4]. The input to the placement step is a collection of *clusters*. Each of these clusters performs a digital function, and can be implemented on one of the FPGA blocks. A cluster output can feed the inputs of one or more other clusters. We call a *net* the collection of one cluster output (the *net source*) and one or more cluster inputs (the *net sinks*).

The task of the placement step is to assign each cluster to a single block, ensuring each block is assigned at most one cluster. The quality of a placement can be evaluated in multiple ways, of which we discuss the two most common:

Total Wirelength. Every block output feeds a network of wires that is connected to other blocks' inputs. The total wirelength is the sum of the lengths of all these wires. In order to minimize the total wirelength closely interconnected clusters should be placed close to each other.

Critical Path Delay. One can follow a flip-flop's output along wires and blocks to another flip-flop's input. Every wire and block introduces a delay, the sum of which is the *path delay* for that path. The *critical path delay* (CPD) is the maximal path delay in the circuit. A low CPD allows for a high operating frequency. In order to minimize the CPD blocks on long paths should be placed close to each other.

The criticality of a block-to-block connection is a number between 0 and 1. A criticality of 1 means the connection is part of a path whose path delay is equal to the CPD. A criticality of 0 means the connection is only part of paths whose path delay is equal to 0.

II. Analytical Placement

A popular heuristic for FPGA placement is simulated annealing (SA). It has been applied to FPGA placement ever since the invention of FPGAs [6] and features a high placement quality but requires long runtimes. A more recent class of heuristics that shows promising results is analytical placement. Here we discuss one specific analytical placer called timing-driven analytical placement (TDAP). It is a timing-driven version of HeAP [5], developed at our department on which GDP is based.

TDAP consists of two steps that are executed alternately: linear solving and legal solving. The linear solver calculates the optimal position for each of the clusters, regardless of device restrictions. The resulting placement is *illegal*: clusters overlap and are positioned in between blocks. transforms this linear solution into legal placement on the FPGA. While doing so it tries to perturb the linear solution as little as possible.

A. Linear Solving

The linear solution is calculated in the x- and y-dimension separately. Here we discuss the x-problem, the y-problem is entirely analogous.

The linear solver build a system of N equations in N variables, with N the number of movable clusters in the circuit. The variables are x_1 through x_N: the positions of the movable clusters. Terms are added to the equations, such

that when the system is solved the resulting cluster positions minimize the total wirelength and/or the CPD.

Adding terms to the equations is equivalent to adding elastic (contracting) springs between clusters; solving the equations is equivalent to bringing the system of springs in equilibrium. In wirelength-driven mode springs are added according to the bound2bound net model: springs are added between the leftmost cluster and every other cluster, and between the rightmost cluster and every other cluster. In timing-driven mode additional springs are added between net sources and their sinks, but only for connections with a high criticality.

B. Legalization

When the linear x- and y-problem are solved, every cluster has a continuous position on the FPGA, as shown in Figure 1a. The legalizer maps each cluster to its closest block. For now we allow multiple clusters to be mapped to a single block. The legalizer builds a set of regions according to the following rules:

- Regions don't overlap.
- Regions are rectangular.
- Every region contains a number of blocks, and each block contains zero or more clusters. The total number of clusters inside a region is not bigger than the number of blocks in that region.

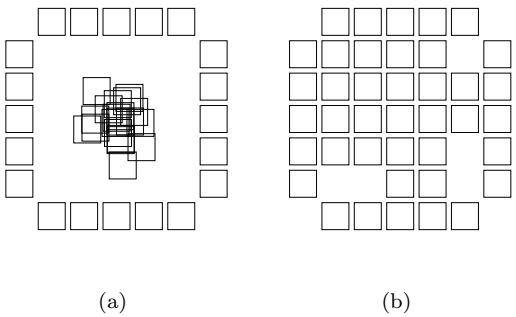
The legalizer always starts with a block that contains more than one cluster, a *overutilized* block. It keeps adding neighboring rows or columns to the region, until the region is not overutilized anymore. When a row or column is added that already (partially) belongs to another region, the regions are merged.

When a complete set of regions is found, each is legalized recursively. Consider a region with more columns than rows. The region is split in half yielding two new subregions (left and right) that are approximately equal in size. The clusters are sorted along the x-axis and the resulting ordered list is split in half as well. Both subregions are assigned their respective ordered sublist and subsequently legalized separately. The process for splitting a region with more rows than columns is analogous.

The recursive process stops when a region contains only one cluster. The cluster is mapped to the closest block in the region. After all regions have been processed a legal placement is obtained, as shown in Figure 1b

C. Pseudo Connections

The positions found during legalization are incorporated in the next linear solving step. Terms are added to the equations that pull the clusters to their legal position. In each linear solving step the legal positions from the last iteration are used, and the magnitude of the pulling forces increases throughout the iterations. Every iteration the linear solution attaches more importance to the legal positions, and less to the cost function.



(a) (b)

Fig. 1: Example of a linear (a) and corresponding legal (b) solution

III. Gradient Descent Placement

Our new heuristic gradient descent placement (GDP) is based on TDAP. The mechanism of alternating between linear and legal solving steps is the same and the legal solving step is unaltered. The linear solving step is approximated in an iterative way that resembles a classical gradient descent solving procedure. We call this the gradient solving step.

In what follows we describe one of these iterations. This procedure is repeated many times per legalization: up to 300 times for high quality placements.

A. A Gradient Solving Iteration

Springs. Like TDAP the x- and y-problem are solved separately, and we discuss only the x-problem. We adopt the concept of springs used in TDAP. The pulling force of a spring F_s depends on it's length d. Hooke's law imposes a linear relation between length and force, but we deviate slightly from this:

$$F_s = k\frac{d \times d_{max}}{d + {}^{d_{max}}/_{2}}$$

This formula reduces the force exerted by long springs, as shown in Figure 2. k and d_{max} are placer-wide parameters.

In wirelength-driven mode springs are added only between the leftmost and rightmost cluster of each net. In timing-driven mode additional springs are added between net sources and their sinks, but only for critical connections.

Step Direction and Size. A cluster is usually part of multiple nets, so multiple springs can be attached to a single cluster. When all springs have been added we count the number of springs on the left and right side of each cluster. A cluster will move in the direction with the most springs, regardless of their force. The step size

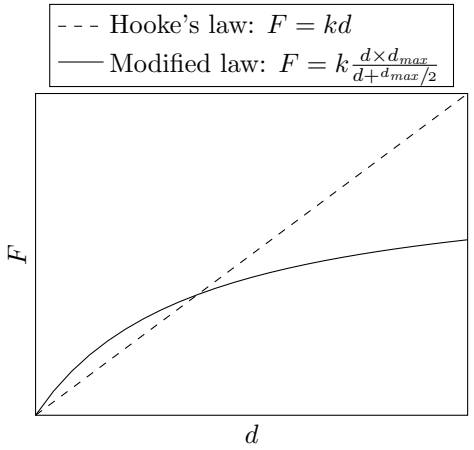


Fig. 2: Relation between spring distance and force: Hooke's law and the modified law used in GDP

is calculated as the average of the forces of the springs pulling in the step direction.

A position calculated this way is called an *optimal* position.

Including Legalized Positions. During all but the first gradient solving step legalized positions are known. Instead of moving a cluster to it's optimal position, we interpolate linearly between the optimal position and the linear position according to an interpolating factor w_{pseudo}. In the first gradient solving step the w_{pseudo} is zero: legalized positions are not taken into account. Every solving step a fixed value is added to w_{pseudo}: the importance of legalized positions increases. The algorithm stops when the interpolating factor reaches a predefined value, for example 0.9.

Speed Averaging. The calculation method described above leads to irregular cell movement. To reduce these irregularities slightly we average the speed of the cells over the iterations. This is done by calculating an extrapolated position based on the cell speed in the previous iterations. We interpolate linearly between this extrapolated position and the position calculated in the previous paragraph.

Figure 3 illustrates the calculation of a single cluster position in an iteration. To reduce clutter both the x- and y-dimension are shown.

B. Effort Level

We call the number of iterations in a gradient solving step the *effort level* of that iteration. There are two parameters that control the effort level in all solving steps: e_{first} and e_{last}, with $e_{last} \leq 1$. The effort level decreases linearly from e_{first} in the first solving step to $e_{last} \times e_{first}$ in the last solving step.

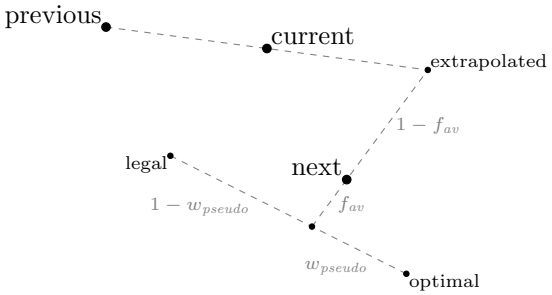


Fig. 3: Calculating the next position of a cluster using the previous position, the current position, the optimal gradient position and the legal position. The interpolation factors shown are $w_{pseudo} = 0.4$ and $f_{av} = 0.3$.

	SA	TDAP	GDP
runtime (s)	426	1.240	1.133
total wirelength	32,579	1.178	1.211

TABLE I: The performance of wirelength-driven SA, TDAP and GDP. Results for TDAP and GDP are shown relative to SA. Lower results are better.

IV. Results

We have used the Titan23 benchmark suite [4] to compare the performance of gradient descent placement (GDP) with timing-driven analytical placement (TDAP) and simulated annealing (SA). This suite contains benchmarks ranging in size from 25,000 to 800,000 LUTs, which is much larger than the benchmarks that are commonly used in academic research. The three largest benchmarks were excluded from our experiments: gaussianblur, directrf and LU_Network. VPR requires more than 64 GB of RAM to generate the timing information for these circuits, which we do not have at our disposal.

We run simulated annealing at the default VPR effort level of 1, GDP at an effort level of 300 and TDAP with an anchor weight multiplier of 1.007. The results are summarized in Table I for wirelength-driven modes and **??** for timing-driven modes.

In wirelength-driven mode the QoR of GDP is slightly lower than TDAP: about 3%. The gap with SA however is large for both TDAP and

	SA	TDAP	GDP
runtime (s)	3,361	0.301	0.186
total wirelength	43,589	1.050	0.996
CPD (ns)	22.8	0.997	1.072

TABLE II: The performance of timing-driven SA, TDAP and GDP. Results for TDAP and GDP are shown relative to SA. Lower results are better.

GDP: on average SA achieves a 15% lower wirelength cost than TDAP. This contradicts earlier research [5]. We believe this is due to our legalizer implementation. The implementation of the TDAP linear solving step is straightforward, but due to a lack of documentation we have had to design parts of the legalizer ourselves.

In timing-driven mode TDAP achieves approximately the same critical path delay as SA with a 5% increase in wirelength cost. GDP achieves approximately the same wirelength cost as SA with a 7% increase in critical path delay. Since we are primarily interested in minimizing the critical path delay we summarize this table as follows: TDAP performs almost as good as SA; GDP has a rather large gap in critical path delay.

The wirelength-timing trade-off in GDP tends too much to the wirelength side compared to TDAP and SA. Using the currently available parameters it is not possible move this balance more to the timing side, which is an important shortcoming.

V. Open Source Placement Framework

The only open-source framework available for placement research is VPR. It is written in C and contains large amounts of code that is hard to read, maintain or extend. In order to carry out the abovementioned experiments we have developed a placement framework written in Java. We have released the full source code on GitHub [1] and by doing so hope to lower the threshold to future FPGA placement research.

The framework is fully compatible with the VPR toolflow and replaces its placement step completely. The architecture used for placement is read in from a VPR architecture file. A part of the architecture specification is not supported yet.

In addition to the architecture file a packed netlist as generated by the VPR packer must be provided. The framework outputs a place file that is accepted by the VPR router. Currently three placement heuristics have been implemented: simulated annealing, TDAP and GDP. All heuristics have a wirelength- and timing-driven mode and can be controlled using command-line parameters. Thanks to a highly object-oriented, self-documenting code base adding more heuristics is straightforward.

Our framework is not capable of estimating the delay between two tiles on the FPGA. Instead we rely on VPR for these estimations. When starting the placement of a circuit VPR is called to calculate the delay lookup tables for that circuit. These tables are then read by our placer and used throughout placement. The critical path delay estimate for a placed circuit generated by our framework is always equal to VPR's.

VI. Conclusion and Future Work

We have created a new placement heuristic called GDP that is based on TDAP, an analytical placer developed at our department. In wirelength-driven mode there is a small performance gap with TDAP. In timing-driven mode the trade-off between wirelength and timing objectives tends too much to the wirelength side. As a result there is a significant gap in critical path delay.

We conclude that GDP does not perform better than TDAP. However the gap is small and we believe there are many opportunities for improvements.

The mechanism for combining springs should be reviewed and it's likely optimizations are possible. Different methods for incorporating timing information should be investigated to improve the wirelength-timing trade-off.

We believe it is possible to implement the gradient solving step of GDP on a general purpose GPU with little modifications. We have kept this future goal in mind during the development of the heuristic, but have not had time to do this ourselves.

The legalizer implementation should be refined. This will benefit both TDAP and GDP, and make them more competitive with simulated annealing.

Finally, we have create an open-source placement framework that is fully compatible with the VPR toolflow. We believe this will simplify future research into FPGA placement heuristics.

References

[1] The Java Placement Framework. `https://github.com/EliasVansteenkiste/FPGA-Placement-Framework`. Accessed: 2016-05-29.

[2] VTR user manual. `https://vtr.readthedocs.io/en/latest`. Accessed: 2016-05-29.

[3] Jason Luu, Jeffrey Goeders, Michael Wainberg, Andrew Somerville, Thien Yu, Konstantin Nasartschuk, Miad Nasr, Sen Wang, Tim Liu, Nooruddin Ahmed, et al. VTR 7.0: Next generation architecture and CAD system for FPGAs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 7(2):6, 2014.

[4] Kevin E Murray, Scott Whitty, Siyuan Liu, Jason Luu, and Vaughn Betz. Titan: Enabling large and complex benchmarks in academic CAD. In *23rd International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8. IEEE, 2013.

[5] Marcel Gort and Jason H Anderson. Analytical placement for heterogeneous FPGAs. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 143–150. IEEE, 2012.

[6] Carl Sechen and Kai-Win Lee. An improved simulated annealing algorithm for row-based placement. In *Proc. IEEE International Conference on Computer Aided Design*, pages 478481, 1987.

Contents

Acknowledgements **vii**

Abstract **ix**

Extended abstract **xi**

Contents **xv**

List of Abbreviations **xix**

List of Figures **xxi**

List of Tables **xxiii**

1 Introduction **1**

2 Background **3**
2.1 The Field-Programmable Gate Array (FPGA) . . . 3
2.1.1 A simple FPGA Architecture . . . 3
2.1.2 Heterogeneous FPGA Architectures . . . 4
2.1.3 Conventional FPGA Computer Aided Design (CAD) Flow . . . 6
2.1.4 Commonly Used Tools . . . 8
2.2 The Placement Problem . . . 9
2.2.1 Total Wirelength . . . 9
2.2.2 Critical Path Delay . . . 10
2.3 Simulated Annealing . . . 13
2.3.1 The General Algorithm . . . 13
2.3.2 Application to FPGA Placement . . . 13
2.3.3 Detailed and Greedy Placement . . . 14
2.4 Conclusion . . . 15

3 Analytical Placement **17**
3.1 Overview . . . 17
3.2 Linear Solving . . . 18
3.2.1 Building the Objective Function . . . 19
3.2.2 Net Models . . . 20
3.2.3 Pseudo Connections . . . 22
3.2.4 Timing-Driven Connections . . . 22
3.2.5 Solving the Linear System . . . 23

3.3 Legalization . . . 23
3.3.1 Overview . . . 23
3.3.2 Building the Regions . . . 23
3.3.3 Legalizing a Region . . . 25
3.4 Implementation details . . . 25
3.4.1 Incremental Sparse Matrix . . . 25
3.4.2 Two-Dimensional Linked List . . . 26
3.5 Conclusion . . . 28

4 Gradient Descent Placement **29**
4.1 Previous Work . . . 29
4.2 Modeling the Problem . . . 29
4.2.1 Adding the Springs . . . 30
4.2.2 Combining the Forces . . . 31
4.2.3 Timing-Driven Springs . . . 32
4.2.4 Including Legalized Positions . . . 32
4.2.5 Speed Averaging . . . 33
4.3 Optimizations . . . 34
4.3.1 Effort Level . . . 34
4.3.2 Recalculate Criticalities . . . 34
4.4 Running the Placer . . . 35
4.5 Conclusion . . . 37

5 Methods and Results **39**
5.1 The Java Placement Framework . . . 39
5.1.1 Overview . . . 40
5.1.2 Architecture Support . . . 40
5.1.3 Wirelength Cost Estimation . . . 40
5.1.4 Timing Cost Estimation . . . 41
5.2 Simulation Environment . . . 41
5.2.1 Benchmarks . . . 41
5.2.2 Running the Experiments . . . 43
5.2.3 Simulated Annealing . . . 43
5.2.4 TDAP . . . 44
5.3 Optimizing the GDP Parameters . . . 44
5.3.1 Optimizing Movement Parameters . . . 44
5.3.2 Runtime-Quality Trade-Off . . . 45
5.3.3 Wirelength-Driven Parameters: Summary . . . 46
5.3.4 Timing-Driven Parameters . . . 46
5.4 Comparison With Analytical Placement and Simulated Annealing . . . 47
5.4.1 Wirelength-Driven . . . 47
5.4.2 Timing-Driven . . . 50
5.5 High Effort Performance . . . 54

6 Conclusions and Future Work **57**
6.1 Conclusions . . . 57
6.1.1 A Fast Placement Heuristic . . . 57
6.1.2 An Open-Source Placement Framework . . . 57
6.2 Future Work . . . 58

6.2.1 Architecture Support 58
6.2.2 Better Legalization for TDAP and GDP 58
6.2.3 More Powerful Timing-Driven Version 58
6.2.4 Parallelization 59

Bibliography **61**

List of Abbreviations

AP	analytical placement
ASIC	application-specific integrated circuit
BB	bounding box
CAD	computer-aided design
CSR	compressed sparse row
FF	flip-flop
FPGA	field-programmable gate array
GDP	gradient descent placement
GPGPU	general-purpose computing graphics processing units
HDL	hardware design language
HPWL	half-perimeter wirelength
IO	input/output
TDAP	timing-driven analytical placer
LB	logic block
LE	logic element
LUT	lookup table
PCB	printed circuit board
RAM	random access memory
SA	simulated annealing
VPR	Versatile Packing, Placement and Routing
VTR	Verilog to Routing

List of Figures

2.1 The operation of a D flip-flop 4
2.2 A simple logic element (LE) 4
2.3 A simple logic block (LB) 5
2.4 Global view of a simple FPGA 5
2.5 Detail of a routing network using a switchblock 5
2.6 Global view of a heterogeneous FPGA with two types of hard blocks 6
2.7 The FPGA synthesis flow 7
2.8 The runtime of the synthesis steps 8
2.9 Two nets 10
2.10 The scaling factor $q(n)$ used for calculating the HPWL 11
2.11 A simple timing graph 11

3.1 Example of a linear (a) and corresponding legal (b) solution 18
3.2 Different net models 21
3.3 The process of growing a region until it is not overutilized 24
3.4 The process of merging two regions 24
3.5 Splitting a doubly linked list in two 27

4.1 Relation between spring distance and force 31
4.2 A cluster on which five forces pull 31
4.3 Calculating the next position of a cluster 34
4.4 Recalculating the criticalities every step is not necessary 35
4.5 The placement cost over the course of a GDP execution 36

5.1 The trade-off between runtime and wirelength cost for the *other benchmarks* . 48
5.2 The trade-off between runtime and wirelength cost for the Titan23 benchmarks 49
5.3 The trade-off between wirelength and timing cost for TDAP and GDP 50
5.4 The trade-off between runtime and QoR for the *other benchmarks* 52
5.5 The trade-off between runtime and QoR for the Titan23 benchmarks 53

List of Tables

2.1 A 2-input boolean function 4

5.1 Details for the eight largest of the *other benchmarks* 42
5.2 Details for 20 Titan23 benchmarks 42
5.3 The performance of our simulated annealing placer relative to the VPR placer . 43
5.4 Comparison of high quality wirelength-driven SA, TDAP and GDP for the Titan23 benchmarks 55
5.5 Comparison of high quality timing-driven SA, TDAP and GDP for the Titan23 benchmarks 56

Chapter 1

Introduction

This thesis treats the subject of field-programmable gate array (FPGA) placement. This is an optimization problem with an enormous solution space. In order to find good solutions in an acceptable time heuristics must be used. Different solution methods have been proposed and are currently in use, often with large runtimes. In this thesis we propose a new, fast solution method called gradient descent placement (GDP).

An FPGA is an electronic device. It can be configured repeatedly without intervention of the manufacturer. Most commonly used FPGAs consist of a matrix of tiles. These tiles can be configured to compute logic or mathematical functions and/or to store bits of information. A configurable network of electronic interconnections wires tiles to one another. This device can be programmed to implement the behavior of an arbitrary digital circuit. The building blocks of an FPGA are discussed in chapter 2.

An FPGA is programmed using a hardware description language (HDL). A computer synthesizes this human-readable code into a configuration for the used FPGA-device, much in the same way that a compiler translates source code written in a programming language into machine code. An important and time-consuming step during this synthesis is the placement. The input for the placement step is a list of circuit components (clusters) and the way they must be connected to each other. Each of these clusters must be assigned a position on the FPGA. In other words: each clusters must be *placed* on exactly one tile of the FPGA. No tile can contain more than one cluster, and the placement should be chosen so that strongly interconnected clusters are placed close to each other. More details on FPGA synthesis and placement in particular are provided in chapter 2.

Analytical Placement and Gradient-Descent Placement

A family of placement methods that features high quality placements and low runtime is analytical placement (AP). AP heuristics typically consist of two steps that are repeated until convergence occurs. The first step calculates the optimal position for each of the clusters. These optimal positions need not be aligned with the FPGA tiles. Furthermore, if we were to assign each cluster to it's closest tile, many tiles would contain more than one cluster, which is not allowed. We call this an optimal but *illegal* placement.

The second step is call *legalization*. The clusters are redistributed until all clusters are aligned with a tile and each tile contains at most one cluster. The placement is *legal* but suboptimal. The problem from the first step is now modified using information from the legal placement. The modified problem is solved again, yielding a placement that is more similar to the legal placement. This modified solution is again legalized, etc. The placer alternates between these two steps until the illegal and legal placement are the approximately the same. This heuristic is discussed in detail in chapter 3.

In this thesis we propose gradient descent placement (GDP), a new placement method based on AP. In AP the optimal but illegal solution is calculated by modeling the problem as a set of equations with the cluster positions as the unknown variables. Solving the equations yields the optimal cluster positions. The idea for GDP comes from the observation that needless effort is spent calculating the exact solution to the equations. Subsequent to every linear solving step the optimal solution is heavily perturbed during legalization. We assumed that an approximate solution to the linear system might be good enough for our purpose. The details of this new heuristic will be explained in chapter 4.

Chapter 5 compares the performance of GDP with that of existing algorithms. Chapter 6 concludes this work and proposes subjects for future work.

Chapter 2

Background

2.1 The Field-Programmable Gate Array (FPGA)

A field-programmable gate array (FPGA) is a reconfigurable integrated circuit. FPGAs are digital devices: every signal that propagates through an FPGA is either 0 or 1. An FPGA can be reprogrammed any amount of times without intervention of the device manufacturer. It can be used to implement the behavior of an arbitrary application specific integrated circuit (ASIC).

First we will describe the architecture of a fictional FPGA device. Many of today's FPGAs are based on this simplified architecture. We will not discuss the architectures used in this thesis because the detailed inner workings of an FPGA are not essential for understanding the placement problem. A detailed description of the used architectures can be found in [12] and [16].

We will then describe the process of programming an FPGA, and discuss some of today's commonly tools for doing so.

2.1.1 A simple FPGA Architecture

FPGAs contain two types of elementary building blocks: lookup tables and flip-flops. A lookup table (LUT) is a device with two or more inputs and one output. For every possible combination of input signals, an output signal (0 or 1) is defined. In other words: the LUT implements a boolean function. FPGA LUTs usually have four to six inputs, and can be configured to implement an arbitrary boolean function. Table 2.1 shows a possible boolean function with two inputs.

A flip-flop (FF) is a clocked device that serves as a 1-bit memory. FPGAs use D flip-flops. They have one input. On each clock tick, the FF output is updated so that it matches its input. The output signal remains unchanged at least until the next clock tick. Figure 2.1 illustrates the operation of a D flip-flop.

A LUT and a FF are packed together into a unit that is commonly referred to as a logic element (LE), as shown in Figure 2.2. The multiplexer pictured on the right is used select the output signal.

input	output
0 0	1
0 1	0
1 0	1
1 1	0

Table 2.1: A 2-input boolean function

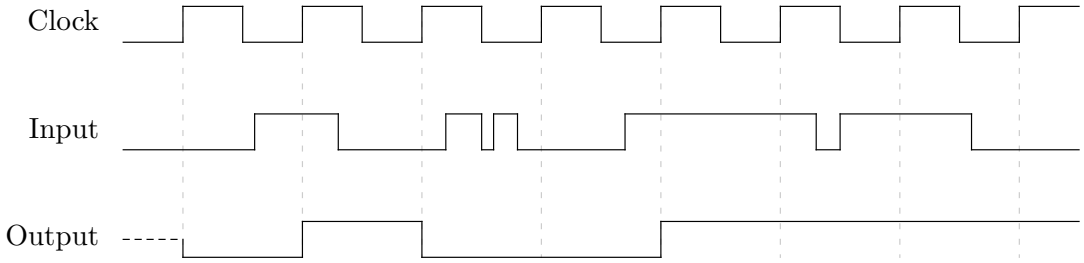


Figure 2.1: The operation of a D flip-flop

We group ten LEs into what is called a logic block (LB), shown in Figure 2.3. A LB has 30 inputs and 10 outputs to receive and send signals from other LBs. The interconnection network inside the LB is made configurable by means of a crossbar. This crossbar connects LB inputs and LE outputs to LE inputs. Each LE output has a fixed connection to a LB output.

We arrive at a global view of the FPGA in Figure 2.4: logic blocks are arranged in rows and columns. On the edges of the FPGA are input/output (IO) blocks. These are connected to pins on the FPGA package, and allow for communication with an external electronic circuit.

A global interconnection network connects the outputs of blocks to the inputs of other blocks. We call this the routing network. On every junction between four logic blocks there is a switchblock. Fixed routing tracks are laid out between adjacent switchblocks. A switchblock can be configured to connect certain tracks to each other. Furthermore the inputs and outputs of logic blocks can connect to tracks on that side of the block. An example of a configured routing network is depicted in Figure 2.5.

2.1.2 Heterogeneous FPGA Architectures

The FPGA architecture we have so far described is homogeneous: every logic block is identical. This type of FPGA is capable of implementing any electronic circuit, but certain operations

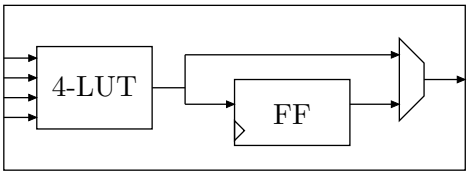


Figure 2.2: A simple logic element (LE)

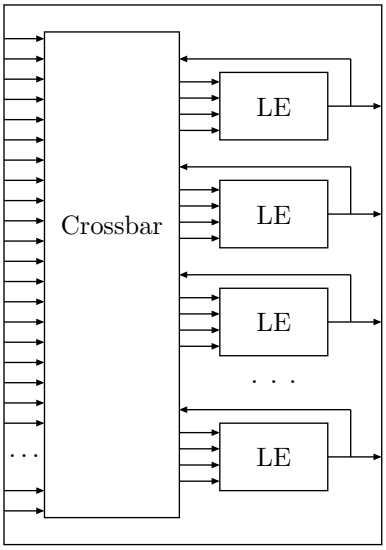


Figure 2.3: A simple logic block (LB)

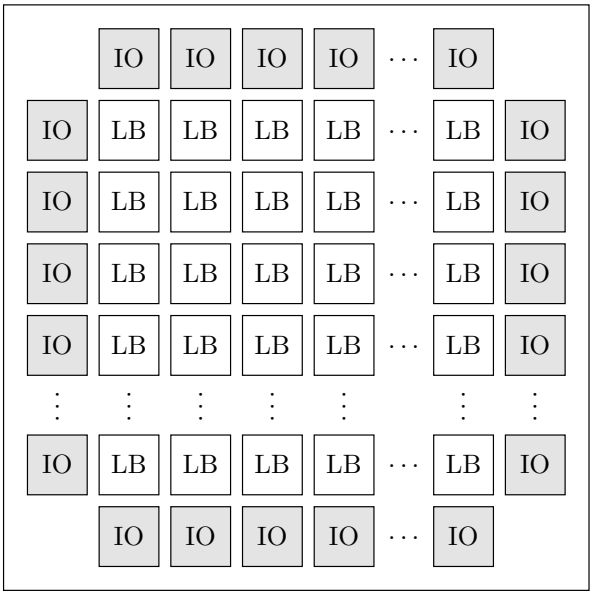


Figure 2.4: Global view of a simple FPGA

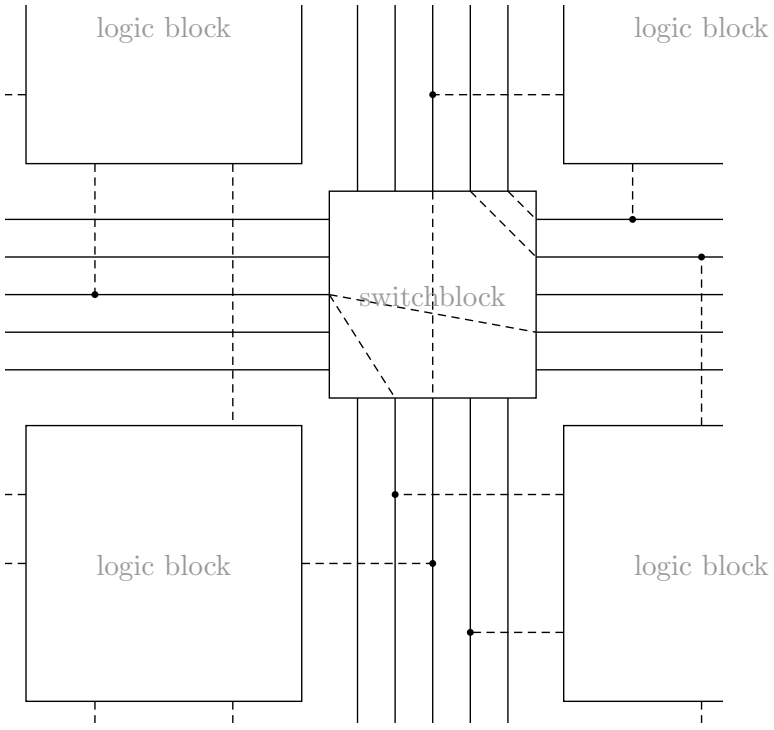


Figure 2.5: Detail of a routing network using a switchblock. Dashed connections are configurable.

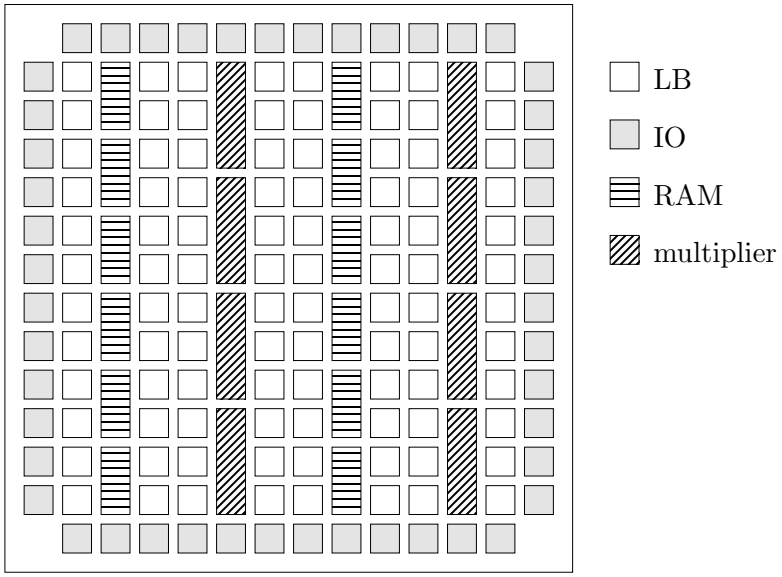


Figure 2.6: Global view of a heterogeneous FPGA with two types of hard blocks

can only be executed slowly. These operations can be sped up by replacing columns of logic blocks with columns of hard blocks. Examples of hard blocks are multipliers, random access memory (RAM) and digital signal processing (DSP) blocks that perform multiply and accumulate operations. A hard block can have a height that spans multiple logic blocks, as shown in Figure 2.6.

The routing network of the FPGA is slow because signals have to propagate through crossbars and switch blocks. In order to speed up long paths carry chains can be used. These are direct connections from one logic block to to an adjacent block. Inside a logic block the carry chain usually propagates from one logic element to the next as well. This type of connection enables efficient arithmetic operations like counters and adders. In this thesis only vertical carry chains are considered: a block can have a direct connection only to the block below and/or above it.

Most FPGA architectures are more complex than the one discussed so far. Logic elements have multiple operation modes, and crossbars and switchblocks are not fully populated. The architectures used in this thesis are the comprehensive architecture from [12] and an architecture modeled after the Stratix IV device, as described in [16].

2.1.3 Conventional FPGA Computer Aided Design (CAD) Flow

A circuit designer usually programs his designs in a hardware description language (HDL). This is a computer language similar to a programming language but with a notion of time. It takes into account the time it takes to perform calculations. Commonly used HDLs are Verilog, VHDL and SystemC.

A computer can synthesize an HDL program to an FPGA configuration or any other type of electronic circuit like a printed circuit board (PCB) or application-specific integrated circuit (ASIC) design. This process is analogous to compiling a computer program: a high-level human-readable circuit description is transformed into a configuration that can be directly im-

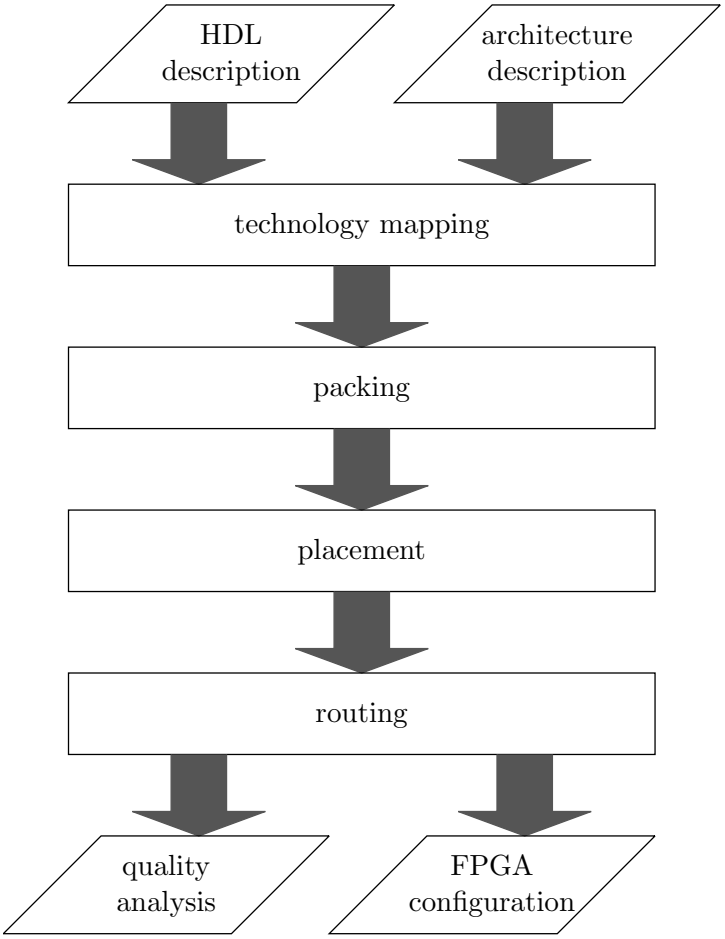


Figure 2.7: The FPGA synthesis flow

plemented onto one specific technology. Figure 2.7 shows the entire process of FPGA synthesis. We will briefly describe each step.

Logic synthesis The first step translates an HDL description into a graph of logic primitives. These primitives are LUTs with an unlimited amount of inputs and flip-flops. Depending on the FPGA architecture other primitives like multipliers can be included.

Technology mapping The logic primitives that an FPGA can implement are limited in size. For example some fictional FPGA device could contain 4-LUTs, 5-LUTs, flip-flops and 16-bit multipliers. During technology mapping the graph of logic primitives is transformed into a new graph that contains only primitives of the available sizes. For example an 8-LUT can be split up into three 4-LUTs, and a 32-bit multiplier can be split up into four 16-bit multipliers and a number of adders.

Packing We explained in section 2.1.1 how each logic block in an FPGA contains a fixed amount of elementary blocks like LUTs and flip-flops. Logic primitives in the circuit description

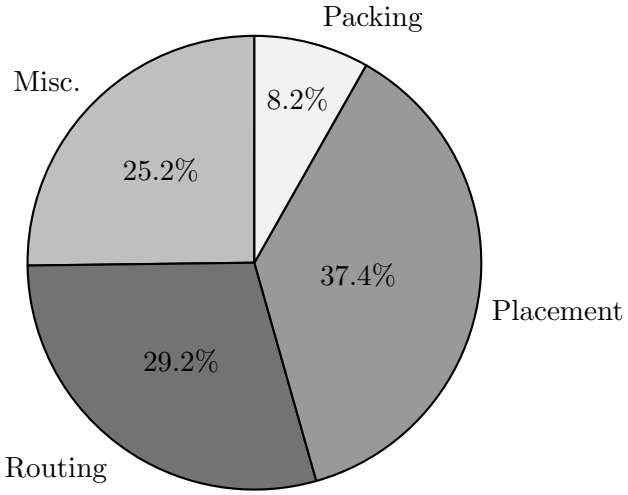


Figure 2.8: The runtime of the synthesis steps in Altera's Quartus II CAD flow

must be implemented on one of these elementary blocks. A packer groups logic primitives together into what we will call a cluster. The contents of each cluster should fit in a single logic block. Intra-LB connections are much faster than inter-LB connections, so the packer aims to group closely interconnected clusters together.

Placement The logic blocks on the FPGA are arranged in rows and columns. Every cluster must be assigned to one of these logic blocks. Again this should be done in such a way that closely interconnected logic blocks are placed close to one another. Since the subject of this thesis is a new placement method, the placement step is discussed in detail in section 2.2.

Routing The final step in FPGA synthesis is routing. In the placement step each logic primitive has been assigned a position on the FPGA. The routing step configures the routing channels and switchblocks (see Figure 2.5) in such a way that the logic blocks are connected correctly.

Runtime Breakdown

In [16] the runtimes of packing, placement and routing in the Altera Quartus II flow were investigated for a set of large circuits. The results are summarized in Figure 2.8. Placement is the most time-consuming of the three steps, occupying on average 37% of the runtime.

2.1.4 Commonly Used Tools

Vendors of FPGA devices provide tools for the design and synthesis of circuits, targeting their own devices. The two major FPGA manufacturers are Xilinx and Altera, who have a combined market share of approximately 90% [15]. Altera's design suite is called Quartus while Xilinx has a different suite depending on the device: ISE and Vivado.

The current state of the art open source synthesis tool is Verilog to Routing (VTR). It performs a complete synthesis starting from a Verilog program and an architecture description. There are three main components in VTR. ODIN II and ABC perform the technology mapping. Versatile Packing, Placement and Routing (VPR) handles the remaining three steps. After synthesis VTR analyses the quality of the resulting configuration. It is not commonly used to configure actual FPGA devices, but rather for academic research in FPGA architecture and synthesis algorithms/heuristics.

Proprietary tools outperform open source tools both in runtime and quality of results (QoR). According to [7] VPR is 3.5 times slower than Xilinx ISE while the resulting critical path (see section 2.2.2) is on average 56% longer.

2.2 The Placement Problem

The synthesis steps described above can take up to several hours for large circuits. It is obvious that speedups in this process can increase the circuit designer productivity. Placement is one of the most time-consuming steps in FPGA synthesis. It takes on average 37% of the total runtime in Altera's Quartus II CAD flow [16]. The goal of this thesis is to decrease placement runtime.

Placement is a complex optimization problem. An FPGA device can bee seen as a grid of blocks of different types (see Figure 2.6). Prior to placement a packer has grouped logic primitives together into clusters. Each cluster should be assigned to a block of the correct type, and no block can contain more than one cluster. This type of optimization problem is called an *assignment problem*. In FPGA architectures with carry chains there is an additional condition: blocks with a direct connection between them must be placed next to each other. We call these collections of clusters that are glued together *macros*.

During placement there are several circuit properties one can try to optimize. The two most common ones are the *total wirelength* and the *critical path delay*. The total wirelength leads to more convex optimization functions, and is generally easier to optimize. We make a distinction between *wirelength-driven* placers that optimize only the wirelength, and *timing-driven* placers that optimize both properties simultaneously.

2.2.1 Total Wirelength

On a global level, an FPGA consists of blocks and nets. A net is a connection between one block output (the net source) and one or more block inputs (the net sinks). A block output or input cannot be connected to more than one net. Figure 2.9 shows two nets: one with a single sink and one with three sinks. The total wirelength of an design is simply the sum of the wirelengths of all the nets.

Inter-block connections are laid out in the routing step, so they are not known during placement. Connecting the net blocks using the minimal amount of wirelength is equivalent to building a minimum rectilinear Steiner tree. This is a NP-hard problem, it would take too much time to repeatedly solve this type of problem during placement. It is therefore not feasible te calculate the exact wirelength of a net while placing. The wirelength of a net is often approximated

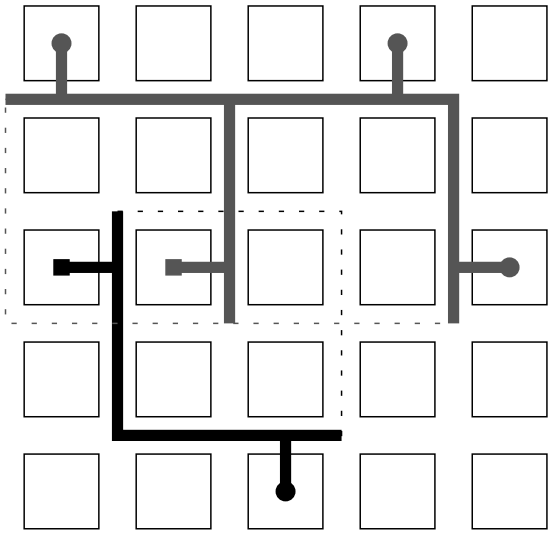


Figure 2.9: Two nets and their bounding boxes (dashed lines). The net sources are marked with a circle, the net sinks are marked with a square.

using the half-perimeter wirelength (HPWL). This is simply the sum of the two dimensions of the bounding box: bb_x and bb_y.

The HPWL systematically underestimates the net wirelength, and the estimation error grows with the number of blocks in the net. This can be seen in Figure 2.9: we call the distance between two blocks the unit size, and exclude the small wires that connect to the blocks. The HPWL of the net with four blocks is then $bb_x + bb_y = 4 + 2 = 6$. However the actual required wirelength is 8.

VPR compensates for this deviation by multiplying the HPWL by a parameter q that depends on the number of blocks in a net, as shown in Figure 2.10. The wirelength cost of a net is then equal to:

$$HPWL_{VPR} = q(n) \times (bb_x + bb_y) \tag{2.1}$$

2.2.2 Critical Path Delay

We can see the circuit as a graph of clocked nodes (e.g. flip-flops) and non-clocked nodes (e.g. LUTs). The nodes are connected by wires. Both the wires and the non-clocked nodes add a delay to every signal that propagates through them. We call this a *timing graph*. A simple example is shown in Figure 2.11.

A path is an (indirect) connection between two clocked nodes. It starts at the output of a clocked node (the path source), passes through zero, one or more non-clocked primitives and ends at the input of a clocked node. The path delay is the sum of the node and connection delays on the path. The critical path is the path with the longest delay in the entire design. The maximum clock frequency of the circuit is equal to the inverse of the critical path delay.

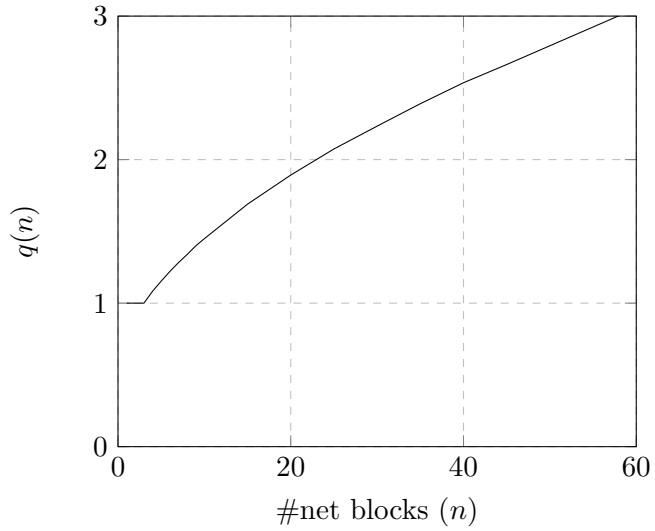


Figure 2.10: The scaling factor $q(n)$ used for calculating the HPWL of a net as a function of the number of blocks in the net n

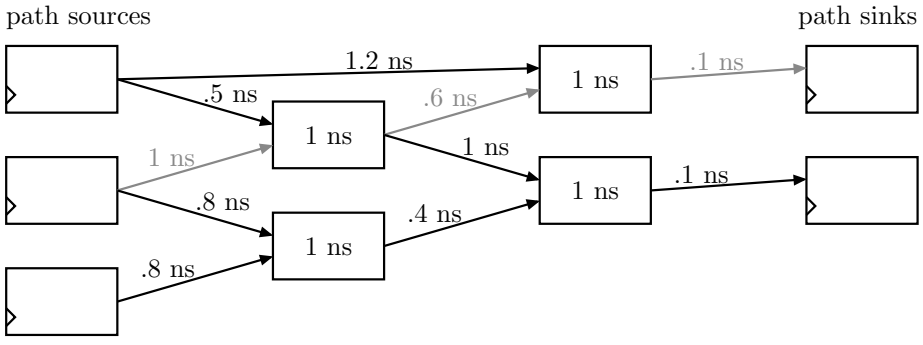


Figure 2.11: A simple timing graph. The critical path is drawn in gray.

In Figure 2.11 the critical path goes from the middle-left node to the top-right node, and has a delay of 3.2 ns.

The critical path delay should be minimized in order to maximize the circuit operating frequency. However the critical path delay cannot be calculated accurately during placement, because the routing of the inter-block connections is not yet known. In VPR the estimated delay between two blocks depends only on the horizontal (Δx) and vertical (Δy) distance between the two blocks [13]. Prior to placement a delay lookup matrix is built. The matrix element at position $[i, j]$ stores the minimum possible delay for a connection of horizontal length i and vertical length j.

Criticality

VPR and many other timing-driven placers make use of the concept "criticality" to optimize the critical path delay. Every connection has an associated criticality, whose calculation is outlined here in compliance with [13].

The timing graph source nodes are given an arrival time of 0. The arrival times of the other nodes are calculated with a breadth-first search:

$$T_{arrival,i} = \max_{j \in sources(i)} (T_{arrival,j} + delay_{i \to j}) \tag{2.2}$$

The critical path delay is equal to $D_{max} = \max_i (T_{arrival,i})$ The required time of the timing graph sink nodes is set to D_{max}. The other nodes' required times are calculated with a reverse breadth-first search:

$$T_{required,i} = \max_{j \in sinks(i)} (T_{required,j} - delay_{i \to j}) \tag{2.3}$$

The slack of a connection is:

$$slack_{i \to j} = T_{required,j} - T_{arrival,i} - delay_{i \to j} \tag{2.4}$$

And finally, the criticality of a connection:

$$criticality_{i \to j} = 1 - \frac{slack_{i \to j}}{D_{max}} \tag{2.5}$$

Connections with a high criticality have a high probability of becoming a part of the critical path. A timing-driven placer gives these connections a higher priority when minimizing wirelengths.

2.3 Simulated Annealing

In the previous section we have seen two metrics that can be used to evaluate the quality of a placement. We will now describe the simulated annealing (SA) heuristic, a popular approach to the FPGA placement problem. It is a conceptually simple heuristic which may help the reader in grasping the placement process.

2.3.1 The General Algorithm

Simulated annealing is an optimization heuristic that can be applied to a diverse range of problems. The name comes from the process of metallurgic annealing where a hot material is slowly cooled down, allowing crystals to grow and defects to disappear. The problem should have a discrete search space, and each state in the search space should have a known or calculable cost. SA then finds a state with a cost that approaches the optimal (minimal or maximal) cost over all possible states.

The heuristic starts with an initial state s and a temperature T. In each iteration it finds a new state s' that is very similar to s: a neighboring state. If the heuristic transitions to the new state the current cost changes with an amount $c = cost(s') - cost(s)$. The probability of accepting s' as the new state is a function of the temperature T and the magnitude of c. The probability increases with increasing T and decreasing c (when minimizing the cost). At the end of each iteration the temperature is updated. As the heuristic progresses the temperature slowly decreases: the heuristic is "cooling down". The heuristic finishes when Algorithm 2.1 describes this procedure in pseudocode.

Algorithm 2.1 General simulated annealing

```
procedure SIMULATED ANNEALING(T, s)          ▷ Temperature T, initial state s
    while !STOPCRITERION() do
        s' = GETNEIGHBORSTATE(s)
        if P(COST(s), COST(s'), T) > RANDOM(0, 1) then
            s ← s'
        T ← UPDATETEMPERATURE()
    return s
```

2.3.2 Application to FPGA Placement

Simulated annealing has been a popular heuristic for FPGA placement ever since the invention of FPGAs [19]. It remains in wide use, e.g. in the Altera Quartus II and VPR synthesis tools [13]. In SA placement every possible placement is a state. Neighboring states are placements in which the positions of two clusters has been swapped.

The placer starts with a random placement, with a temperature that is so high that almost all swaps (state transitions) are accepted. It tries N swap transitions, and then updates the temperature. To maintain a high acceptance probability throughout the heuristic the search space shrinks as the heuristic progresses. This means that the maximal distance between two swapping blocks gradually decreases. The heuristic stops when the temperature is lower than a fraction of the current placement cost. Algorithm 2.2 describes the procedure in pseudocode.

Algorithm 2.2 Simulated annealing placement

```
procedure SIMULATED ANNEALING PLACEMENT(innerNum, stopT)
    p ← RANDOMPLACEMENT()
    c ← COST(s)
    T ← INITIALTEMPERATURE()
    dist_max ← CIRCUITSIZE()                ▷ Maximum distance between swapping blocks

    while T > T_stop × c do
        numAcceptedSwaps ← 0
        for i ← 1 : innerNum do
            s ← RANDOMSWAP(p, dist_max)
            Δc ← DELTACOST(s)                          ▷ Negative if cost decreases

            if exp(−Δc/T) > RANDOM(0, 1) then          ▷ Negative cost is always accepted
                APPLYSWAP(p, s)
                c ← c + Δc
                numAcceptedSwaps ← numAcceptedSwaps + 1

        T ← UPDATETEMPERATURE(numAcceptedSwaps)
        dist_max ← UPDATEMAXSWAPDISTANCE(numAcceptedSwaps)
```

2.3.3 Detailed and Greedy Placement

Some placement heuristics are only good at global placement: starting from a random placement and finding a reasonably good position for each cluster. They fail however at detailed placement: making small adjustments to the global placement to optimize the quality.

We have altered our simulated annealing heuristic slightly so that it can take over this detailed placement step from another heuristic. The simulated annealer is called with an initial placement, so calculating a random placement is not needed anymore. We make the assumption that the initial placement could have been created by the simulated annealing placer. The initial placement quality is not sufficient; this means the simulated annealing would have been cut off early, at some temperature T_{stop}. For an optimal runtime-quality trade-off the detailed placer should be started at that same temperature. We estimate this temperature using the method described in [18].

The initial value of $dist_{max}$ is chosen prior to placement, and should reflect the quality of the initial placement. For high-quality global placements, $dist_{max} = 3$ is a good choice, lower quality placements require a higher value.

For fast detailed placement the simulated annealing placer can be run in greedy mode. The initial temperature is set to 0 which means swaps are only accepted if they they decrease the cost function. The outer loop is executed exactly once. $dist_{max}$ is again chosen prior to placement, and the value of *innerNum* is usually set higher than for a non-greedy placement.

2.4 Conclusion

We have briefly described the concepts that are necessary to understand the remainder of this thesis, most importantly:

- The global architecture of FPGA devices
- The placement problem, for which we will later propose a new solution approach
- The properties of a good placement

We have finished by discussing simulated annealing, a widely used and conceptually simple placement heuristic. In the next chapter we explain a different and more involved approach: analytical placement.

Chapter 3

Analytical Placement

The simulated annealing approach to FPGA placement discussed in the previous chapter yields placements with a very high quality, at the cost of a high runtime. Circuit sizes have grown over the years, and placing a large circuit with the VPR placer can now take up to multiple hours, which is often impractical.

A lot of research has gone into speeding up placement. One approach is to parallelize the simulated annealing heuristic. This approach has yielded speedups of 2.2× on four cores with no quality loss [11] and 10× using GPGPU with very small quality losses [4].

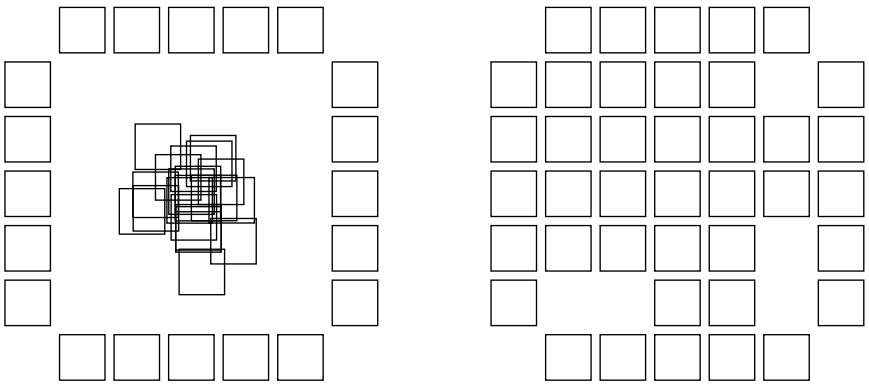
Other heuristics have been proposed to solve the placement problem. One class of heuristics that shows promising results, is analytical placement. Analytical placers were first invented in the 1980s in the field of ASIC design [9] and continue to be widely used in that field. For FPGAs, Xilinx first started using analytical techniques in 2007 [6]. Academic research to analytical placers is still lagging behind proprietary tools in terms of QoR.

Multiple analytical placers have been proposed in academic literature that are competitive with the VPR placer, most notably [5, 23, 10, 22, 21]. None of these placers have publicly available source code. A master's thesis student at our department has created a timing-driven analytical placer [14] which we will call TDAP. This placer is based on the wirelength-driven placer HeAP [5], which is in turn based on an analytical placer for ASICs called SimPL [8].

It is essential to understand the inner workings of TDAP in detail. The main contribution of this thesis is a new placement heuristic which is based on TDAP. It will be discussed in the next chapter.

3.1 Overview

Analytical placement generally consists of two steps that are executed alternately: linear solving and legalization. The linear solver calculates the optimal position for each of the clusters, regardless of device restrictions. What constitutes an optimal position and how it is calculated depends on the placer. The resulting placement is *illegal*: clusters overlap and are positioned in between blocks. The legalizer transforms this linear solution into legal placement on the



(a) (b)

Figure 3.1: Example of a linear (a) and corresponding legal (b) solution

FPGA. While doing so it tries to perturb the linear solution as little as possible. Figure 3.1 shows an example linear and legal solution.

The legalizer typically doesn't take all optimization requirements into account. As a result the placement found by the legalizer after one cycle is suboptimal. The linear solver is once again called, this time with extra information so that the linear solution will approach a legal placement a little more. This cycle is repeated, and each cycle the difference between the linear and legal solution will become smaller. The heuristic stops when the two solutions are similar enough.

A completed placement is often refined using a swap-based placer, either a simulated annealing placer that is started at a very low temperature or a greedy swapping placer that accepts only swaps that decrease the cost.

3.2 Linear Solving

During placement a circuit is treated as a collection of clusters, and interconnections from to output of one cluster to the inputs of one or more other clusters. The linear solver finds an optimal x-coordinate and y-coordinate for each cluster. TDAP solves this problem for the two dimensions separately. We will describe the problem in the x-dimension. The problem in the y-dimension is entirely analogous.

The TDAP linear solver finds an optimal coordinate x_i for each movable cluster in the circuit: $1 \leq i \leq N_{clusters}$. IO-clusters are not moved: they retain the original position on the edges of FPGA. This is a reasonable limitation: the FPGA input and output position are often fixed by the circuit design outside the FPGA. The solver formulates a convex objective function with $N_{clusters}$ variables: x_1 through $x_{N_{clusters}}$. The objective function is minimized by solving it's derivative using any of the many existing linear solving methods.

3.2.1 Building the Objective Function

The objective function takes the form

$$\Phi(\vec{x}) = \frac{1}{2}\sum_{i,j} w_{i,j}(x_i - x_j)^2, \qquad 1 \leq i,j \leq N_{clusters} + N_{IO} \tag{3.1}$$

where $w_{i,j}$ is the weight of the connection between cluster i and cluster j. Connections can exist between both movable and fixed clusters.

An equivalent matrix form is:

$$\Phi(\vec{x}) = \frac{1}{2}\vec{x}^T Q_x \vec{x} + \vec{c}_x^T \vec{x} + const. \tag{3.2}$$

where Q_x represents the connections between two movable clusters and $\vec{c}_x$ represents the connections between movable and fixed clusters. The objective function is minimized by setting it's derivative to zero:

$$\nabla\Phi(\vec{x}) = Q_x\vec{x} + c_x = 0 \tag{3.3}$$

A connection between two clusters i and j adds the following terms to the objective function:

$$w_{i,j}\left(\frac{1}{2}x_i^2 + \frac{1}{2}x_j^2 - x_i x_j\right) \tag{3.4}$$

If both cluster i and cluster j are movable this contributes $w_{i,j}$ to $Q_x[i,i]$ and $Q_x[j,j]$, and $-w_{i,j}$ to $Q_x[i,j]$ and $Q_x[j,i]$. If cluster i is movable and cluster j is fixed this contributes $w_{i,j}$ to $Q_x[i,i]$ and $w_{i,j}x_j$ to $c_x[i]$. By solving the derivative the weighted sum of the squared lengths of all included connections is minimized.

Macros

In section 2.2 we explained that a macro is a collection of clusters whose mutual positions are fixed. A macro consists of multiple clusters, but only has a single coordinate variable x_m. Every cluster in the macro has a fixed offset to this coordinate $x_{o,i}$.

Consider a cluster i in macro m with coordinate $x_m + x_{o,i}$ and a cluster j in macro n with coordinate $x_n + x_{o,j}$. A connection between these two clusters adds the following terms to the objective function:

$$w_{i,j}\left[\frac{1}{2}x_m^2 + \frac{1}{2}x_n^2 - x_m x_n + (x_{o,i} - x_{o,j})(x_m - x_n) + \frac{1}{2}(x_{o,i} - x_{o,j})^2\right] \tag{3.5}$$

For clusters that are not part of a macro $x_{o,i} = x_{o,j} = 0$ and the above equation simplifies to Equation 3.4.

3.2.2 Net Models

Only direct connections between two clusters can be added to the objective functions. Nets – which are connections from one source cluster to one or more sink clusters – must be represented as a collection of cluster-to-cluster connections. To do this various *net models* exist, as depicted in Figure 3.2.

(a) The most straightforward is the clique net model, in which every cluster in a net is connected to every other cluster in the net. For large nets the number of connections grows quickly and the net model becomes impractical to use.

(b) In the star net model a dummy cluster is created for each net. It's coordinate is equal to the average of the coordinates of the clusters in the net. Every cluster in the net is connected to this dummy cluster. This net model has been used with success recent analytical placers like StarPlace [22].

For large nets the star net model greatly reduces the number of connections compared to the clique net model. For nets with two or three clusters however it increases the amount of required calculations. Sometimes the clique net model is used for these smaller nets [21].

(c) SimPL [8], HeAP [5] and TDAP [14] use a bound2bound net model. It first selects the two clusters with minimal and maximal coordinate. Every internal cluster is connected to these extreme clusters, and an extra connection is added between the two extreme clusters. In this net model the connections included in the x-dimension are different from the connections included in the y-dimension.

The bound2bound net model is good at minimizing the bounding box cost of a net because it exerts the most force on the extreme clusters.

(d) The source-sink net model connects the net source to each sink, but doesn't add any connections between sinks. It is used in TDAP for timing-driven connections (see section 3.2.4).

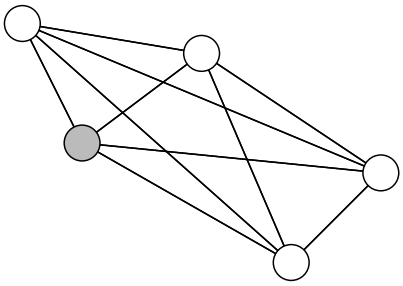
(e) The extreme points net model connects is proposed in our gradient descent placer (see section 4.2.1). It is similar to the bound2bound net model, but only includes one connection: the one between the two extreme clusters.

TDAP uses the bound2bound net model. The weight of each connection depends on the number of clusters in the net (n). It is chosen so that the estimated net wirelength equals the sum of net connections' weights:

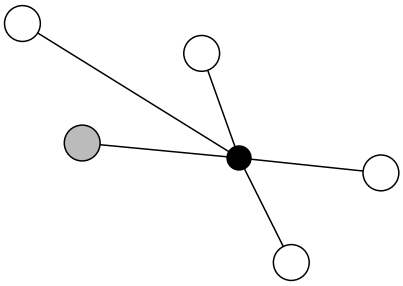
$$w_{i,j} = \frac{q(n)}{n-1} \tag{3.6}$$

See section 2.2.1 for the meaning of the factor $q(n)$. Because the linear solver can only solve quadratic wirelength, each connection weight is further multiplied with the inverse of the current connection length:

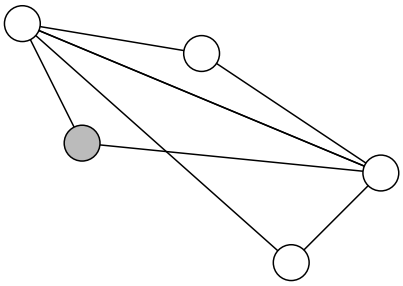
$$w_{i,j} = \frac{1}{|x_{i,prev} - x_{j,prev}|} \frac{q(n)}{n-1} \tag{3.7}$$

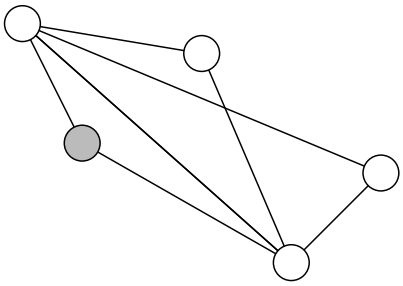


(a) Clique net model

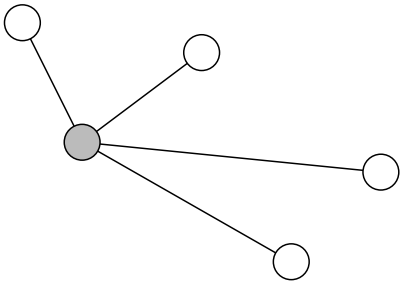


(b) Star net model

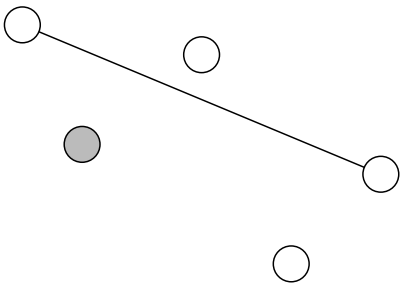


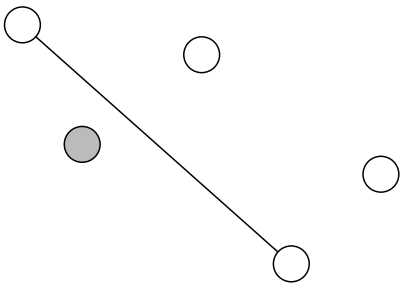


(c) Bound2bound net model for the x-dimension (left) and y-dimension (right)



(d) Source-sink net model





(e) Extreme points net model for the x-dimension (left) and y-dimension (right)

Figure 3.2: Different net models. The net source is colored in gray.

This linearization factor doesn't enable the quadratic solver to actually minimize the linear wirelength, but it does approximate it. A previous position of all blocks is required for the calculation of the weights. In the first run of the linear solver the coordinates of a random legal placement are used. In subsequent runs the previous linear solution is used.

3.2.3 Pseudo Connections

Subsequent linear solutions should be more and more similar to a legal solution. To accomplish this pseudo connections are added from a cluster to its legal position found in the last legalization phase. In the first linear solving phase – when no legalization has taken place yet – no pseudo connections are added. Every iteration the weight of the pseudo connections increases, which reduces the differences between the linear and legal solution.

A pseudo connection can be seen as a connection between a movable cluster (the actual cluster) and a fixed cluster (the legal position). Adding pseudo connections happens in the exact same way as described in section section 3.2.1. The weight for these connections is the same for all pseudo connections. It is a placer-wide parameter w_{pseudo} that is low at the beginning of the placement, and increases every iteration. It is not necessary to linearize the connection weights, in fact this reduces the quality of the placement.

In [5] and [14] the value of the parameter w_{pseudo} increases linearly through placement: every iteration a fixed value is added to the parameter. We have found that exponential growth produces comparable results in slightly less iterations.

3.2.4 Timing-Driven Connections

The timing-driven placer adds a third type of connections to the linear system: source-sink connections. As explained in section 2.2.2 every connection from a cluster output to a cluster input has a criticality *crit*. A connection from cluster i to cluster j is added to the linear system with the weight

$$w_{i,j} = \begin{cases} 0, & crit < \theta_c \\ \frac{tradeoff \times crit^{\epsilon_c}}{|x_{i,prev} - x_{j,prev}|}, & \text{otherwise} \end{cases} \tag{3.8}$$

There are three placer-wide parameters that determine the weight of the source-sink connections:

- The criticality threshold θ_c. Source-sink connections with a criticality lower than θ_c are not added to the linear system. Increasing θ_c speeds up calculations. Losses in quality are small if θ_c is not set too high, because the influence of non-critical connections on the linear system can be neglected.
- *tradeoff* determines the importance of timing-driven source-sink connections compared to wirelength-driven bound2bound connections.
- The criticality exponent ϵ_c. A higher criticality exponent puts more emphasis on minimizing the wirelength of the most critical connections, at the cost of an increased total wirelength.

3.2.5 Solving the Linear System

When all connections are added to the linear system we have a set of equations of the form

$$\mathbf{Ax} = \mathbf{b} \tag{3.9}$$

Where $\mathbf{A}$ is a $n \times n$ symmetric and positive-definite matrix and n is the number of movable clusters. This linear system can be solved using any of the widely available linear solvers. HeAP and TDAP use the iterative conjugate gradient method. This is a fast algorithm that can only be applied when $\mathbf{A}$ is both symmetric and positive-definite.

3.3 Legalization

The fundamentals of our legalizer are the same as those of the legalizer described in HeAP [5]. However the details provided in that paper are scarce, and no source code is publicly available. We have had to design parts of the legalizer heuristic ourselves.

The legalizer first considers only clusters of type A. When all of these clusters are legalized, it legalizes the clusters of type B, and so on. No conflicts between the cluster types can arise: every block on the FPGA can only contain clusters of exactly one type. We now describe the process of legalizing all the clusters of a given type.

3.3.1 Overview

The legalizer starts by mapping each cluster to the closest block of the correct type. For now we allow multiple clusters to be mapped to a single block, which is not legal. The legalizer proceeds by building a set of regions on the FPGA according to the following rules:

- Regions don't overlap.
- Regions are rectangular.
- Every region contains a number of blocks, and each block contains zero or more clusters. The total number of clusters inside a region is not bigger than the number of blocks in that region.

When a set of regions is found every region is legalized. The clusters are spread out in such a way that each cluster is mapped to a block, and no block contains more than one cluster.

3.3.2 Building the Regions

The legalizer first investigates the block that is closest to the center of the FPGA. It then spirals outwards until all blocks have been investigated.

No action is taken if the investigated block contains zero or one clusters, or if the block is already contained in a region. If the block contains more than one cluster and hasn't been assigned

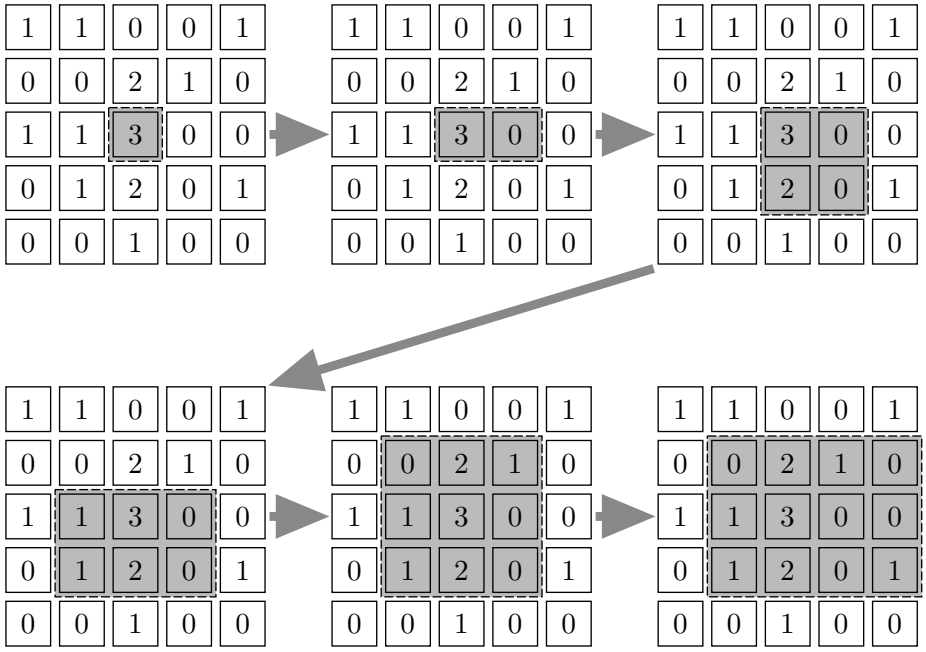


Figure 3.3: The process of growing a region until it is not overutilized. The numbers inside each block denote the number of clusters mapped to that block.

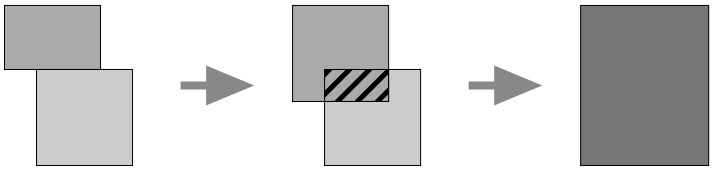


Figure 3.4: The process of merging two regions. When the top region grows to the bottom it overlaps with the bottom region. Both regions are merged into one bigger region.

to a region, a new region is created. The new region initially contains just the investigated block. We call this region overutilized: it contains more clusters than blocks. The region is then grown until it is no longer overutilized.

Because regions are always rectangular, growing a region is equivalent to adding a row or a column of blocks to the region. When selecting the next row or column that will be added, the legalizer tries to:

- Keep the region approximately square.
- Keep the initial block approximately in the center of the region.

Figure 3.3 illustrates the process of growing a region until it is not overutilized anymore.

When a block must be added to the region, but the block already belongs to another region, the two regions are merged. This means that a new rectangular region is created that contains both the involved regions. This process is depicted in Figure 3.4.

Once the region is not overutilized anymore, the legalizer finds a new overutilized block.

3.3.3 Legalizing a Region

Legalizing a single region is a recursive process. The region is split into two smaller regions, and the clusters are distributed among these two regions so that the total deviation from the cluster's optimal positions is minimized, and both subregions are not overutilized. The two subregions are then legalized separately.

Consider a rectangular region with width w and height h, and $h >= w$. The following discussion is analogous for the case where $h < w$. Two rectangular subregions are created with dimensions $w_1 = w_2 = w$, $h_1 = \lceil \frac{h}{2} \rceil$ and $h_2 = \lfloor \frac{h}{2} \rfloor$. The blocks in the region are sorted along the y-axis according to the positions found in the linear solving phase, forming a list of size N. This list of blocks is split into two lists with size N_1 and N_2 so that $N = N_1 + N_2$ and $\frac{w_1 h_1}{w_2 h_2} \approx \frac{N_1}{N_2}$. The two lists of blocks are assigned to the two subregions, and both are legalized separately. This recursive call continues until a region contains only one cluster. The cluster is then mapped to the block in the region that is closest to the cluster's optimal position, as found in the linear solving phase.

Legalizing Regions With Macros

The above process works only if there are no macros in the circuit (see section 2.2 for an explanation of macros). When there are macros the legalizer may encounter regions that are impossible to legalize. An example is a region with $w = 2$ and $h = 3$ that contains three macros, each of which is two blocks high. The region is not overutilized since the number of clusters equals the number of blocks, but no legal placement exists for this region. In these cases the parent region is greedily legalized.

The greedy legalizer sorts the clusters in the x-direction. Every column in the region is assigned a number of clusters, so that no column is overutilized and the total horizontal deviation of the clusters is minimized. In every column the clusters are sorted in the y-direction, and placed from top to bottom.

3.4 Implementation details

We have put much effort into optimizing the runtime of all the implemented heuristics. We will discuss two custom data structures that are an essential part of the analytical placer.

3.4.1 Incremental Sparse Matrix

The linear system from section 3.2.1 is stored in-memory in a sparse matrix. This matrix is square, it's dimensions are equal to the number of movable clusters in the circuit. It is not practical to store this matrix in a two-dimensional array: for large circuits this would require large amounts of RAM.

We store the matrix in a compressed sparse row (CSR) fashion. This representation requires little memory and allows us to carry out all the required computations quicly. It is however

not practical to construct the matrix in an incremental way: inserting or updating a single element takes on average $O(\sqrt{n})$ time, with n the number of elements in the matrix.

We use an intermediate data structure while constructing the matrix that can easily be transformed to a CSR matrix: a two-dimensional array. Each array represents a matrix row. The elements of the arrays are tuples of the form (column index, value). An array can contain more than one tuple with the same column index: the values of these tuples must be summed to obtain the actual matrix element's value. To update the value of a matrix element we simply append a tuple to the array that corresponds to the element's row.

When constructing the CSR matrix, the tuples in every array are sorted based on their column index. The values of consecutive tuples with the same column index are summed, and the final element values are appended to the CSR matrix. The complexity of this transformation is $O(n \log \sqrt{n})$.

3.4.2 Two-Dimensional Linked List

When legalizing a region (see section 3.3.3) the following steps are repeated:

1. Sort a list of blocks along the x-axis
2. Split the list in two approximately equal parts
3. Sort both sublists along the y-axis
4. Split both sublists in two approximately equal parts
5. Repeat for all four obtained sublists

It is inefficient to forget the ordering along one axis while sorting along the other axis. We designed a linked list whose elements are simultaneously sorted along two axes. The list can be split into two sorted sublists in $O(n)$ time.

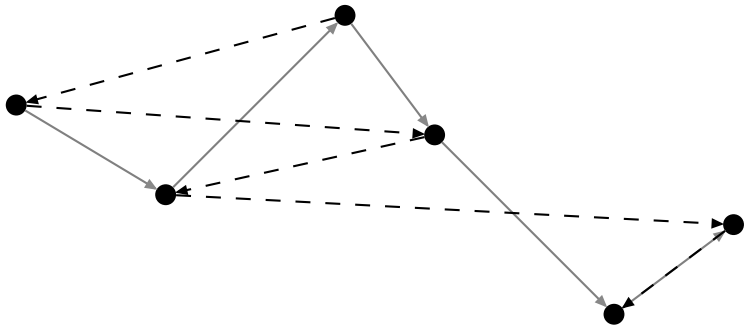
In a regular linked list each list node contains a value (`node→value`) and a pointer to the next node (`node→next`). In our application the values are coordinates: they contain the properties `node→value.x` and `node→value.y`. The list is sorted along the x-axis if and only if

$$\forall \texttt{node} : \texttt{node→value.x} < \texttt{node→next→value.x} \tag{3.10}$$

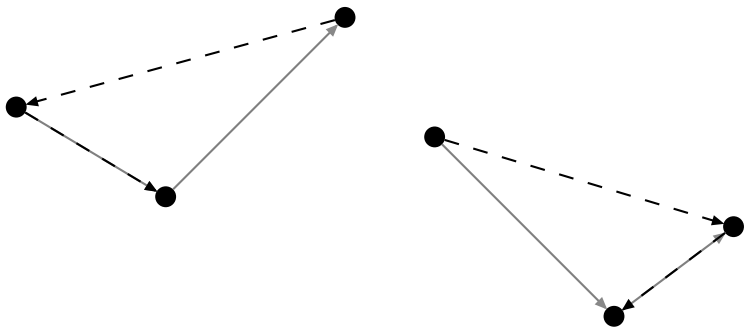
(Analogous for the y-axis.)

In our linked list implementation every node contains two pointers, one for each axis (`n→next_x` and `n→next_y`). After adding all the coordinates to the list, we sort the list twice: once for each axis. This step takes $O(n \log n)$ time and has to be executed only once for each legalization region.

When we split the list along the x-axis, the `next_x`-pointers should only be updated for the edge nodes, the other pointers are still valid. The `next_y`-pointers are invalid and must be updated. For this we iterate over the original list in sorted y-direction. The `next_y`-pointer of every node is updated so that it points to the first subsequently encountered node that is in the same sublist as the current node. This takes $O(n)$ time. Figure 3.5 illustrates this process.



(a) The original linked list



(b) The splitted linked list

Figure 3.5: Splitting a doubly linked list in two along the x-axis. `next_x` and `next_y`-pointers are drawn in solid gray and dashed black respectively.

3.5 Conclusion

In this chapter we have explained the inner workings of an analytic placer that we have called TDAP. This placer alternates between two types of solving steps:

- A linear solving steps yields an optimal but illegal solution
- A legalization steps yields a legal but suboptimal solution

TDAP can be run in two modes: in wirelength-driven mode it optimizes the total wirelength cost, in timing-driven mode the wirelength cost and critical path delay are optimized simultaneously.

In the next chapter we will discuss a new placement heuristic developed for this thesis, which is based on TDAP.

Chapter 4

Gradient Descent Placement

The main contribution of this thesis is a new placement heuristic that we call gradient descent placement (GDP). It is based upon TDAP, the analytical placer described in chapter 3. The mechanism of alternating between linear and legal solving steps is the same and the legal solving step is unaltered. The only difference between the two solvers is the linear solving step. TDAP builds a linear system and calculates the exact solution to this system. GDP approximates this solution in an iterative way. This iterative procedure resembles a classical gradient descent solving procedure, hence the name.

The linear solving step is replaced by an iterative procedure which we will call the gradient solving step. In this chapter the word "iteration" denotes one iteration within a gradient solving step. In every iteration, all the clusters are moved over a small distance. The directions and sizes of these steps are chosen so that the total cost function decreases. The steps are calculated for each cluster individually: there is no global view of the optimization problem at hand. The x-and y-problem are treated separately, as was the case in the analytical placer. In this chapter we will only discuss the x-problem; the y-problem is completely analogous.

4.1 Previous Work

In the last few years there has been a focus in FPGA research on analytical placers. The construction of the linear system varies, but it is always solved entirely using an off-the-shelf linear solver. In [20] the authors describe a process of moving clusters around the FPGA in a way that seems similar to our gradient placer. However it provides very little details, and no indication about the legalizing process.

4.2 Modeling the Problem

Every cluster c is seen as a block that can move in one dimension, e.g. along the x-axis. It is connected with extended springs to fixed locations and to other movable blocks. Every one of these springs exerts a pulling force on the cluster. The magnitude F_s of this force increases with the wire length. Every iteration these forces are combined into a single force with a direction

and a magnitude F_c. In that iteration the cluster will move in that direction over a distance F_c.

In this section we describe the mechanics of adding and combining springs for both the wirelength- and timing-driven version of the placer. All introduced parameters will be optimized in chapter 5.

4.2.1 Adding the Springs

The Extreme Point Net Model

The cost function for wirelength-driven placement is the sum of the half-perimeter wirelengths of all the nets in the circuit. Minimizing the cost function in the x-dimension is accomplished by minimizing the widths of all nets simultaneously. The width of a net is determined by it's leftmost and rightmost cluster: the extreme clusters. The positions of the internal clusters do not matter.

For every net in the circuit we only add one spring, namely between the two extreme clusters. This is the "extreme points net model" as shown in Figure 3.2e on page 21. As a result of this single spring the two extreme clusters will be pulled towards each other without affecting the internal clusters.

At first sight this rudimentary net model doesn't seem to work very well: it does not keep all the clusters in the net together. If an internal cluster of the net is also an extreme cluster in some other net, as is often the case, it can be pulled away far from the other clusters of the net. However every gradient solving step consists of multiple iterations (up to fifty for high-quality solutions). Every iteration different clusters end up at extreme positions. Over the course of multiple iterations the placement converges to a good solution. Experiments with the star and bound2bound net models yielded inferior results.

Spring Force

The magnitude of a spring's pulling force F_s depends on the distance d_{c_1,c_2} between the two clusters. Initially we applied Hooke's law, which means the spring force is proportional to this distance:

$$F_{s,Hooke} = k d_{c_1,c_2} \tag{4.1}$$

All springs have the same stiffness k: this is a placer-wide parameter that influences the step size of the clusters.

We expect that the influence of long springs might be too big: a single long spring pulls hard on a cluster, possibly overshadowing the pulling forces of several small springs pulling in the other direction. For this reason we have experimented with limiting the maximal force a spring can exert. We have obtained good results with the formula:

$$F_s = k \frac{d_{c_1,c_2} \times d_{max}}{d_{c_1,c_2} + {}^{d_{max}}/_2} \tag{4.2}$$

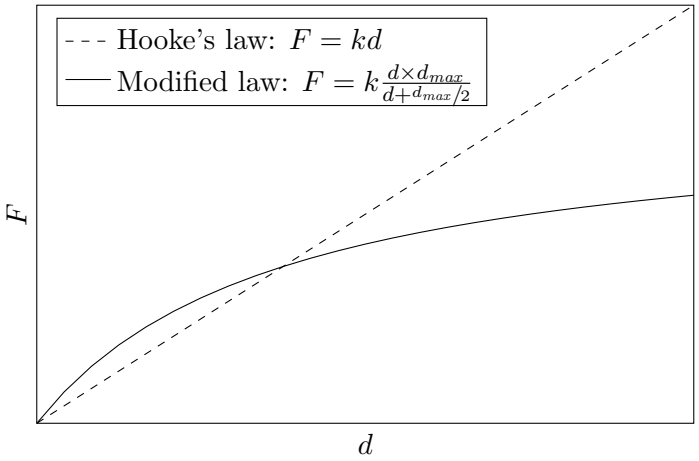


Figure 4.1: Relation between spring distance and force: Hooke's law and the modified law used in GDP

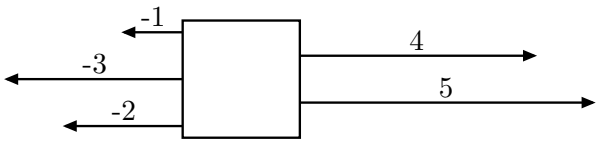


Figure 4.2: A cluster on which five forces pull

where d_{max} is a placer-wide parameter. Using this formula an infinitely long spring will only exert the force that a spring of length d_{max} would exert according to Hooke's law:

$$\lim_{d \to \infty} F_s(d) = kd_{max} = F_{s,Hooke}(d_{max}) \tag{4.3}$$

Figure 4.1 plots Hooke's law and our modified law.

4.2.2 Combining the Forces

As most clusters are part of a large number of nets, many will be an extreme cluster in more than one net. This results in multiple forces pulling on a single cluster, e.g. as shown in Figure 4.2. Using these forces we will calculate a step direction and size that decreases the cluster cost function.

In our small example the cluster is an extreme cluster in five nets: three nets on the left and two other nets on the right. We call this the direction cardinality: $cardinality_{left} = 3$ and $cardinality_{right} = 2$. It is always true that the cluster cost function decreases in the direction with the largest cardinality. In order to minimize the cost function to cluster should take a sufficiently small step in this direction. This is equivalent to descending along the gradient of the cost function, which explains the name of our placer: Gradient Descent Placer.

We now know the direction in which each cluster should step, but not the sizes of the steps. We use the average magnitude of the forces in the step direction as the step size. This step

size is likely to bring the cluster closer to it's cost function minimum quickly. In the example of Figure 4.2 this calculation leads to a step of size 2 to the left.

Alternative step sizes were investigated but yielded inferior results

- The geomean of the spring forces instead of the mean
- The minimal force, or a fraction thereof

4.2.3 Timing-Driven Springs

The timing-driven version of GDP adds a second type of springs: between the net source and net sinks (see Figure 3.2d on page 21). The magnitude of these forces is calculated in the same way as the wirelength-driven forces, i.e. according to Equation 4.2. To balance the importance of wirelength objectives against timing objectives we introduce a new property for every force: the weight.

Every spring in the system is given a weight w_s. The weight of wirelength-driven springs is fixed to 1. Every timing-driven spring is associated with a connection with criticality $crit$. It's weight is closely related to weight of timing-driven connections in TDAP (see Equation 3.8 on page 22):

$$w_s = \begin{cases} 0, & crit < \theta_c \\ tradeoff \times crit^{\epsilon_c}, & \text{otherwise} \end{cases} \tag{4.4}$$

For the meaning of the placer-wide parameters θ_c, $tradeoff$ and ϵ_c we refer to section 3.2.4.

To determine the direction that a cluster will move to GDP does not simply count the number of springs on each side anymore: it now sums the weights of the springs. The cluster will move in the direction that has the highest summed weight. Once the direction is known, the weights do not influence the magnitude of the step: the step size still equals the average of the force magnitudes, as explained in section 4.2.2

4.2.4 Including Legalized Positions

GDP follows the same pattern of gradient solving and legalization as TDAP. In the first gradient solving step GDP places all movable clusters very close to each other near the center of the FPGA, much in the same way that TDAP did. In all subsequent gradient solving steps GDP incorporates the legal positions found in the most recent legalization step.

TDAP includes legal positions by means of pseudo-connections. These are forces that pull a movable cluster to it's legal position. These forces are entirely equivalent to the other forces in the linear system that pull two clusters to each other.

We have tried to use this concept of pseudo-connections in GDP as well. As the heuristic progresses, the importance of pseudo-connections should gradually increase. We did this by giving the connections a "weight", as defined in section 4.2.3. In wirelength-driven mode it only makes sense to increase this weight in steps of 1. However the number of springs attached to

an average cluster is small, so the effect of increasing the weight by 1 is large. As a result there is a pronounced turning point in the course of the heuristic. Before this point the influence of the legalized positions is very small. After this point the clusters get quickly pulled to their legal positions and almost no changes in the placement occur for the rest of the heuristic.

Instead of using pseudo-connections we incorporate legalized information by interpolating between the optimal position and the legal position.

Linear Interpolation

We call x_i the position of a cluster at the beginning of iteration i, and $x_{i,l}$ the legal position for that cluster found in the last legal solving step. If we ignore the legal position of a cluster, the cluster movement is only determined by the wirelength- and timing-driven springs. We calculate a good new position for the cluster as explained in section 4.2.2. We call this position $x_{i,s}$. We combine these two different positions by interpolating linearly between them:

$$x_{i,sl} = (1 - w_{pseudo})x_{i,s} + w_{pseudo}x_{i,l} \tag{4.5}$$

w_{pseudo} is a placer-wide variable that indicates the importance of a cluster's legal position during a gradient solving step. It changes as the placement progresses. In the first gradient solving step w_{pseudo} is equal to 0: the legal position is not taken into consideration. Every gradient solving step a fixed value $w_{pseudo,step}$ is added to w_{pseudo}: the influence of the legal position increases. The placer stops when w_{pseudo} reaches a predefined value $w_{pseudo,stop}$, typically chosen below and close to 1.

We have now introduced two new parameters that make a trade-off between the placer speed and quality of the final placement: $w_{pseudo,step}$ and $w_{pseudo,stop}$. We want to highlight an important consequence of this calculation method: the number of solving steps is determined only by pre-defined parameters, and can be calculated prior to starting the placement. In TDAP the heuristic's execution is stopped when the placement quality reaches some satisfactory level which means the placement quality has to be recalculated every iteration. GDP never needs to know the current placement quality, allowing it to avoid a lot of time-consuming calculations.

4.2.5 Speed Averaging

$x_{i,sl}$ is still not the final position for the cluster in iteration i. There is one step left: taking into account the step size from the previous iteration. First we calculate a position $x_{i,av}$ by adding the cluster step from iteration $i-1$ to x_i:

$$x_{i,av} = x_i + (x_i - x_{i-1}) \tag{4.6}$$

We then interpolate linearly between $x_{i,sl}$ and $x_{i,av}$:

$$x_{i+1} = (1 - f_{av})x_{i,sl} + f_{av}x_{i,av} \tag{4.7}$$

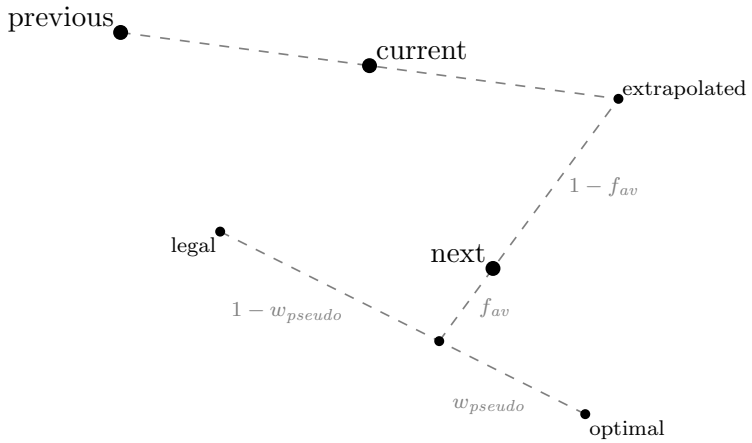


Figure 4.3: Calculating the next position of a cluster using the previous position, the current position, the optimal gradient position and the legal position. The interpolation factors shown are $w_{pseudo} = 0.4$ and $f_{av} = 0.3$.

This calculation can be seen as taking the exponentially weighted moving average of the speed of a cluster over multiple iterations. This reduces irregular cluster movement over the iterations. f_{av} is a placer-wide parameter: the speed averaging factor. It should take a value between 0 and 1; lower values make for a faster decay.

Figure 4.3 summarizes the calculations involved in determining the next cluster position x_{i+1}. To avoid clutter it shows both the x- and y-dimension.

4.3 Optimizations

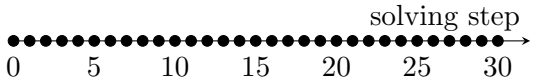
4.3.1 Effort Level

Every gradient solving steps consists of a number of cluster movement iterations. We call the number of iterations in a solving step the effort level. It is an important parameter to control the placement quality: a higher effort level leads to a higher QoR.

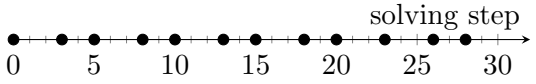
As the heuristic progresses, the average cluster movement in an gradient solving step becomes smaller. We expect that throughout the heuristic the number of iterations needed to find the optimal positions decreases. For this reason we have added two parameters to the placer: e_{first} and e_{last}. The number of iterations in a solving step is interpolated linearly between e_{first} in the first solving step and $e_{last} \times e_{first}$ in the last step.

4.3.2 Recalculate Criticalities

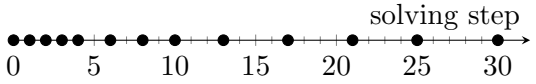
The timing graph contains the criticality for each connection. These criticalities of course depend on the positions of the clusters, and must be updated as the placement heuristic progresses. We can recalculate the criticalities prior to each gradient solving step, as shown in Figure 4.4a. However this recalculation is time consuming. We expect it may be possible



(a) $F_c = 1$



(b) $F_c = 0.4$, $P_c = 1$



(c) $F_c = 0.4$, $P_c = 2$

Figure 4.4: Recalculating the criticalities every step as in (a) is not necessary. Fewer recalculations can be spread (b) uniformly or (c) with an emphasis on the first solving steps.

to yield faster results with the same quality if the criticalities are not calculated every solving step. We introduce a new parameter F_c; this is the fraction of solving steps that require a recalculation. Figure 4.4b shows the recalculations for $F_c = 0.4$.

As the heuristic progresses the average cluster movement decreases, so we assume changes in the timing graph also reduce. This leads us to expect that recalculations are more important in earlier solving steps. We introduce another parameter P_c that controls the recalculation priority. When $P_c = 1$ the recalculations are distributed uniformly over the solving steps, as in Figure 4.4b. Higher values of P_c put more recalculations in the earlier solving steps, as shown in Figure 4.4c.

4.4 Running the Placer

Figure 4.5 plots the cost of the placement over the course of the heuristic. We let the placer start from a random placement, which has a very high cost. In the first gradient solving step no legal positions are known, so the placer simply minimizes the cost. The legalizer calculates a legal placement based on this optimal but illegal placement.

As the heuristic progresses there are less gradient solving steps between legalizations, as explained in section 4.3.1. Throughout the heuristic the cost of the gradient and legal solution converge. The influence of the legal positions on the gradient solving step increases, causing the clusters to move further away from their optimal but illegal positions and leading to a higher gradient cost. However the gradient solving step finds a placement that increasingly resembles a legal placement, requiring less and less disturbance by the legalizer and leading to a lower legal cost.

Note that the costs plotted in Figure 4.5 are not actually calculated while running GDP, as explained in section 4.2.4.

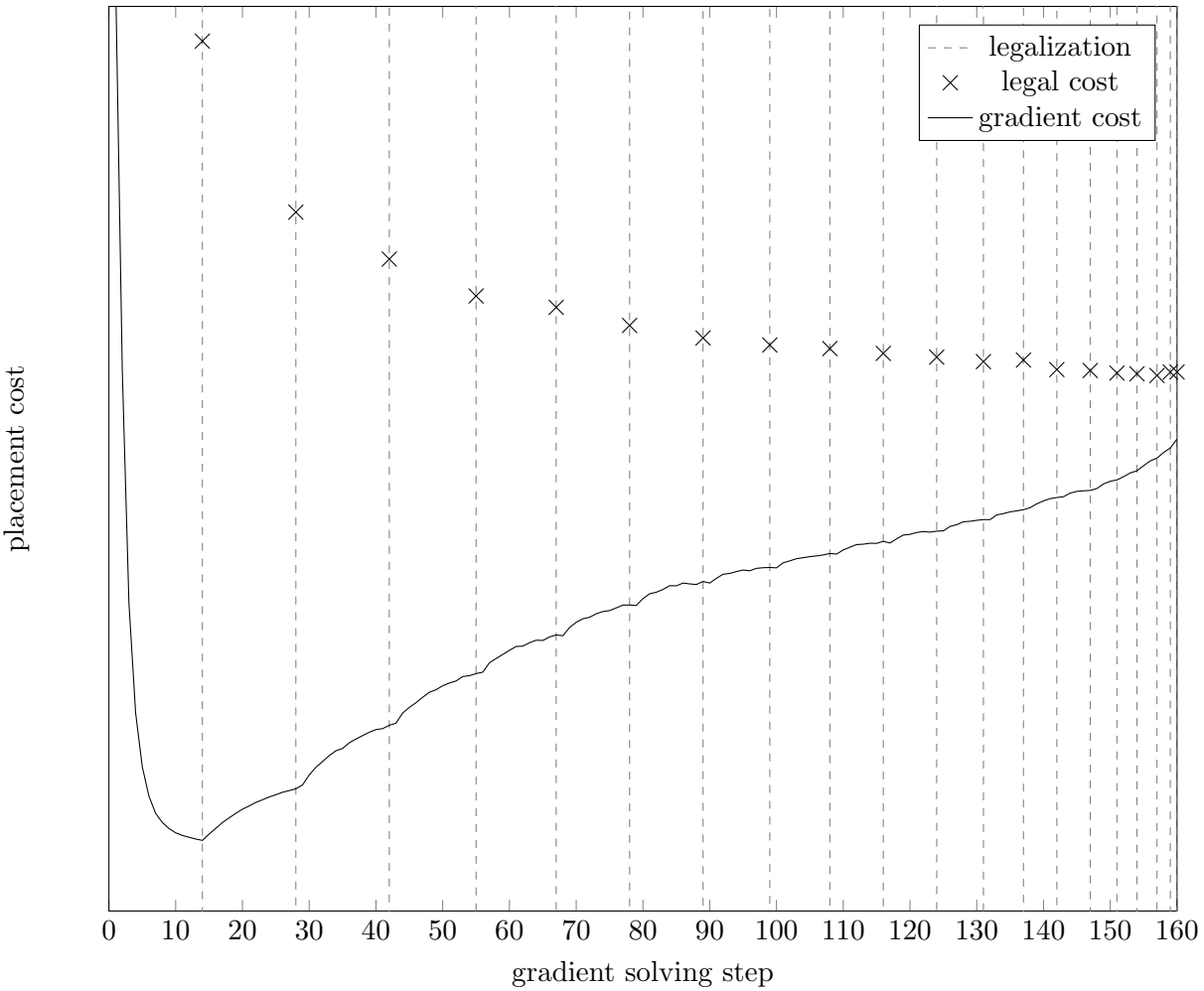


Figure 4.5: The placement cost over the course of a GDP execution.

4.5 Conclusion

In this chapter we have discussed the main contribution of this thesis in detail: a new placement heuristic based on the analytical placer TDAP, called gradient descent placement (GDP). The most important similarities and differences are:

- Both alternate between calculating an optimal but illegal solution and a calculating a legal but suboptimal solution
- The optimal solving step is replaced by an iterative procedure that moves all the clusters around every iteration
- The legal solving step is unaltered
- Both can be run in wirelength- or timing-driven mode

We have explained all the parameters that control the execution of GDP. In the next chapter we will optimize these parameters, and compare the performance of GDP, TDAP and simulated annealing.

Chapter 5

Methods and Results

To test our gradient descent placer (GDP) we have used and extended a placement framework that was developed at our department. In this chapter we first describe the capabilities of this tool. We then discuss the process of optimizing the GDP parameters. We finish with a comparison between the VPR placer, our own simulated annealing placer, our analytical placer and GDP.

A Note on Using VPR

In all experiments during and after development of our tool we use VPR 7.0, as found on GitHub [2]. For the experiments reported in this thesis we have used commit `8f73ad5`.

For some features our tool needs to call VPR. The used VPR binary should be compiled with the macro `PRINT_ARRAYS` defined in the file `place/timing_place_lookup.c:63`.

5.1 The Java Placement Framework

When starting the work for this thesis a placement framework written in the Java programming language was already available at our department. During the research for this thesis we have elaborated on this toolflow. We have among other things:

- Increased compatibility with the VPR toolflow
- Improved timing estimates
- Optimized the runtime and memory efficiency of the already available simulated annealing and analytical placement heuristics
- Implemented the new gradient descent placer (GDP) as described in chapter 4
- Increased usability and testability of the heuristics by providing the ability to control parameters from the command line

Most of the features reported below were developed by us during work for this thesis.

5.1.1 Overview

Our placement framework is compatible with the VPR toolflow, and replaces it's placement step completely. It takes as its input a net file as created by the VPR packer. A placement is generated using one of the three implemented heuristics: simulated annealing, analytical placement or gradient descent placement. All three heuristics have a wirelength-driven and a timing-driven mode, and can be controlled using command line parameters. It is possible to use more than one algorithm, e.g. to do a coarse placement using the analytical placer, followed by a low temperature simulated anneal. All implemented heuristics are deterministic, but different placements can be obtained by specifying a seed for the random number generators.

The output of the framework is a place file that is accepted by the VPR router as its input. This enables researchers to get post-routing statistics for placements. Prior to this thesis the exported place files were not at all compatible with VPR.

The framework is written in a highly object-oriented way. Most classes correspond to a real-world concept in FPGA devices or circuits, leading to self-documenting code. Implementing new placement algorithms is as easy as extending a single abstract class with two abstract methods and adding the new placer to an existing factory class.

The source code for the framework is freely available on GitHub [1]. Anyone interested can use it to carry out experiments with the existing placement heuristics or implement new heuristics.

5.1.2 Architecture Support

Prior to this thesis the tool only supported a single hardcoded architecture. This architecture contained CLBs with non-fracturable BLEs, and columns of multipliers and memory elements. The tool is now able to read in a VPR architecture description file at runtime. We support only a subset of the extensive architecture specification. Most notably the following information is ignored:

- Power consumption
- Routing network
- IO tiles that are not placed on the perimeter of the FPGA
- Logic primitives whose input-output delay is not equal for all port pairs of the same type (see the `delay_matrix`-tag in the VTR user manual [3])

Furthermore SDC-files, describing timing constraints, are ignored. Our tool uses the same timing constraints VPR does when no SDC-file is provided (see the VTR user manual [3]).

5.1.3 Wirelength Cost Estimation

The wirelength cost estimation of our tool for a placed circuit is very close to, but not exactly equal to VPR's post-placement cost estimation. This is due to an approximation in VPR. I/O-blocks that are on the perimeter should not overlap with any other FPGA blocks. For

code simplification VPR does overlap perimeter I/O-blocks with the perimeter logic blocks (see the comment in the VPR source code in `place/place.c:2145`).

In what follows the reported wirelength cost is always the cost calculated by our tool, even if the placement was generated by VPR.

5.1.4 Timing Cost Estimation

Our framework is not capable of estimating the delay of a connection between two tiles on the FPGA. Instead we rely on VPR for these estimations. When starting the placement of a circuit VPR is called to calculate the delay lookup tables for that circuit. These tables are then read by our placer and used throughout placement.

We have found an (unconfirmed) bug in VPR where the command-line critical path delay estimation is often not equal to the estimation found in the file `placement_crit_path.echo` [2]. The critical path delay estimate of our tool is always equal to the estimate found in the echo-file. In what follows we assume this estimate to be the correct one.

5.2 Simulation Environment

5.2.1 Benchmarks

Academic research in FPGA placement often only showcases results on small and homogeneous benchmarks, such as the MCNC benchmark suite. While these benchmarks can give a first indication about the performance of an algorithm, they are far too small to draw any conclusions about it's applicability to real-world circuits.

In our experiments we mainly used the Titan benchmark suite [16]. The 23 largest circuits in the suite range in size from 25,000 to 800,000 LUTs: this is the Titan23 suite. The other 23 benchmarks have between 3,500 and 50,000 LUTs per circuit. We will simply call these the *other benchmarks*. All experiments with Titan benchmarks use the architecture that is included with the benchmark suite. This architecture is modeled after the Altera Stratix IV device.

During development of the algorithm we initially used the homogeneous benchmarks included with VPR. The *other benchmarks* were used for testing while making the algorithm heterogeneous and timing-driven. When implementation of the placer had finished we used the largest eight circuits for parameter optimization and comparison with the other placement algorithms. Details for these eight circuits are shown in Table 5.1.

The Titan23 benchmarks were used to test the scalability and performance on large circuits for all the algorithms. Three benchmarks were excluded: gaussianblur, directrf and LU_Network. VPR needs more than 64 GB of RAM to generate the timing information for these circuits, which we do not have at our disposal. Details for the remaining 20 circuits are provided in Table 5.2.

All used benchmarks were delivered as .blif files. These contain unpacked netlists of LUTs, flipflops and blackboxes (which will be translated into hardblocks). The .blif files were converted to packed netlists (.net files) by the VPR packer using only the default parameters.

Name	# CLBs	# Hardblocks	# LUTs	# Flipflops
EKF-SLAM_Jacobians	2,869	55	49,425	5,223
CHERI	2,189	183	33,105	23,494
MCML	2,050	243	11,680	36,691
SURF_desc	2,188	77	39,579	5,856
random	1,855	63	30,770	5,370
jacobi	1,761	42	30,467	7,166
stap_steering	1,440	26	18,074	22,958
MMM	1,053	305	13,280	14,348

Table 5.1: Details for the eight largest of the *other benchmarks* included in the Titan benchmarks suite. The benchmarks are sorted by descending size (#CLBs + #Hardblocks).

Name	# CLBs	# Hardblocks	# LUTs	# Flipflops
sparcT1_chip2	30,895	533	377,740	430,976
bitcoin_miner	29,239	1,569	455,267	546,597
mes_noc	24,838	808	274,327	248,988
gsm_switch	19,138	1,715	159,400	296,681
LU230	15,433	5,101	209,000	293,177
denoise	17,393	415	322,031	8,811
stap_qrd	15,707	508	65,179	161,822
sparcT2_core	13,028	267	169,504	109,624
cholesky_bdti	9,627	732	76,465	173,385
segmentation	8,180	407	155,582	6,561
bitonic_mesh	6,827	1,753	109,637	49,570
minres	7,211	1,345	107,981	126,105
openCV	6,489	999	108,099	86,460
dart	6,599	530	103,804	87,386
SLAM_spheric	6,392	153	112,772	8,999
cholesky_mc	4,700	487	28,498	74,051
des90	3,958	908	62,875	30,244
sparcT1_core	3,750	132	41,974	45,013
neuron	3,169	278	24,765	61,477
stereo_vision	2,725	211	38,835	49,049

Table 5.2: Details for the 20 Titan23 benchmarks that were used in our experiments. The benchmarks are sorted by descending size (#CLBs + #Hardblocks).

Mode	Effort level	Runtime	Wirelength cost	Critical path delay
Wirelength-driven	0.05	0.78	0.93	
	1	0.58	0.99	
Timing-driven	0.05	0.98	0.98	0.99
	1	2.67	1.04	0.96

Table 5.3: The performance of our own simulated annealing placer for different settings, relative to the VPR placer. A value of 1 means equal performance, lower values are better.

5.2.2 Running the Experiments

The used architectures auto-size to the circuit under consideration. This means that the number of rows and columns of the simulated device will be chosen so that they are as small as possible while still ensuring that the device can contain all the circuit clusters.

All our algorithms start from a random initial placement. When comparing algorithms or parameter sets in our experiments we always start from the same initial placement. The location of IO-clusters is never altered during placement.

The HPWL cost and critical path delay for a placement are the pre-routing results unless mentioned otherwise. When displaying a mean value this is the geomean unless mentioned otherwise.

All experiments were run on a computer with 32 GB of RAM and a Intel Core i7-3770 processor clocked at 3.40 GHz with 8 MB of cache memory. Both VPR and or tool are not parallelized and run in a single thread.

5.2.3 Simulated Annealing

We have implemented our own simulated annealing placer that functions entirely the same as VPR's placer. In Table 5.3 we show the performance of our placer relative the VPR's placer. Results are shown for effort levels 1 (VPR default) and 0.05 in both wirelength- and timing-driven mode. For fair comparison we have excluded the building of the delay tables from VPR's runtime. This is an expensive operation that our placer doesn't perform (see section 5.1).

For all settings the QoR of our placer is comparable to VPR's, indicating that our placer was implemented correctly. We have built our placer from scratch without any old code slowing down execution. As a result we achieve a speedup of 42% at an effort level of 1 in wirelength-driven mode. In timing-driven mode our placer is very slow at this same effort level. The cause for this discrepancy is known: over 80% of the execution time is spent in a hot method that calculates a timing cost and requires many memory accesses. This method should and can be optimized but we have not had enough time to do this ourselves.

In what follows we will use our own simulated annealing placer in comparisons unless stated otherwise. This is because VPR requires more memory than our own placer. By using our own placer we can do comparisons for larger circuits.

5.2.4 TDAP

We have optimized the parameters of TDAP and found optimal values similar to those used in [14]: a starting pseudo weight of 0.5 and a stop ratio of 0.9. The pseudo weight multiplier is used to trade off runtime versus quality.

5.3 Optimizing the GDP Parameters

While developing the GDP algorithm we have introduced a number of parameters that don't have an obvious optimal value. Three parameters influence the course of the pseudo weight (see section 4.2.4):

- $w_{pseudo,start}$: the value of w_{pseudo} at the first solving step
- $w_{pseudo,stop}$: the value of w_{pseudo} at which the algorithm should stop executing
- $w_{pseudo,step}$: the value that is added to w_{pseudo} after each solving step

Two parameters determine the effort level of any given linear solving step using linear interpolation (see section 4.3.1):

- e_{first}: the effort level in the first linear solving step
- e_{last}: the fraction of the effort level in the last linear solving step

Three parameters directly influence the movement of clusters, as explained in section 4.2.1 and section 4.2.5:

- k: the spring stiffness
- d_{max}: determines the maximal spring force
- f_{av}: the speed averaging factor

The timing-driven variant of the algorithm adds four more parameters whose meaning can be found in section 4.2.3 and section 4.3.2:

- ϵ_c: the criticality exponent
- θ_c: the criticality threshold
- *tradeoff*: the trade-off between wirelength cost and critical path delay
- F_c: the fraction of solving steps that require a recalculation of the criticalities

These parameters are now optimized using the eight largest of the *other benchmarks*.

5.3.1 Optimizing Movement Parameters

There are eight parameters in total for the wirelength-driven algorithm. Benchmarking one set of parameters takes between two and ten minutes, so it is impractical to optimize all parameters

simultaneously. We can only optimize simultaneously for the parameters that are most directly influenced by each other.

k, d_{max} and f_{av} are the only parameters that don't influence the runtime of the algorithm, so we optimize these first. Through trial and error we choose values for the 5 other parameters that lead to a suboptimal but decent QoR: $w_{pseudo,start} = 0$, $w_{pseudo,step} = 0.01$, $w_{pseudo,stop} = 0.9$, $e_{first} = 40$ and $e_{last} = 1$.

k and d_{max} are directly related through Equation 4.2. We optimize these simultaneously by sweeping k over 14 values between 0.05 and 0.7 and d_{max} over 9 values between 10 and 50. For every k-d_{max}-pair we calculate the geomean of the wirelength cost of the placed benchmarks. We plot these on a heat map and identify the optimal k-d_{max}-pair: $k = 0.4$ and $d_{max} = 30$.

We sweep f_{av} independently between 0 and 0.98. The QoR is the highest for $0.1 < f_{av} < 0.4$ and the lowest for $fav > 0.7$. We choose $f_{av} = 0.2$ for further experiments.

5.3.2 Runtime-Quality Trade-Off

The five remaining wirelength-driven parameters all represent a trade-off between the runtime and QoR. We will try to shield these parameters from a circuit designer by allowing him/her to set the runtime-quality trade-off with only a single *effort* parameter. The runtime-quality pair of any placement obtained with this parameter should lie close to the runtime-quality Pareto front.

First we sweep $w_{pseudo,start}$ and $w_{pseudo,step}$ simultaneously. The data points with $w_{pseudo,start} = 0$ always lie on or very close to the runtime-quality Pareto front. We set the parameter to that value.

We then sweep over the four remaining parameters using the following value ranges:

- $w_{pseudo,stop}$: 5 values between 0.7 and 0.95
- $w_{pseudo,step}$: 5 values between 0.02 and 0.12
- e_{first}: 6 values between 6 and 30
- e_{last}: 4 values between 0.2 and 1

Again we summarize the QoR of a parameter set as a wirelength cost geomean. In the results we can see that $w_{pseudo,stop} = 0.85$ is a good value regardless of other parameter values. The same holds for $e_{last} = 0.07$.

For different regions on the Pareto front the optimal values of $w_{pseudo,step}$ and e_{first} vary greatly. We have found a relation between the two parameters that leads to near-optimal results:

$$w_{pseudo,step} = \frac{0.7}{e_{first}} \tag{5.1}$$

In the experiments that follow we only explicitly set the value of e_{first}. The value for $w_{pseudo,step}$ is calculated using the above formula.

5.3.3 Wirelength-Driven Parameters: Summary

We have now arrived at a set of parameters that give a good QoR:

- $k = 0.4$
- $d_{max} = 30$
- $f_{av} = 0.2$
- $w_{pseudo,start} = 0$
- $w_{pseudo,stop} = 0.85$
- $w_{pseudo,step} = \frac{0.7}{e_{first}}$
- $e_{last} = 0.07$

The parameter e_{first} should be set by the circuit designer and controls the runtime-quality trade-off. Good values for this parameter are discussed in the next sections.

5.3.4 Timing-Driven Parameters

There are five timing-driven parameters. Their meanings are explained in section 4.2.3 and section 4.3.2.

- ϵ_c: criticality exponent
- θ_c: criticality threshold
- *tradeoff*: the trade-off between optimizing wirelength and timing cost
- F_c: the criticality recalculation ratio
- P_c: the criticality recalculation priority

The *tradeoff* is a choice that is left to the circuit designer, so it should not be optimized right now. While optimizing the other parameters we set it to 300.

We first fix ϵ_c and θ_c to 1 and 0.8 respectively. These values are based on TDAP, and yield good results in informal tests. We sweep over F_c and P_c and find optimal results for $F_c = 0.4$ and $P_c = 1$. Note that one of our hypotheses from section 4.3.2 has proven to be false: spreading the criticality recalculations non-uniformly over the solving steps does not increase the QoR.

We simultaneously sweep over the two remaining parameters, and find the optimal values $\epsilon_c = 1$ and $\theta_c = 0.75$.

5.4 Comparison With Analytical Placement and Simulated Annealing

5.4.1 Wirelength-Driven

In GDP we can trade-off runtime against quality using the parameter e_{first}. In TDAP and simulated annealing we can do the same using the *anchor weight multiplier* and the *inner num* parameter respectively.

Other Benchmarks

We do a sweep over these trade-off parameters for the *other benchmarks* listed in Table 5.1. For every parameter we calculate the geomean of the wirelength cost of all the circuits. The results are plotted in Figure 5.1.

It is immediately obvious that simulated annealing produces better results than both GDP and TDAP. It is not a matter of giving GDP and TDAP more time: for GDP the quality stops improving above 2 s, for TDAP this happens at around 15 s.

This contradicts earlier research [5, 17] where analytical placers produced results that were competitive with VPR's for circuits that are comparable in size with the *other benchmarks*. Since the linear solving step of TDAP is straightforward, it is likely that our legalizer is of inferior quality in comparison with other analytical placers. Of course this also handicaps GDP which uses the same legalizer as TDAP.

When comparing GDP to TDAP, we see that both achieve similar rresults. At very low runtimes GDP performs better than TDAP: GDP can produce a usable placement in less time than either TDAP or simulated annealing. At higher runtimes TDAP yields slightly better results: the wirelength cost is around 1% lower than GDP.

At 10 s both GDP and TDAP have approximately reached their maximum achievable quality. At this point their wirelength cost is about 15% higher than SA's, but the SA cost keeps decreasing as the runtime increases.

Titan23 Benchmarks

Figure 5.2 shows the runtime versus quality of the three algorithms for the Titan23 benchmarks listed in Table 5.2. The graph is very similar to a scaled version of Figure 5.1. GDP and TDAP now keep improving slightly as the runtime increases but the gaps between the three placer's costs are approximately the same.

Conclusion

GDP performs comparable to TDAP. GDP can calculate a usable placement in less time than TDAP, but at higher runtimes the TDAP placement cost is about 1% lower than GDP's. For

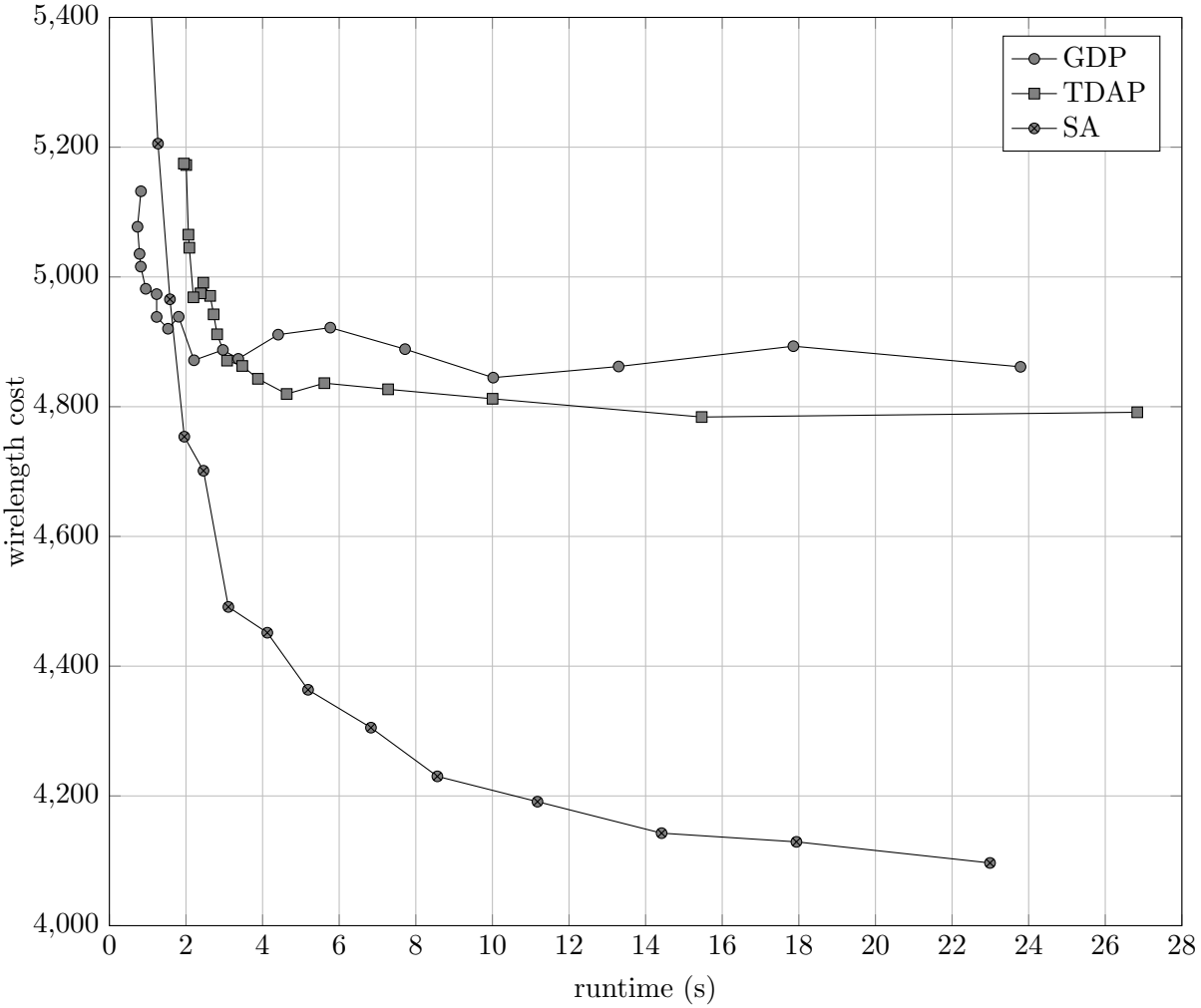


Figure 5.1: The trade-off between runtime and wirelength cost for GDP, TDAP and SA with the eight largest of the *other benchmarks*

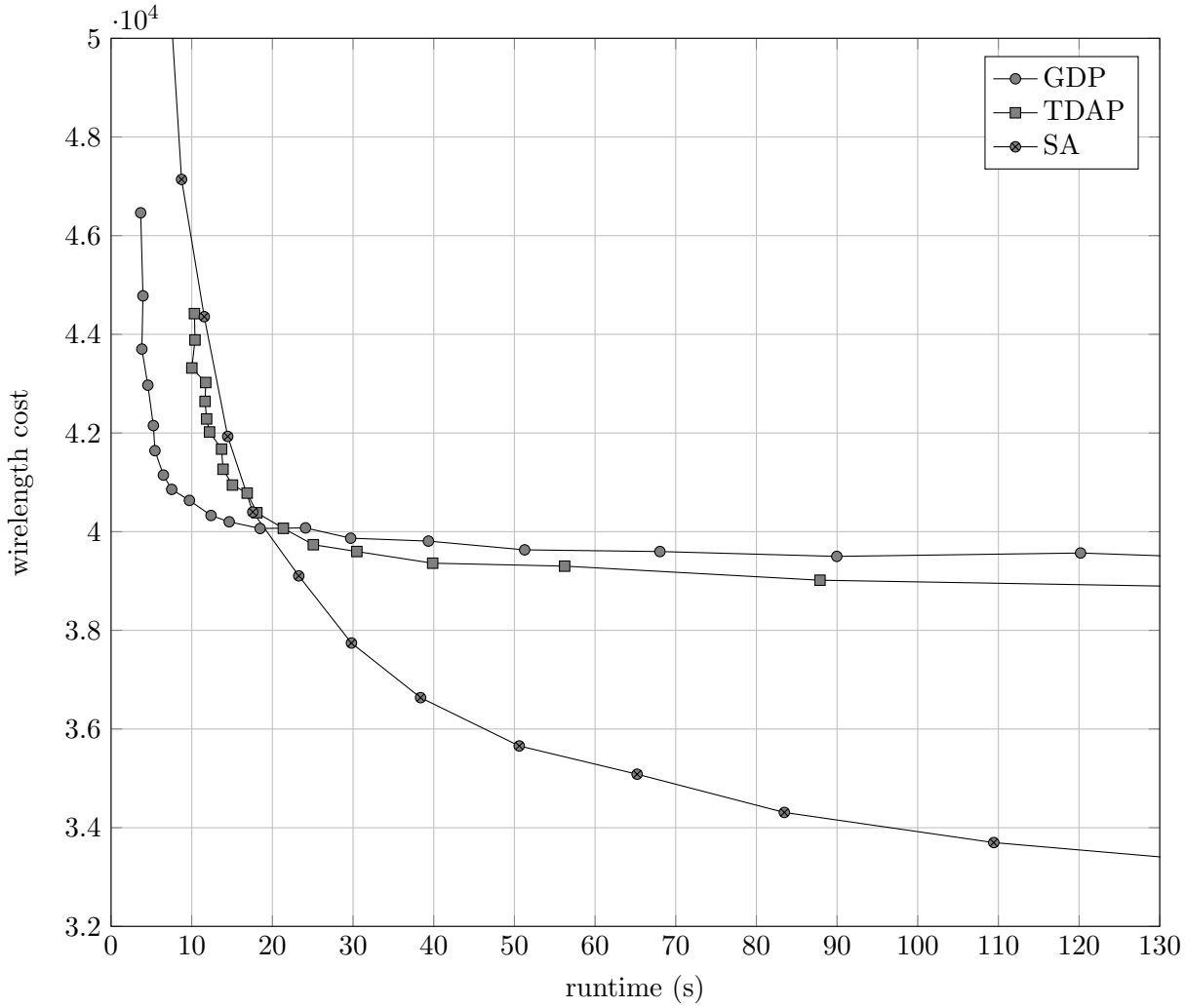


Figure 5.2: The trade-off between runtime and wirelength cost for GDP, TDAP and SA with the Titan23 benchmarks

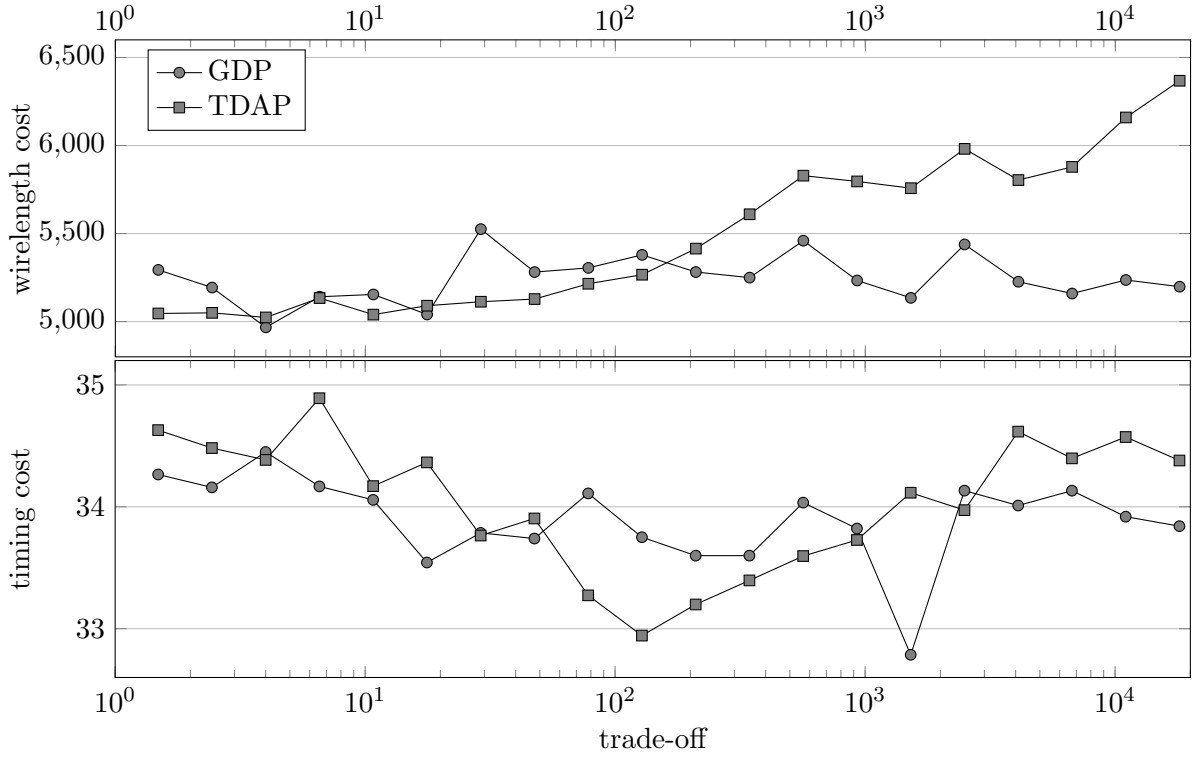


Figure 5.3: The trade-off between wirelength and timing cost for TDAP and GDP

the whole range of runtimes SA yields results that are better than GDP, except for big circuits and very low runtimes. In these cases GDP is faster, but the QoR deteriorates quickly as runtime decreases further.

5.4.2 Timing-Driven

For the timing cost we use the *max delay* metric, as explained in section 2.2.2. All the timing-driven placers have a parameter that controls the trade-off between wirelength and timing cost. For simulated annealing this parameter takes a value between 0 and 1; we use the default VPR value of 0.5.

The trade-off parameter in GDP has the same meaning as in TDAP. It takes a value between 0 and $+\infty$. We place the *other benchmarks* with TDAP and GDP with a different values for this parameter while keeping the runtime constant. The geomeans of the wirelength and timing cost are shown in Figure 5.3.

For TDAP the wirelength cost increases with the trade-off. The timing cost decreases as the trade-off increases up to 150 and increases after that. For GDP there the relation between trade-off and wirelength or timing cost is more ambiguous. We obtain a good quality for a trade-off of around 200. In further experiments we use a trade-off value of 150 for TDAP and 200 for GDP.

We now repeat the experiments from section 5.4.1 with the timing-driven placers. The results for the *other benchmarks* and the Titan23 benchmarks are shown in Figure 5.4 and Figure 5.5

respectively. As explained in section 5.2.3 our timing-driven simulated annealing placer is much slower than VPR's. For fair comparison we have added the VPR results in the *other benchmarks*-experiment. For the Titan23 benchmarks this is not possible because VPR requires too much memory.

Other Benchmarks

The timing-driven version of GDP achieves inferior critical-path delay compared to TDAP. At low runtimes the TDAP critical-path delay of TDAP placements is significantly lower than GDP's, while GDP achieves a slightly lower wirelength cost. For runtimes above 35 s this gap diminishes but it doesn't entirely close.

Compared to the wirelength versions there is now a bigger range of runtimes where both GDP and TDAP perform better than the simulated annealing algorithms. For runtimes below 30 s the critical path delay and wirelength cost achieved by GDP and TDAP are both better than simulated annealing. VPR starts to perform better at 30 s, our Java simulated annealer at around 60 s.

Titan23 Benchmarks

The timing-driven experiments with Titan23 benchmarks show the same trends as the *other benchmarks* in a more pronounced way. TDAP achieves a much lower critical path delay than GDP, and a slightly higher wirelength cost. For runtimes above 140 s the SA critical path delay is lower than GDP's.

The critical path delay obtained by TDAP is not matched by SA, even at a geomean runtime of 2000 s. A detailed discussion of the high effort performance of the different algorithms is provided in section 5.5.

Conclusion

The QoR of timing-driven GDP is low compared to TDAP and SA. The wirelength-timing trade-off tends too much to the wirelength side: the wirelength cost is lower than other algorithms but the timing cost is far higher. Our trade-off parameter is unable to change this trade-off sufficiently. Already in Figure 5.3 we could see that the influence of the trade-off is small and ambiguous. The results in Figure 5.4 and Figure 5.5 confirm that the influence of timing-driven connections on the gradient solving step is too small.

Minimizing the critical path delay using only timing-driven information typically gives bad results. This can be seen by setting the trade-off in SA to 1, or by excluding wirelength-driven connections from TDAP. The reason is that the problem of minimizing the critical path delay is a complex problem, where very similar solutions can have a very different quality.

For this reason timing-driven placers simultaneously minimize wirelength and timing cost. A solution with a low wirelength cost typically has a low critical path delay as well. The key to a good timing-driven placer is that it should minimize the wirelength cost as well. Only

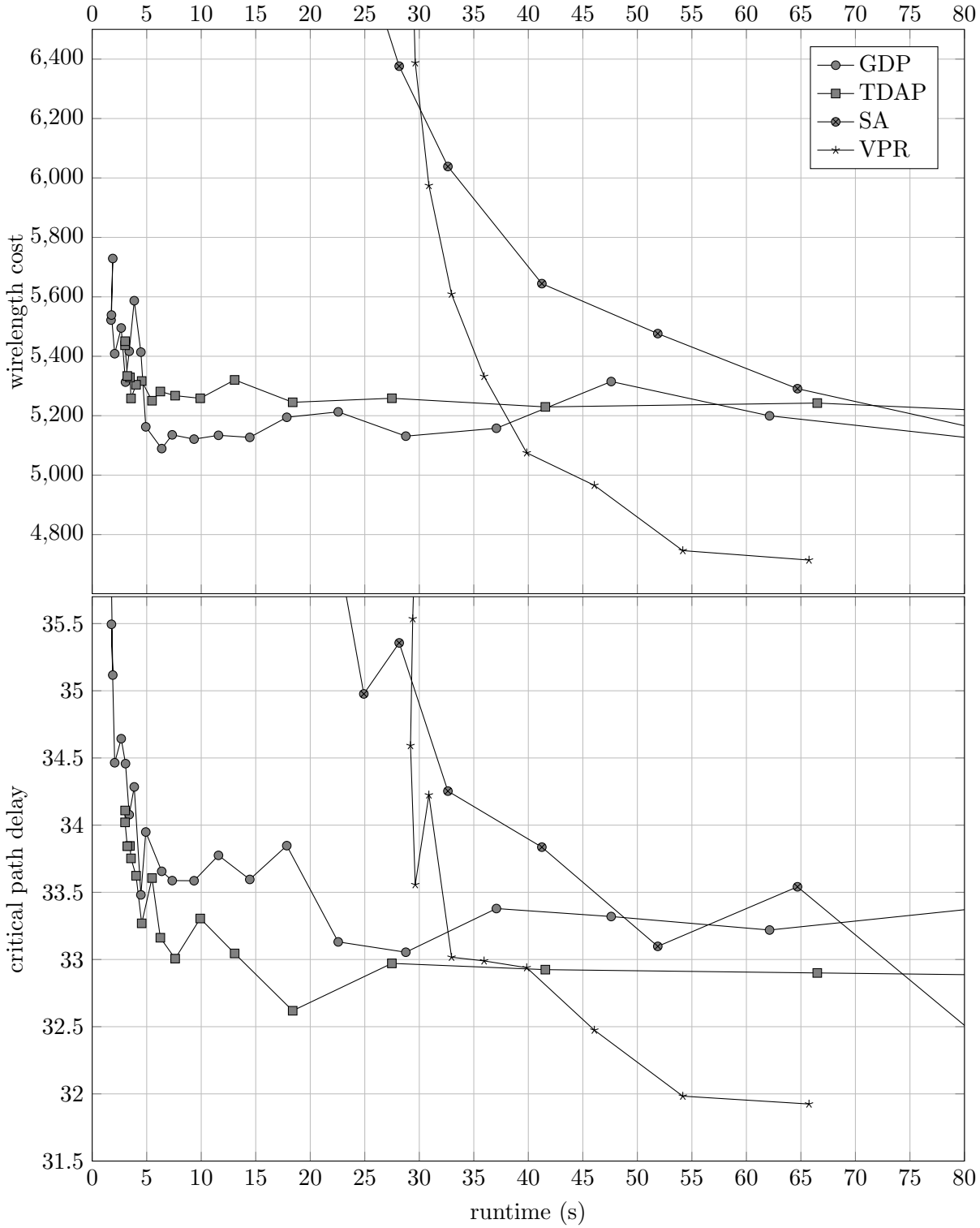


Figure 5.4: The trade-off between runtime and QoR for the timing-driven versions of GDP, TDAP and SA with the eight largest of the *other benchmarks*

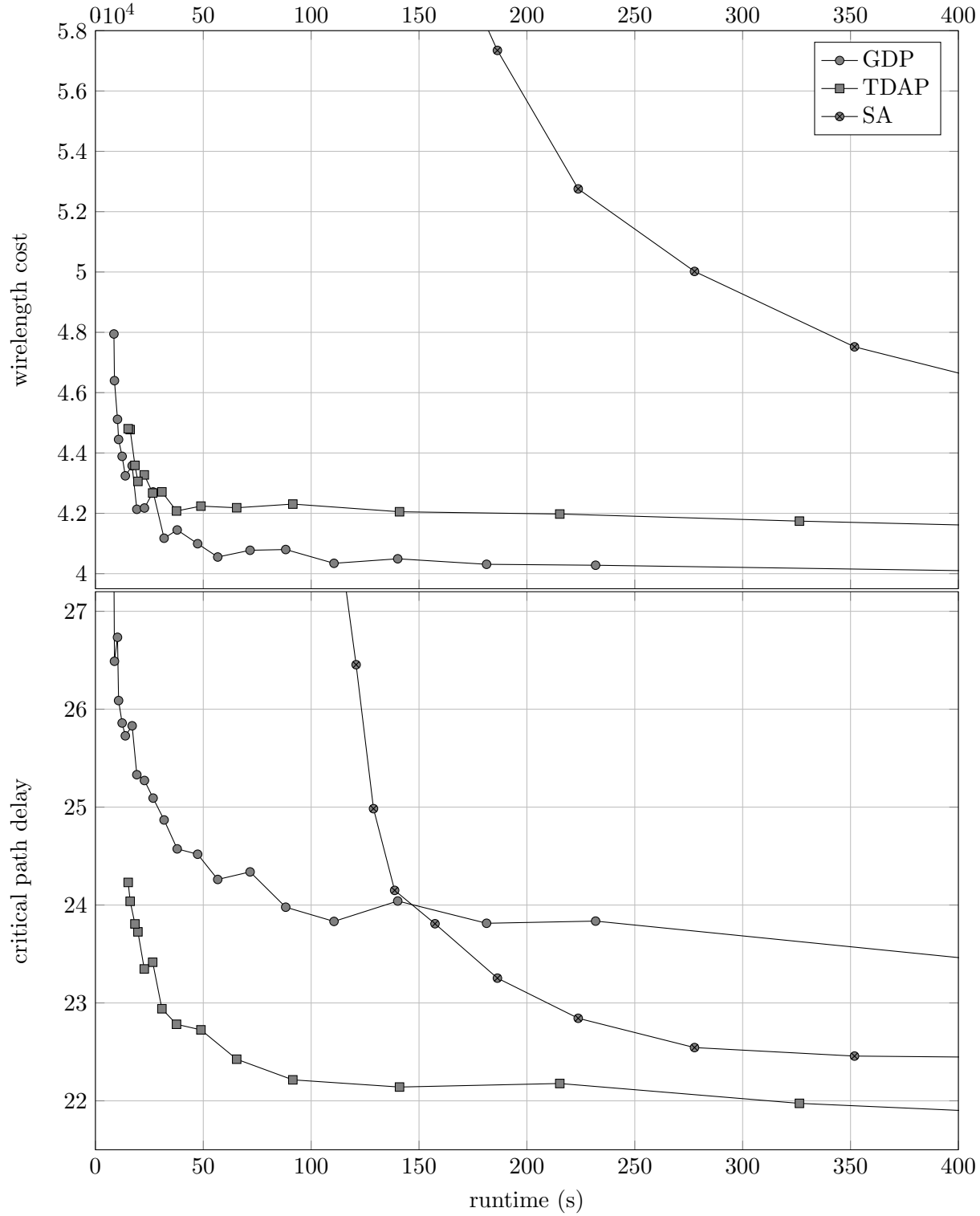


Figure 5.5: The trade-off between runtime and QoR for the timing-driven versions of GDP, TDAP and simulated annealing with the Titan23 benchmarks

for clusters on a near-critical path the timing information should override this wirelength information.

It is obvious that timing-driven GDP is currently not able to properly balance these two goals. The used method for including timing information (as explained in section 4.2.3) may be unsuitable. More powerful methods may exist, but we have not had the time to investigate this ourselves.

5.5 High Effort Performance

We have seen that at low runtimes GDP and TDAP perform better than simulated annealing in terms of both wirelength and timing cost. As runtime increases the QoR improves slowly. The SA QoR is bad at low runtimes but increases more quickly with the runtime. We now compare the performance of the three algorithms at high runtimes that are not shown in the previous graphs.

We place all the Titan23 benchmarks with SA, TDAP and GDP. We allow each placer to run for a long time, approaching it's maximal achievable QoR. We use the effort levels 1 for SA, 300 for GDP and an *anchor weight multiplier* of 1.007 for TDAP. The results are shown in Table 5.4 for wirelength-driven mode and Table 5.5 for timing-driven mode.

In wirelength-driven mode the QoR of GDP is slightly lower than TDAP: about 3%. The gap with SA however is large for both TDAP and GDP: on average SA achieves a 15% lower wirelength cost than TDAP.

In timing-driven mode TDAP achieves approximately the same critical path delay as SA with a 5% increase in wirelength cost. GDP achieves approximately the same wirelength cost as SA with a 7% increase in critical path delay. Since we are primarily interested in minimizing the critical path delay we summarize this table as follows: TDAP performs almost as good as SA; GDP has a rather large gap in critical path delay.

Circuit	WL cost		
	SA	TDAP (relative to SA)	GDP (relative to SA)
sparcT1_chip2	94,739	1.158	1.267
bitcoin_miner	182,460	1.192	1.097
mes_noc	68,341	1.232	1.196
gsm_switch	74,547	1.298	1.330
LU230	189,768	1.170	1.278
denoise	37,048	1.136	1.147
stap_qrd	31,236	1.140	1.231
sparcT2_core	40,625	1.158	1.216
cholesky_bdti	25,259	1.153	1.127
segmentation	18,828	1.097	1.073
bitonic_mesh	50,509	1.275	1.256
minres	34,846	1.160	1.379
openCV	37,029	1.186	1.193
dart	26,533	1.112	1.121
SLAM_spheric	18,005	1.121	1.121
cholesky_mc	9,209	1.222	1.221
des90	23,396	1.208	1.274
sparcT1_core	10,317	1.174	1.155
neuron	8,840	1.251	1.323
stereo_vision	7,940	1.121	1.214
Arithmetic mean		1.178	1.211
Average runtime	426 s	1.24	1.13

Table 5.4: Comparison of high quality wirelength-driven SA, TDAP and GDP for the Titan23 benchmarks. The benchmarks are sorted by descending size. *T cost* is the critical path delay, expressed in nanoseconds.

	SA		TDAP (relative to SA)		GDP (relative to SA)	
Circuit	WL cost	T cost	WL cost	T cost	WL cost	T cost
sparcT1_chip2	106,195	31.48	1.595	1.066	1.104	0.979
bitcoin_miner	238,959	12.03	0.947	0.994	0.847	1.361
mes_noc	82,255	14.63	1.106	0.970	0.990	1.183
gsm_switch	95,668	11.93	1.083	1.081	1.031	1.173
LU230	208,070	27.56	1.280	1.001	1.191	1.007
denoise	46,674	1,086.90	1.025	1.008	1.048	1.051
stap_qrd	39,680	8.41	0.953	1.099	0.969	1.250
sparcT2_core	46,492	11.19	0.988	1.040	1.019	1.064
cholesky_bdti	34,424	9.00	0.951	1.052	0.848	1.163
segmentation	22,519	1,071.83	1.039	1.010	1.119	1.039
bitonic_mesh	67,265	15.54	1.036	0.917	0.949	1.123
minres	40,493	8.64	1.035	1.015	1.147	1.137
openCV	49,962	15.23	0.947	0.721	0.900	0.782
dart	38,319	15.74	0.841	0.933	0.769	0.948
SLAM_spheric	20,250	100.11	0.988	0.990	1.042	0.984
cholesky_mc	12,182	8.46	0.974	0.960	0.919	0.989
des90	31,057	13.33	0.955	0.993	0.941	1.089
sparcT1_core	12,390	8.83	1.042	1.089	0.952	1.064
neuron	10,245	10.96	1.157	1.007	1.135	0.979
Arithmetic mean			1.050	0.997	0.996	1.072
Average runtime	3361 s		0.301		0.186	

Table 5.5: Comparison of high quality timing-driven SA, TDAP and GDP for the Titan23 benchmarks. The benchmarks are sorted by descending size. *T cost* is the critical path delay, expressed in nanoseconds.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

6.1.1 A Fast Placement Heuristic

The primary goal for this thesis was to develop a new, fast FPGA placer. We have partially achieved this goal, but more research is required before our placer could become a viable alternative for the existing heuristics, in particular the open-source simulated annealing-based placer included in VPR.

We have created a placement method that we have called gradient-descent placement (GDP). It is based on an analytical placer developed at our department called timing-driven analytical placement (TDAP). We have replaced the linear solving method from TDAP with a faster gradient-descent method. The legalization of the obtained optimal solution is identical to the TDAP legalization method.

In wirelength-driven mode our placer achieves a $1-3\%$ decrease in quality for the same runtime. We consider this a good result, taking into account the fact that this is only a first attempt at creating a gradient-descent based placer. We believe better results are achievable by further optimizing the mechanics described in section 4.2.

The timing-driven version of our placer performs inadequately. The achieved critical path delays are around 8% worse than TDAP's. On the other hand the achieved wirelength costs are lower, but this is of secondary importance.

6.1.2 An Open-Source Placement Framework

Prior to this thesis the only open-source framework available for FPGA placement research was VPR. It is written in C, non-object oriented and contains large amounts of code that is hard to read, maintain or extend. Academics interested in researching FPGA placement heuristics needed to build on this difficult code or develop their own framework. Both options involve a high threshold.

By building on work carried out at our department before this thesis we have created a new placement framework that lowers this threshold significantly. Some of the framework's key assets are:

- Easy to extend. Written in Java in a highly object-oriented way, featuring self-documenting and easy to understand code.
- Highly compatible with the VTR toolflow: our placer can be chained between the VPR packer and router without any further configuration.
- Fast: Our wirelength-driven simulated annealer is 40% faster than VPR's.

The framework source code is publicly accessible and freely modifiable. It can be found on GitHub [1].

6.2 Future Work

6.2.1 Architecture Support

Our placement tool already supports a large subset of the VPR architecture specification (see section 5.1.2). Some essential information is still ignored: non-perimeter IO tiles, `delay_matrix`-tags and non-default timing constraints. Developing a more comprehensive architecture support is useful to increase compatibility between our tool and VPR.

6.2.2 Better Legalization for TDAP and GDP

In section 5.4.1 we have seen that the wirelength-driven version of TDAP performs poorly in comparison with SA. This contradicts earlier research on analytical placers, in particular HeAP [5]. We believe this is a consequence of an insufficiently powerful legalizer.

The legalizer used in TDAP and GDP was implemented by us. Very little sources are available that discuss the inner workings of legalizers, especially for heterogeneous FPGA architectures. As a result many of the legalizer details were researched and developed by ourselves, with too little time for optimizations and investigation to alternatives.

We are confident that the current legalizer implementation leaves much room for improvements. More research in this field would benefit both TDAP and GDP.

6.2.3 More Powerful Timing-Driven Version

The wirelength-driven version of GDP is approximately on par with TDAP. The timing-driven version however focuses too much on minimizing wirelength cost. Our experiments show that the critical path delay of GDP placements is on average about 8% higher than TDAP placements. The parameters that we added to control GDP's behavior are incapable to move the focus more to minimizing critical path delay.

The way timing-information is incorporated into GDP using timing-driven springs is explained in section 4.2.3. It may be impossible to achieve good QoR using this method. Other methods to incorporate the timing-driven information should be investigated.

6.2.4 Parallelization

We have developed the GDP gradient solving step with possible future parallelization in mind. We believe it is possible to implement the current implementation on a general purpose GPU (GPGPU) with little modifications. We expect that high speedups are possible using GPU parallelization.

Once the data initialization is done, all circuit and placement information is stored in arrays of integers and floating point numbers. The only required operations are simple arithmetics. Each net is stored in a separate array, so all nets can be processed in parallel. During the processing of a net, all net clusters require an update. This update is a short blocking operation: memory read, summation and memory write to the same location. So blocking in the algorithm could only occur when two nets are being processed that contain common clusters.

Our legalizer requires a significant portion of the algorithm runtime, and it is not ready for parallelization. In [17] however a legalizer similar to ours was implemented on a GPU. This could serve as an inspiration for a parallel implementation of our gradient solver.

Bibliography

[1] The Java Placement Framework. URL `https://github.com/EliasVansteenkiste/FPGA-Placement-Framework`. Accessed: 2016-05-29.

[2] Issue 116 on the vtr bugtracker. URL `https://github.com/verilog-to-routing/vtr-verilog-to-routing/issues/116`. Accessed: 2016-05-29.

[3] VTR user manual. URL `https://vtr.readthedocs.io/en/latest`. Accessed: 2016-05-29.

[4] Alexander Choong, Rami Beidas, and Jianwen Zhu. Parallelizing simulated annealing-based placement using GPGPU. In *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, pages 31–34. IEEE, 2010.

[5] Marcel Gort and Jason H Anderson. Analytical placement for heterogeneous FPGAs. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 143–150. IEEE, 2012.

[6] Subodh Gupta, Jason Helge Anderson, Linda Farragher, and Qiang Wang. CAD techniques for power optimization in Virtex-5 FPGAs. In *CICC*, pages 85–88. Citeseer, 2007.

[7] Eddie Hung. Mind the (synthesis) gap: Examining where academic FPGA tools lag behind industry. In *Field Programmable Logic and Applications (FPL), 2015 25th International Conference on*, pages 1–4. IEEE, 2015.

[8] Myung-Chul Kim, Dong-Jin Lee, and Igor L Markov. SimPL: An effective placement algorithm. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 31(1):50–60, 2012.

[9] Jürgen M Kleinhans, Georg Sigl, Frank M Johannes, and Kurt J Antreich. GORDIAN: VLSI placement by quadratic programming and slicing optimization. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 10(3):356–365, 1991.

[10] Tzu-Hen Lin, Pritha Banerjee, and Yao-Wen Chang. An efficient and effective analytical placer for FPGAs. In *Proceedings of the 50th Annual Design Automation Conference*, page 10. ACM, 2013.

[11] Adrian Ludwin, Vaughn Betz, and Ketan Padalia. High-quality, deterministic parallel placement for FPGAs on commodity hardware. In *Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, pages 14–23. ACM, 2008.

[12] Jason Luu, Jeffrey Goeders, Michael Wainberg, Andrew Somerville, Thien Yu, Konstantin Nasartschuk, Miad Nasr, Sen Wang, Tim Liu, Nooruddin Ahmed, et al. VTR 7.0: Next generation architecture and CAD system for FPGAs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 7(2):6, 2014.

[13] 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*, pages 203–213. ACM, 2000.

[14] Arno Messiaen. A new less memory intensive net model for timing driven analytical placement. Master's thesis, Ghent University, 2015.

[15] Kevin Morris. Xilinx vs. Altera calling the action in the greatest semiconductor rivalry, 2014. URL `https://web.archive.org/web/20150725111721/http://www.eejournal.com/archives/articles/20140225-rivalry`. Accessed: 2016-02-03.

[16] Kevin E Murray, Scott Whitty, Siyuan Liu, Jason Luu, and Vaughn Betz. Titan: Enabling large and complex benchmarks in academic CAD. In *23rd International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8. IEEE, 2013.

[17] R. Pattison, C. Fobel, G. Grewal, and S. Areibi. Scalable analytic placement for FPGA on GPGPU. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6, Dec 2015. doi: 10.1109/ReConFig.2015.7393356.

[18] J. Rose, W. Klebsch, and J. Wolf. Temperature measurement and equilibrium dynamics of simulated annealing placements. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 9(3):253–259, Mar 1990. ISSN 0278-0070. doi: 10.1109/43.46801.

[19] Carl Sechen and Kai-Win Lee. An improved simulated annealing algorithm for row-based placement. In *Proc. IEEE International Conference on Computer Aided Design*, pages 478–481, 1987.

[20] N. Venuopal and Manimegalai R. Wirelength driven placement for FPGA using soft computing technique. In *Soft-Computing and Networks Security (ICSNS), 2015 International Conference on*, pages 1–5, Feb 2015. doi: 10.1109/ICSNS.2015.7292389.

[21] N. Viswanathan and C. C. N. Chu. FastPlace: efficient analytical placement using cell shifting, iterative local refinement,and a hybrid net model. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(5):722–733, May 2005. ISSN 0278-0070. doi: 10.1109/TCAD.2005.846365.

[22] M Xu, Gary Gréwal, and Shawki Areibi. StarPlace: A new analytic method for FPGA placement. *INTEGRATION, the VLSI journal*, 44(3):192–204, 2011.

[23] Yonghong Xu and Mohammed AS Khalid. QPF: efficient quadratic placement for FPGAs. In *International Conference on Field Programmable Logic and Applications, 2005.*, pages 555–558. IEEE, 2005.