Ultra-Fast Automatic Placement for FPGAs

by

Yaska Sankar

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

Ultra-Fast Automatic Placement for FPGAs

Master of Applied Science, 1999
Yaska Sankar
Department of Electrical and Computer Engineering
University of Toronto

The demand for high-speed Field-Programmable Gate Array (FPGA) compilation tools has escalated for three reasons: first, as FPGA device capacity has grown, the computation time devoted to placement and routing of circuits has grown more dramatically than the available computer power. Second, there exists a subset of users who are willing to accept a reduction in the quality of result (using a larger FPGA or more resources on a given FPGA) in exchange for a high-speed compilation. Third, high-speed compile has been a long-standing desire of users of FPGA-based custom computing machines, since their compile time requirements are ideally closer to those of regular computers.

This thesis focuses on the placement phase of the compile process, and presents an ultra-fast placement algorithm for FPGAs. The algorithm is based on a combination of multiple-level, bottom-up clustering and hierarchical simulated annealing.

It provides superior area results over a known high-quality placement tool on a set of large benchmark circuits, when both are restricted to a short run time. For example, in 10 seconds of placement time on a 300 MHz Sun UltraSPARC, the ultra-fast tool realizes an average wirelength improvement of 30% compared to the high-quality tool. It can also generate a placement for a 100,000-gate circuit in 10 seconds that is only 31% worse than a high-quality placement that takes 524 seconds using a pure simulated annealing implementation. For this circuit, the ultra-fast tool achieves this level of placement quality 5 times faster than the high-quality tool.

In addition, when operating in its fastest mode, the ultra-fast placement tool can provide an accurate estimate of the wirelength achievable with good quality placement (within 6%, on average). This can be used, in conjunction with a routing predictor, to very quickly determine the routability of a given circuit on a given FPGA device.

Acknowledgments

I am profoundly grateful to my advisor, Jonathan Rose, for allowing me to draw upon his vast technical expertise, his infectious zeal, and his unwavering confidence in my abilities as a researcher. I am indebted to him for setting high standards for his students in both technical achievement and communication, and for going beyond the duties of a supervisor to act as a teacher, mentor, promoter, motivator, counselor, and loyal supporter. I value his technical lessons, patience, encouragement, sage advice, and above all, enthusiastic guidance during my time here.

I have also greatly benefited from the collective wisdom of the members of Professor Rose's research group: Vaughn Betz, Jordan Swartz, Mohammed Khalid, Steve Wilton, Mike Hutton, Sandy Marquardt, Rob McCready and Paul Leventis. I am particularly grateful to Vaughn not only for faithfully providing the infrastructure and support of the software used to house this work, but also for the time, advice, and guidance he offered throughout. Jordan and Vaughn merit special thanks for their helpful and insightful comments on an earlier draft of this thesis.

My time here would have been far less educational and enjoyable without the antics of my many friends and colleagues in LP392 and SF2206: the other founding members of the infamous Package Deal (Jason "Virtual Beverage Engineer" Podaima, Vaughn "Mr. Optimal" Betz, Jordan "Human Torch" Swartz), Sandy "Formatting Monkey" Marquardt, Qiang "Be A Man" Wang, Jason A., Andy, Marcus, Ali, Warren, Mazen, Vincent, Mark, Khalid, Jeff, Dave, Javad, Rob, Ken, Wai, Nirmal, Paul, Guy, Dan, Alex, Derek, Rob, Gia, Sush, Keith, Marcel, Jane, Kevin, Bob, Aris, John, Sudarsan, Duncan, and the Electric Fielders softball teams with whom I had the distinct pleasure of playing, to name a few. They are wholly responsible for provoking stimulating discussions and providing spirited diversions, and I thank them for it.

I consider myself fortunate to know a few kind souls whose friendship, support, encouragement, advice, and willingness to listen made things bearable during the difficult times and delightful the rest of the time: Steve, John, Heather, Mike VDP, and Nav.

Whatever success I have achieved here I owe to the steadfast support, encouragement, patience, faith and love of my parents and my brother.

Finally, this work would not have been possible without the financial support from the Natural Sciences and Engineering Research Council, the University of Toronto, Communications and Information Technology Ontario, Lucent Technologies Inc., Xilinx Corporation, and the Information Technology Research Centre of Ontario.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Field-programmable gate arrays (FPGAs) have been highly successful because they can realize any digital circuit simply by the specification of the millions of bits that control a sea of programmable logic and interconnect. The ability of an FPGA to be re-programmable within a system offers designers the ability to implement different circuits, fix errors in circuits, or add new features to existing circuits in a matter of seconds. This programmability gives FPGAs significant advantages over customized application-specific integrated circuits (ASICs): flexibility, quick time-to-market, zero non-recurring engineering costs, and easier debugging. All these benefits do come at a price, however, in that circuits realized on an FPGA typically occupy at least ten times the area and operate at least three times slower than their ASIC counterparts [Brow92].

A set of automated computer-aided design (CAD) tools is necessary in order to generate an FPGA programming bitstream that implements a desired circuit. These software tools first take the circuit description (where the circuit is specified in a hardware description language such as VHDL, or in schematic form) through the synthesis stage, where the circuit is represented as a netlist of technology-mapped logic blocks and connections. Following this, the CAD tools perform the placement and routing steps. During placement, the logic blocks implementing the circuit are assigned to physical locations on the FPGA such that the wiring area is minimized. In routing, the point-to-point connections are made by specifying which physical switches are acti-

vated in the programmable wiring such that full routability is achieved and circuit speed is maximized. This can be used to generate the programming bitstream. The complete set of tasks the CAD tools perform to translate a circuit description into a programming bitstream is known as FPGA compilation.

One key advantage of FPGAs over mask-programmable gate arrays and ASICs is that they provide quick turnaround times for circuit designers between the conception of the circuit and its implementation on a chip. However, this rapid prototyping advantage has been reduced as the capacities of these programmable devices grow. While current CAD algorithms provide high-quality solutions, they require a great deal of CPU time, and the compilation times for large circuits are growing more rapidly than the available computer power. This adversely impacts: hardware designers, who must wait longer to map their designs to FPGAs; logic emulation system users, who must compile hundreds of FPGAs at a time [Quic98]; and FPGA-based custom computing machine users, who desire compilation times similar to those of a microprocessor.

Place-and-route times for large FPGAs at present (those with approximately 5000 lookup-table (LUT) / flip-flop pairs and higher) can last many hours on a modern processor, and there is no guarantee of successful completion. For example, an 8383 LUT circuit (approximately 100,000 gates) requires almost 1.2 hours for placement and routing using the Xilinx M1 CAD tools (version 4.1.2) [Xili98] on a 300 MHz Sun UltraSPARC workstation [Swar98b]. For a subset of designers, these prohibitively long compile times may nullify any gains that had been realized by using FPGAs in the first place. Since the time-to-market advantage and ability to create rapid prototypes are severely compromised, some users may opt to return to the world of mask-programmed gate arrays (MPGAs) or standard cells. With million-gate FPGAs on the horizon, it is imperative to design FPGA compilation tools that will scale well with device sizes, and provide an acceptable trade-off between quality[1] and compile time [Rose97]. Only then will the characteristic FPGA benefits of fast design and manufacturing cycles be maintained. We contend that there are users who are willing to sacrifice circuit quality for speed of compilation.

1. We define quality as the wirelength required by the circuit or the speed at which the circuit can operate when mapped to the FPGA. Greater wirelength will require the use of a larger FPGA or the use of more resources on a given FPGA than is otherwise necessary.

These trends provide a compelling motive to explore methods for fast compilation for FPGAs. In this thesis, we shall focus on the placement phase of the FPGA compile process and present an ultra-fast placement tool that aims to minimize area [Sank99]. Although a fast timing-driven placement tool should also seek to minimize circuit delay, we believe that area-based minimization is a prudent first step. Furthermore, while top-down partitioning, floorplanning, macro-based placement, and incremental placement methods may be cited as alternate approaches to mitigating the long compile times of the next generation of large FPGAs, we contend that there will always be a need for a fast, flat placement tool.

1.2 Goals and Scenarios

There are two main objectives of our fast placement tool: 1) to provide placements very quickly with the minimum amount of degradation in circuit quality, and 2) to provide very fast predictions as feedback to the user based on the fast placements generated.

In order to precisely articulate what we mean by fast placement, we have set the following goal for our placement tool: be able to perform a full placement of a 100,000-gate circuit in 10 seconds on a modern CPU. We believe this goal is justifiable: extremely fast placement is essential if it is to be used as a guide within upstream CAD tools, or in emulation systems and reconfigurable computing applications, or to satisfy impatient hardware designers. More importantly, we wish to ensure that the running time varies linearly with the size of circuit, and has a small proportionality constant. This is crucial because as long as our algorithms are of linear complexity or close to that, the same technology advancements that permit FPGA devices to become more dense also increase processor speeds. Therefore, our algorithm will scale well.

A key element of our fast placement tool is that we offer a tunable “knob” that allows the user to smoothly trade quality for compile time. So, not only do we aim to provide more area-efficient placements in very short run times, we also aim to provide high-quality placements given longer run times. We further expect that FPGA users will benefit from being able to run a quick placement of their circuit, rather than wait for a complete high-quality placement, to ascertain what size FPGA device to purchase to implement their design.

In addition, since some placement problems may be extremely difficult (the circuit barely fits onto the device, and the CAD tools need a great deal of time to generate a usable placement) or impossible (the circuit cannot fit onto the device), we believe it is important to quickly supply the user with the predicted area versus compile time trade-off for a circuit of similar size.

We envision three scenarios in which fast compile would be used, once a user has designed a circuit and targeted an FPGA of a specific size:

1. If the user explicitly states a compile time restriction, then the fast CAD tools should estimate how much extra space -- or how much larger a device -- will be necessary for the design to be placed in the desired running time, and produce the placement.
2. Alternatively, if the user explicitly states that the design must fit into the desired FPGA, then the fast tools can inform the user of one of the following: that the circuit can be placed and routed quickly (and provide the placement and routing files), that the circuit will be placed and routed given more time, or that the task is impossible. The work by Swartz et al. offers a method for making the "fit/no-fit" prediction *given a placement* and its total wirelength [Swar98a].
3. The user is supplied with an area versus compile-time trade-off curve and selects the point appropriate to his goals. In this case, there must be sufficient free space in the FPGA. Those users willing to sacrifice circuit area for a faster compile time, via the tunable "knob", can accommodate the increased area in several ways: they can reduce the complexity of a single design by partitioning the circuit onto multiple FPGAs, or can select an FPGA with greater logic capacity. They can also eliminate part of the circuit by reducing the amount of parallelism in the hardware.

Our second goal is to leverage our fast placement tool to provide fast feedback to the user in a number of areas. We provide, for a given compile time restriction, an estimate of how much extra area the circuit will require with that much time devoted to placement. Conversely, for a given area restriction, we provide an estimate of the shortest amount of time needed to produce a placement that will fit. Furthermore, we furnish, in that short amount of time, an estimate of what the wirelength would be if we allowed the placement tool to run without a compile time restriction and try to attain maximum quality. In so doing, we can quickly supply to Swartz' prediction tool [Swar98a] a rough idea of what the final best placement will be, which can then be used to inform the user quickly and reliably of whether or not the circuit will fit onto the targeted FPGA. This fast estimation of high-quality wirelength, as well as the ultra-fast placement, may be exploited by the CAD tools that precede the placement stage to quickly gauge what the circuit will look like after placement. These are all useful features, since some FPGA users bemoan the fact that many industry FPGA CAD tools do not provide adequate predictability with respect to area and speed.

1.3 Research Approach

Our experimental research methodology involves first examining an existing academic FPGA CAD tool, VPR [Betz97], which is known to provide high-quality placement and routing solutions over a large suite of benchmark circuits in a reasonable amount of time. Having determined how well VPR performs over a series of different run times for a set of large circuits from a variety of sources, we create a fast placement algorithm that can be incorporated into VPR's infrastructure. In this way, we can make a fair evaluation of how well our new placement tool performs with respect to both run time and area compared to an existing tool on the same platform, with the same set of large benchmark circuits and the same physical FPGA architecture. We then examine the influence of the various enhancements to the placement algorithm that make it "ultra-fast", and make some empirical observations. We also offer some insight into the fast prediction of high-quality wirelength and its accuracy, given a fast placement, and how this is valuable feedback to the user. Finally, we explore simple methods to predict the area versus compile time relationship of a circuit before it is placed, and then invoke the ultra-fast placement algorithm with appropriate automatically generated parameters to meet either a compile time or an area restriction.

1.4 Thesis Organization

This thesis is organized as follows: Chapter 2 presents some of the previous work done in developing and applying VLSI placement algorithms for FPGAs, although little of this work is primarily targeted towards our stated goal of high-speed compilation. We also discuss some of the prior work conducted in the related area of clustering, as well as the recent work accomplished in algorithms and tools targeted for fast FPGA compilation. In Chapter 3, we describe the details of our ultra-fast placement algorithm. In Chapter 4, we offer a variety of results obtained from running our tool on a suite of large benchmark circuits, using a simple and general FPGA architecture. We provide a direct comparison between our tool and a known high-quality placement tool with respect to run time and area, and present results of our fast wirelength prediction scheme. Finally, Chapter 5 highlights some of the key conclusions of this work and proposes directions for future research in this area.

Chapter 2

Background and Previous Work

In this chapter, a precise definition of the placement problem for FPGAs is provided, followed by a brief description of some of the relevant previous work in this area. This background material is divided into sections covering general VLSI placement algorithms, the use of clustering algorithms to reduce problem complexity, and algorithms specifically designed to address the issue of fast compile for FPGAs.

2.1 Definition of FPGA Placement Problem

In the physical layout stage of circuit synthesis, placement follows high-level design, technology-independent logic optimization, and technology mapping to a set of basic blocks, but precedes the routing stage where actual interconnections are made between blocks. The basic placement problem for FPGAs begins with a technology-mapped netlist of logic blocks[1], input and output (I/O) pads, and their interconnections. The result of placement is an assignment of the blocks and pads to specific physical locations of the FPGA that minimizes a specific cost function [Brow92]. A logic block is the basic unit of an FPGA that performs a specified logic function. A netlist is a hypergraph representation of a circuit, where each vertex represents a circuit element (block), and each hyperedge, or net, represents a wire that connects a set of blocks together. A pad is an I/O block that is the physical interface between the circuit and the outside world. Placement

1. For this thesis, a logic block will be one 4-input lookup table (4-LUT) and one D flip-flop.

refers to the mapping of the circuit elements in the netlist onto the circuit elements in the physical architecture of the target device [Leng90]. In the specific case of FPGA placement, it is a mapping of blocks and pads in the circuit netlist to the blocks and pads arranged on the physical FPGA array, as shown in Figure 2.1.

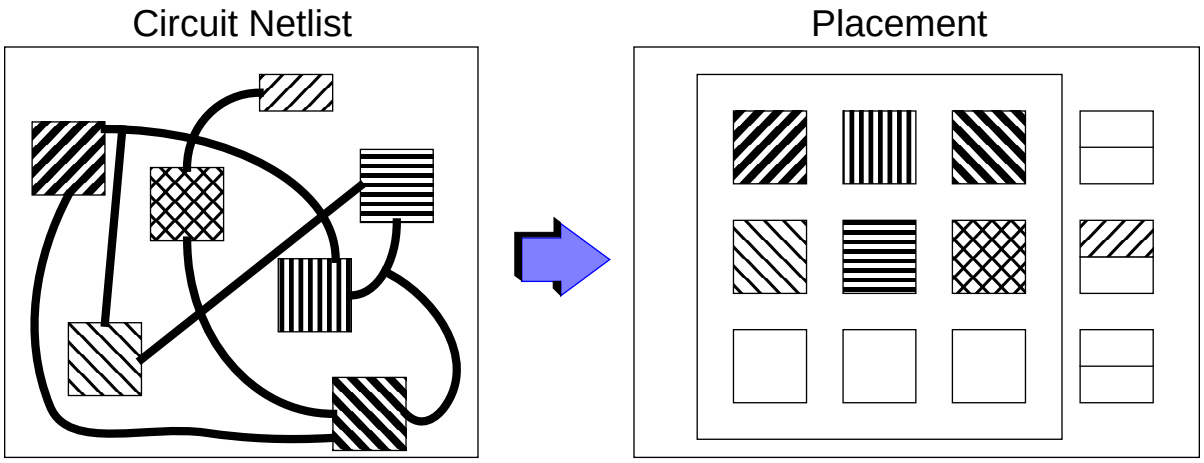

Figure 2.1: Simple example of FPGA placement.

More formally, the FPGA placement problem can be expressed as [Leng90]:

- Given: a hypergraph $G = (V, E)$ representing the circuit, where V is the set of vertices (blocks), and E is the set of edges (nets), with edge costs $w(e) \in R_+$ for each $e \in E$; $|V| = n$; an FPGA grid of size $r \times s$, where $r, s \in N$, and $r \cdot s \geq n$.
- Find: all placements -- mappings $p : V \rightarrow [1, r] \times [1, s]$ of blocks to block locations on the FPGA grid.
- Minimize: a cost function $c(p)$.

Since we have stated our ultra-fast placement goal to be to provide an area-efficient placement very quickly, we will attempt to minimize the total wirelength (length of routing wire) required to map the circuit to the FPGA. This is because the cost of the device is proportional to the amount of silicon required to implement it. We can minimize the amount of silicon required (and thus the device cost) by minimizing the area required to wire the circuit components together. Since this total wiring area is only known after the routing stage, an effective estimator is needed at the preceding placement stage, and estimated total wirelength has been shown to be

suitable [Leng90]. Other cost functions that have been used as placement quality metrics include circuit delay and wiring density. The basic placement problem is known to be NP-hard, and therefore many heuristics have been employed [Leng90]. The following section describes some of these heuristics.

2.2 Placement Algorithms

Surveys of VLSI placement algorithms are offered in [Shah91] and [Hana72], which describe three main varieties that are currently most popular: 1) Min-cut, or partitioning-based placement algorithms [Dunl85] [Huan97]; 2) Analytical placement algorithms that use quadratic programming [Klei91] [Sigl91] [Alpe97a] [Alpe97b], some of which incorporate iterative improvement [Doll91]; 3) Simulated-annealing-based placement algorithms [Kirk83] [Sech85] [Sech88] [Sun95] [Betz97] [Sarr97]. Previous simulated-annealing-based placement tools have achieved similar or higher quality solutions compared to the other types of placement algorithms, though in some cases, with longer execution times. Consequently, our ultra-fast placement algorithm is based on simulated annealing, and its performance is measured against another known high-quality simulated-annealing implementation. For this reason, this section focuses on simulated-annealing-based placement algorithms.

2.2.1 TimberWolf

TimberWolf [Sech85] [Sech88] is an integrated set of placement and routing tools that provided the first simulated-annealing-based placement algorithms targeting standard cells, macro/custom cells, and gate arrays. The basic simulated annealing algorithm proposed in [Kirk83] was adapted to explore a number of different placement configurations stochastically to minimize a cost function that estimates overall wiring area. Figure 2.2 shows the pseudo-code for the simulated annealing algorithm, and [Sech85] contains a detailed description of the basic algorithm and the various cost functions used for the different types of placement problems.

The central idea of the algorithm is the notion that the exploration of numerous placement configurations is guided by a parameter, T (temperature), that determines the probability of whether configurations that reduce the quality of the placement will be accepted in the process of searching through different placements. This temperature value is gradually reduced as the search space is explored. Given a random initial placement, a source module is chosen randomly (either a cell or an I/O pad). Then, a target location is chosen at random for this module such that it lies

```
X = Initial_Random_Placement();
T = Set_Initial_Temperature();  /* T=T0 */
Dlimit = Set_Initial_Range_Limit();  /* Dlimit = whole chip */
while (Exit_Criterion() == false) {  /* annealing not done yet */
        while (Inner_Loop_Criterion() == false) { /* work per temperature not done yet */
                Xnew = Generate_Move(X, Dlimit);
                /* returns a new configuration generated incrementally from previous one */
                /* by random pairwise exchange or translation within range limit */
                ΔC = Cost(Xnew) - Cost(X);  /* calculate change in cost */
                r = Get_Random_Number(0,1);
                /* r = random number uniformly distributed between 0 and 1 */
                if (r < e -ΔC/T)
                        X = Xnew;  /* update current placement */
                /* always accept move (p=1) if it improves placement (ΔC < 0) */
                /* accept "bad" moves (ΔC > 0) with probability p = e -ΔC/T */
                /* when T is large, all bad moves likely to be accepted, */
                /* when T is small, only bad moves with small ΔC likely to be accepted */
        }  /* end inner loop */
        /* exploration at current temperature complete */
        T = Update_Temperature(α, T);  /* T = αT */
        Dlimit = Update_Range_Limit(Dlimit);
}  /* end outer loop */
/* annealing complete, X = final placement solution */
```

Figure 2.2: Pseudo-code for a basic simulated annealing-based placement algorithm. [Sech85] [Betz98]

within the displacement range specified by a range limit mechanism, and the target can house the same type of module. If that target location is occupied, then the target module is swapped with the source module (pairwise interchange) and the cost of the resulting placement is evaluated. If the target location is originally empty, then the cost of the new placement with only the source module displaced to the target location (single block translation) is evaluated. In either case, if the new cost is less than the cost of the previous undisturbed placement, the move is accepted. If the new cost is more, then the move is only accepted with probability $e^{-\Delta C/T}$, where ΔC is the change in placement cost due to the move or swap, and T is the current temperature. A large value of T is used at the beginning, meaning that almost all moves, irrespective of cost, are accepted. As the placement quality improves with the accumulation of moves, the temperature is gradually

reduced, making it less likely that moves that degrade the placement will be accepted. Eventually, the value of T is so low that only moves which improve the placement quality are accepted, making the heuristic greedy at that point. The parameter T is what permits probabilistic hill-climbing to take place and helps the placement solution avoid being caught in local minima.

The rate at which the temperature is reduced (called the temperature update factor, α), the number of configurations to explore at each temperature (known as the inner loop criterion, or *InnerNum*), the exit criterion by which the annealing algorithm terminates, and the behaviour of the range limiting mechanism are all crucial details that are specified by an *annealing schedule*. In TimberWolf, the value of α starts at 0.8, is gradually increased to 0.95, and gradually decreased back to 0.8 over the course of the entire anneal. This is to ensure that for the portions of the anneal where the cost function is decreasing rapidly, the configuration space is explored more slowly and thoroughly. The number of moves generated per temperature is set to 20-100 times the number of modules in the circuit. The range limit mechanism that sets bounds on the displacement of a module during a move or swap is adjusted so that it is the entire chip at the outset, and as T decreases, so does the window of permissible target locations. Finally, the annealer terminates when the cost function over the last three temperatures is found to be unchanging.

The original cost function used for standard cell placement consists of three components: 1) the total estimated wirelength (W), which is computed as the sum over all nets of the half-perimeter of the bounding box that encompasses all the pins on each net; 2) a penalty function (P_O) for any overlap between cells; 3) a penalty function (P_R) for row length mismatches, which ensures that the lengths of the rows of cells do not vary considerably from each other. The cost function can be expressed as:

$$C = W + \mu P_O + \lambda P_R \tag{2.1}$$

The two penalty components (and their scaling factors, μ and λ) do not arise for FPGAs since all logic blocks are of equal size and shape, and logic blocks are only allowed to swap with other logic blocks, and I/O pads are allowed to swap only with other pads.

2.2.2 Hierarchical Clustering and Annealing

An innovative hierarchical clustering and placement algorithm is proposed in [Sun95] and is incorporated into an updated version of TimberWolf (TimberWolfSC v7.0). As an improvement to the previous cost function in TimberWolf, the penalty functions are eliminated, and only the total wirelength term remains. If a move potentially violates a row length limit, it is discarded. If a move is accepted, cells in the affected rows are shifted to prevent any cell overlap. Thus, every placement generated is a feasible one. The revised wirelength-based cost function is an incremental one keeping track of the change in placement cost (ΔC), and has two components: 1) the change in net lengths (ΔW) for those nets connected to the cell or cells that were moved or swapped; 2) the change in net lengths (ΔW_S) for those nets connected to cells in the affected rows that need to be shifted as a result of the move or swap. The former component, ΔW, is computed exactly, while the latter, ΔW_S, is estimated, but in both cases the computation is fast. The revised cost function can be written as:

$$\Delta C = \Delta W + \Delta W_S \tag{2.2}$$

The hierarchical placement methodology consists of clustering and simulated-annealing phases, and it proceeds as follows: first, the original, large, flat netlist is condensed using two levels of clustering, the details of which will be covered in Section 2.3.2. The purpose of this clustering is to reduce the complexity of the circuit so that it is easier to place. It tries to group those cells that will eventually be close to each other in the final placement, and collapses as many flat nets as possible into the clusters while keeping the size of the clusters the same.

Following this bottom-up clustering, the above simulated annealing algorithm is employed to do a top-down placement of the various levels of netlists. The two resulting clustered netlists are subjected to a 3-stage annealing schedule: in the first stage, a high temperature anneal is performed on the top-level netlist of clusters for the first 50% of the total number of moves that are attempted. After the top-level clusters are decomposed into first-level clusters, each first-level cluster is randomly placed within the boundaries laid out by the top-level cluster in which it was contained. Then, the next lower level (first level) of clusters are annealed from 50% to 70% of the total number of moves. Upon decomposing the first-level clusters and placing the original flat

cells within the first-level cluster boundaries, the final annealing stage is conducted, occupying the final 30% of all the moves. In each of the latter two stages, the cells or clusters are permitted to move across the cluster boundaries specified from the previous higher level, and the initial temperature at each stage is computed as:

$$T_0 = \frac{-\overline{\Delta W}}{\log(R_{accept})} \tag{2.3}$$

where $\overline{\Delta W}$ is the average change in wirelength and R_{accept} is the desired ratio of accepted moves to attempted moves. Over the whole placement process, all timing requirements (restrictions on circuit delay) are satisfied as well.

The combination of hierarchical clustering and annealing serves to speed up the entire placement process, with run times that are between 3.6 and 7.5 times faster than those obtained using the previous version of the tool, TimberWolfSC v6.0. The average reduction in wirelength between TimberWolfSC v7.0 and TimberWolfSC v6.0 is 12%. When compared to the quadratic placement tool Gordian/Domino, the placements produced by TimberWolfSC v7.0 have 8% less wirelength on average, and require between 3% and 26% less run time on circuits with more than 5000 cells.

In [Roy93], hierarchical clustering and annealing are utilized to perform FPGA placement. A clustering and annealing-based, timing-driven, *N*-way chip partitioner is used as a global placement tool for a single-chip FPGA (*N*=1) with emphasis on both wirelength and execution time. A bottom-up hierarchical clustering is used to merge those modules that form “natural” clusters (dense subgraphs, in the accumulative weighted graph used to represent the circuit netlist). Then, the clusters are refined through an adaptive technique where clusters are merged so that small nets (2-pin nets, typically) are collapsed and the fanouts of large nets are reduced. This means the total number of nets in the clustered netlist is reduced and the average fanout of the remaining inter-cluster nets is reduced. These both assist in speeding up the annealing-based partitioning/placement.

Once the physical chip is partitioned into *N* sub-chips, the core is divided into a grid of bins, to make the wirelength calculation more accurate. The clusters of modules are already built, so the simulated-annealing-based *N*-way partitioner is invoked. The clusters are moved from bin to bin as the annealer progresses, and the location of a cluster is the center of the bin in which it cur-

rently resides. The cost function consists of total weighted wirelength, *W*, (where wirelength is estimated using the sum across all nets of the half-perimeter bounding box for each net), and a penalty, P_t, for timing violations (sum of all the penalties over all critical paths specified). The cost function can be written as:

$$C = W + P_t \tag{2.4}$$

For large nets, an incremental net-span updating scheme is used, and since the granularity of the grid is a single bin, and the algorithm operates on clusters of blocks, the updating is simpler and faster than with the flat netlist. Moves are generated as mentioned previously, with the added restriction that roughly the same utilization in each bin (amount of logic clusters per bin) needs to be maintained. Once the annealing-based partitioning phase is complete, the clusters within each of the *N* partitions are decomposed, and detailed placement of the constituent flat modules is performed using a low temperature flat anneal. Compared to an industrial placement tool, this technique reduced the total number of unrouted nets that remain after routing by 90%.

2.2.3 VPR - Versatile Place and Route

In [Betz97] [Betz98], a dynamic adaptive annealing schedule that leads to high-quality placements for FPGA circuits in a reasonable amount of run time is described. It includes some of the features from the work done on annealing schedules by Huang et al. [Huan86], Lam and Delosme [Lam88], and Swartz and Sechen [Swar90], but it also implements a novel temperature update scheme and stopping criterion. The annealing schedule parameters are adjusted automatically depending upon the size of the circuit, and a bounding box wirelength cost function is used with correction factors for multi-terminal nets [Chen94].

The initial temperature is set to 20 times the standard deviation in cost after a set of N_{blocks} moves are made, where N_{blocks} is the total number of logic blocks and pads in the circuit. Since the initial placement is a random assignment of logic blocks and pads to the physical array, this ensures that the temperature is high enough that almost all initial moves are accepted. At each temperature, $InnerNum \cdot N_{blocks}^{4/3}$ moves are attempted, where the scaling factor *InnerNum* has a default value of 10. The temperature is reduced in such a way that if there is little change in cost either due to the acceptance rate being too low (high quality placement already) or too high (poor placement quality), the temperature is reduced by a larger fraction. So, as long as the cost is

changing significantly and a substantial number of moves, but not all, are being accepted, the temperature is reduced more gently, so that this space of placement configurations is explored more thoroughly. Table 2.1 shows how the temperature update factor, α, is automatically determined according to what the acceptance rate of moves was at the last temperature stage.

Table 2.1: Automatic temperature update schedule for VPR. [Betz97]

Fraction of Moves Accepted (R_{accept})	Temperature Update Factor (α)
$R_{accept} > 0.96$	0.5
$0.8 < R_{accept} \leq 0.96$	0.9
$0.15 < R_{accept} \leq 0.8$	0.95
$R_{accept} \leq 0.15$	0.8

A range limiting mechanism is used to maintain a target acceptance rate of 44%. If the acceptance rate falls below 44%, the range within which candidates for pairwise swaps are found is shrunk. Conversely, if the acceptance rate grows beyond 44%, the range is expanded. This is accomplished using the following relationship between the new range limit (D_{new_limit}), the previous range limit (D_{old_limit}), and the previous acceptance ratio, R_{accept} :

$$D_{new\_limit} = D_{old\_limit} \cdot (1 - 0.44 + R_{accept}) \tag{2.5}$$

Note that if the acceptance ratio is exactly 44%, there is no change in the range limit, and that D_{new_limit} is restricted to the range [1, maximum FPGA dimension]. This range limit then gradually shrinks over the course of the anneal from covering the whole chip at the beginning, when the acceptance ratio is typically very high, down to 1 (nearest neighbors) at the end of the anneal when only local refinement is tolerated.

Finally, the annealer terminates when the temperature falls below a certain fraction of the average cost per net. If there are N_{nets} nets in the circuit, and the average placement cost over all the moves at the current temperature is *Cost*, the annealer exit criterion, T_f, can then be expressed as:

$$T_f < \frac{0.005 \cdot Cost}{N_{nets}} \tag{2.6}$$

VPR placement parameters that can be specified by the user from the command line include: the initial temperature (T_0), the exit temperature (T_f), the temperature update factor (α), the scaling factor (*InnerNum*) for the number of moves to make at each temperature, and the initial random seed (as long as the same seed for the random number generator is used, the placement algorithm is deterministic). These allow the user to tune the placement tool to achieve different quality versus run time trade-offs for a particular circuit.

VPR also addresses the long execution time of typical simulated annealing implementations by performing fast incremental bounding box updates when evaluating the cost of a placement after a move or swap. For each net, a data structure contains not only the coordinates of the four sides of the net bounding box, it also contains the number of pins on the net that lie on each side. This information is used to determine the new net bounding box after a swap by only examining the pins that moved rather than a brute force calculation for every pin on the affected net.

VPR currently holds the world record among academic FPGA placement and routing tools with the minimum total number of tracks required to place and route a set of standard benchmark circuits [Betz97b].

2.2.4 Algorithms Based on Variations of Hierarchical Clustering and Simulated Annealing - NRG and Simulated Quenching

In [Sarr97], the NRG standard cell, row-based placement tool is proposed. It performs in succession a global placement, a detailed placement, and a final refinement, each of which is based on simulated annealing. The main objectives are to reduce the search space of potential placement configurations to obtain higher quality results faster, and to provide a fast prediction of the high-quality placement possible using NRG itself. The placement problem is divided into a global placement phase and a detailed placement phase. The motivation is that a good global placement can assign modules to approximate locations quickly and the detailed placement concerns itself with the exact location and timing of a module, given the constraints set by the global placement.

In the global placement phase, flat modules are assigned to a coarse grid via annealing, where each grid location (“bin”) can hold multiple modules. The cost function being minimized includes total bounding box wirelength ($P_{WIRELENGTH}$) and a penalty function (P_{GBCD}) to prevent unbalanced numbers of modules in each bin on the global placement grid. So, the cost of a placement *x* is given by:

$$f(x) = P_{WIRELENGTH} + \lambda P_{GBCD} \quad (2.7)$$

where λ is a scaling factor. The authors argue that this is different from clustering modules in the flat netlist first, even though the goal of problem simplification is the same. They reason that operating on the flat netlist (rather than a netlist of indivisible clusters) provides more flexibility, and that clustering modules takes a local view of the placement problem rather than a global one.

In the detailed placement phase, the global bin assignments from the annealed result of the global placement phase are decomposed into the flat grid. Then, a low temperature annealing schedule is used to perform the detailed placement of the modules, minimizing a cost function consisting of wirelength, overlap penalty, and row penalty. The follow-up refinement phase shifts cells to remove any remaining overlap.

The grid size for the global placement phase is arrived at using a binary search to minimize the difference in wirelength between a very fast global placement and a very fast detailed placement for that grid size. Each phase is sped up by simply reducing the number of moves attempted by the annealer at each temperature. NRG achieves wirelengths that are as good or better than those obtained by the commercial version of TimberWolf for a set of five benchmark circuits. In addition, NRG can be used to quickly obtain an estimate of the wirelength NRG itself can provide when allowed to achieve the highest quality placement. NRG's wirelength predictions are 3 - 20% more than the actual wirelengths from high-quality NRG placements. Finally, the run time of NRG can reduced by up to a factor of two, if a quality degradation of 1 - 3% can be tolerated.

In [Sato97], an iterative, partitioning-based placement algorithm known as "simulated quenching" (SQ) is proposed for linear (1-dimensional) placement problems. Many portions of the algorithm mimic ideas from simulated annealing; the algorithm operates in the following manner:

- "Moves" are generated by partitioning the linear placement of blocks into subgroups using a particular pitch (subgroup size) and a randomly chosen offset;
- A force value for each block in each subgroup is computed based only on the nets that cross subgroups; a force value represents the direction in which moving a block decreases the length of an inter-subgroup net connected to that block. These accumulated force values are then used to sort the blocks within each subgroup.
- This process is repeated multiple times with different partitionings of the netlist, each with the same pitch, but different offsets.

- After a number of iterations, the pitch value (subgroup size, p) is reduced according to a predetermined "schedule", $p = p - 0.03 * (p / \log_2 p)$ (this is similar to temperature reduction in simulated annealing), and the entire procedure is repeated until the pitch value falls below 2.

When this method is compared to a pure simulated annealing implementation, similar wirelength results are obtained for a set of very small MCNC benchmark circuits [Yang91], but SQ exhibits superior run time. However, for the MCNC circuit s38417, the SQ algorithm requires 8 hours to reach a stable solution when run on a 166 MHz Hypersparc workstation.

2.2.5 Choice of Starting Temperature for Simulated Annealing

One crucial feature of any automatically-generated, dynamic, and adaptive annealing schedule for a variety of circuits is the choice of the starting temperature, T_0, for a given placement. The reason is that if the temperature is set too high, subsequent annealing will destroy the existing placement structure, which makes any previous work toward placing the circuit useless. Conversely, if the temperature is set too low, the annealer is unlikely to improve upon the existing placement significantly, as it will be unable to escape local minima.

In [Rose90], a method is proposed to compute a good starting temperature for simulated annealing placements. The idea is that there exists a temperature for a given simulated annealing placement where the placement is in a state of equilibrium. In this state, there is no expected net change in the cost function after a set of moves, which implies that the expected change in placement cost is zero:

$$E(\Delta C) = 0 \tag{2.8}$$

Let $P(\Delta C)$ be the probability that a move with change in cost ΔC is generated from the current placement state. If this distribution is known, then if N is the number of moves attempted on the current placement, we can express Equation (2.8) as:

$$E(\Delta C) = \int_{-\infty}^{\infty} (\Delta C \cdot P(\Delta C) \cdot P_{accept}(\Delta C))\, d\Delta C = 0 \tag{2.9}$$

where $P_{accept}(\Delta C)$ is the probability that a move with cost ΔC is accepted. From Section 2.2.1, we know $P_{accept}(\Delta C)$ is commonly defined as:

$$P_{accept}(\Delta C) = \begin{cases} e^{(-\Delta C)/T}, & \Delta C > 0 \\ 1, & \Delta C \leq 0 \end{cases} \tag{2.10}$$

By substituting Equation (2.10) into Equation (2.9), two separate integrals are obtained:

$$E(\Delta C) = \int_{-\infty}^{0} (\Delta C \cdot P(\Delta C))\, d\Delta C + \int_{0}^{\infty} (\Delta C \cdot P(\Delta C) \cdot e^{(-\Delta C)/T})\, d\Delta C = 0 \tag{2.11}$$

So, the equilibrium temperature of a given simulated annealing placement with a known distribution $P(\Delta C)$ is the temperature $T = T_{eq}$ for which Equation (2.11) is satisfied.

The process of calculating this equilibrium temperature for a given placement is informally referred to as a "simulated thermometer." In order to implement this temperature measurement scheme to compute the equilibrium temperature for a given placement, a set of N discrete samples is used to approximate the continuous distribution of ΔC over all moves. This means Equation (2.9) can be written as:

$$E(\Delta C) = \sum_{i=1}^{N} (\Delta C_i \cdot P(\Delta C_i) \cdot P_{accept}(\Delta C_i)) = 0 \tag{2.12}$$

Ideally, $P(\Delta C_i)$ is measured on a running simulated annealing process while at the equilibrium temperature, T_{eq} . However, if each move i is actually being generated, then the number of moves produced with a change in cost ΔC_i will be $N \cdot P(\Delta C_i)$. That means moves with a particular change in cost will be generated with about the same frequency as they would appear in the continuous distribution. So, as long as N is sufficiently large, the set of sample moves generated will approximate the distribution of ΔC over all possible moves, and the $P(\Delta C_i)$ term in Equation (2.12) is inherent from the move generation and in the calculation of $E(\Delta C)$.

The simulated thermometer takes an initial placement of blocks, and performs a large number of moves (N), none of which are permitted to change the placement. The change in cost associated with each move i is recorded (ΔC_i), as well as whether the move would have been accepted or not, based on Equation (2.10). The expression for $E(\Delta C)$ in Equation (2.12) is then evaluated, and a binary search over temperature is performed to find the temperature at which the expected value of the overall change in placement cost is zero. When the temperature value, T_{eq}, that satisfies Equation (2.12) is found, the temperature at which the given placement is in a state of equilibrium is determined, and this is a suitable initial temperature to begin annealing the placement.

As the binary search for the equilibrium temperature progresses, only the probability of accepting “bad” moves is affected (moves that increase the placement cost -- $P_{accept}(\Delta C_i)$ for $\Delta C_i>0$). So, the only portions of Equation (2.12) that need to be recomputed are those for each recorded “bad” move at a particular temperature during the search. This simplifies and speeds up the calculation. It is also important to ensure that enough moves are made to obtain an accurate probability distribution for $P(\Delta C_i)$. Between 10,000 and 100,000 moves are recommended [Rose90].

2.3 Clustering Algorithms

In this section we discuss the motivation behind using netlist clustering algorithms to speed up heuristics that solve problems such as placement and partitioning, and list some of the cost functions used in the prior research to build good clusters.

2.3.1 Using Clustering to Reduce Problem Complexity

Whether the problem is partitioning or placement, the virtues of using bottom-up netlist clustering are well documented in [Sun95] [Roy93] [Shin93] [Hage97] [Kary97] [Alpe97c]. The primary goal of this clustering is to reduce the problem size so that a smaller and more easily solvable problem is obtained. This assists in decreasing the time required for iterative algorithms to obtain a good solution for the overall problem.

A clustering groups netlist modules into disjoint subsets, known as clusters. When the modules are packed into each cluster, the netlist that is induced is a condensed version of the original problem that may be solved more easily and quickly. This is particularly crucial for iterative algorithms whose performance tend to degrade as the problem size and complexity increase. Hagen and Kahng suggest that the advantage offered by clustering in reducing the problem size permits the algorithm operating on the condensed problem to focus on the most difficult and time-consuming portion [Hage97]. Both [Sun95] and [Roy93] state that through effective netlist clustering, the number of clusters to manipulate, the number of inter-cluster nets and pins, and the average fanout of the remaining nets are all substantially reduced. This can decrease the computation time required by an order of magnitude compared to operating on the original flat netlist.

If one level of clustering is insufficient to simplify the complexity of the problem, then additional levels of clustering (hierarchical or multi-level clustering) may be performed until the problem size becomes manageable. With multi-level clustering, the compaction of the original flat netlist can proceed more gently, with progressively smaller clustered netlists being produced at each level of the hierarchy. Intuitively, the benefit of multi-level clustering is that the iterative improvement strategy has more opportunities for refinement. Furthermore, as the refinement moves from coarser to finer levels, the iterative algorithm can avoid bad local minima because of the large steps taken at the highest levels of the hierarchy. The progressively smaller and more detailed steps at the lower levels of the hierarchy enable the algorithm to find good final solutions [Alpe97c].

In the specific case of hierarchical placement, the clustering serves as a bottom-up preprocessing step. A good clustering algorithm should identify those groups of blocks that are tightly coupled in the netlist and will end up being placed in close proximity in the final placement. Even though top-down partitioning achieves the same divide-and-conquer philosophy as clustering (though starting from a global view and working its way down to a local view of the circuit), the growth in circuit size may make this problem prohibitive (this is why clustering is used to simplify large partitioning problems). It is important, however, that the clustering step be fast itself, otherwise the speedup benefits are nullified. As well, the clusters should be built with as uniform a size as possible, since algorithms that swap clusters become less effective when the sizes of the objects vary greatly [Sun95]. Placement algorithms tend to have more success in minimizing the net lengths of low fanout nets rather than high fanout nets [Roy93]. Consequently, creating a hierarchy of clusters that shrinks the problem size, so that the average fanout of a net is reduced, can be advantageous to achieving both good quality and run time.

2.3.2 Cost Functions Used to Build Clusters

In [Sun95], two levels of clustering are performed prior to hierarchical simulated annealing-based placement. A single level of clusters is constructed in linear time based on connectivity, with the clusters having similar size. The cost function is designed so that nets with small fanout are absorbed into a single cluster. These nets are easier to fit in a single cluster than nets of greater fanout. Each net i is assigned a weight w_i that is inversely proportional to its fanout. So, if the set of pins on net i is F_i, $w_i = 1 / (|F_i| - 1)$. A tree model for multi-terminal nets is used, which means

that an *n*-pin net has *n*-1 edges; if that net spans *m* clusters, then there are *m*-1 inter-cluster edges, and if there are *j* pins of that net contained in a cluster, then that cluster has *j*-1 edges. Let B_k be the set of all pins contained within cluster *k*. The weight, W_k, of cluster *k* is then defined as the sum of all the edge weights in that cluster:

$$W_k = \sum_{\langle \forall i | (F_i \cap B_k) \neq \varnothing \rangle} (|F_i \cap B_k| - 1) \cdot w_i \tag{2.13}$$

The first component of the product term in the summation represents the number of edges of net *i* that are completely contained within cluster *k*. Note that if the entire net was absorbed by cluster *k* (all pins of net *i* are contained within boundaries of cluster *k*), the total edge weight due to that net would be 1. Let *N* be the desired number of clusters to be constructed at a specific level. The objective of the clustering algorithm is to maximize the cost function *C* below, without violating the constraints on cluster capacity (a minimum and maximum size for clusters at a specific level):

$$C = \sum_{k=1}^{N} W_k \tag{2.14}$$

[Sun95] employs a simulated annealing algorithm to build the clusters while maximizing the above cost function, where blocks are swapped or moved across clusters as long as the blocks have connections in each cluster.

In [Roy93], a bottom-up hierarchical technique is also used to construct clusters for partitioning. An accumulative weighted graph is used to represent the circuit netlist, where each node represents a circuit module, and each edge indicates a net containing those two nodes. An *n*-pin net then results in *n*(*n*-1)/2 edges in the complete graph. The clustering targets those nodes in the graph that qualify to be merged with each other. As in [Sun95], each net is weighted by the inverse of its fanout, (1/(*n*-1), for an *n*-pin net). All the edges between every pair of nodes in the graph are collapsed into single edges with a total weight equal to the sum of all the edge weights between the same two nodes.

In the first clustering phase, “natural” clusters are constructed from those nodes which are connected by edges that have weight greater than some threshold. This tends to cluster the most dense subgraphs from the weighted graph of the netlist, and higher levels of clustering proceed in the same way with a condensed netlist of clusters and an updated value for the threshold.

The second phase of clustering uses an adaptive method to refine the clusters created in the first phase. Its goal is to further simplify the netlist while making sure the sizes of the clusters are approximately equal. Small clusters on 2-pin nets are merged together to form larger clusters, but if there are no such nets remaining, then the smallest cluster on the multi-terminal net with the largest fanout is merged with a given small cluster. In this way, both the total number of nets in the revised netlist is reduced, and the average net fanout of the remaining nets is also reduced. The clustered netlist is then transferred to the partitioning/placement tool.

In [Shin93], a single level of bottom-up clustering is utilized to simplify a partitioning problem, as well as provide a hierarchical partitioning tool with a good initial solution that can be efficiently evaluated. The intuition is that a clustering-based initial partitioning solution is superior to a random solution since the blocks that are merged into clusters should ultimately reside in the same partition.

The clustering process begins by considering each circuit block a cluster itself (cluster size = 1). Let *CommonNets(C, D)* represent the set of nets common to clusters *C* and *D*, *Pins(C)* and *Pins(D)* represent the set of pins contained in clusters *C* and *D*, respectively, *ClusterSize(C, D)* be the size of the new cluster formed by merging *C* and *D*, and *AvgClusterSize* be the average size of all the clusters. The “closeness” of two clusters *C* and *D* is given by [Shin93]:

$$closeness(C, D) = \frac{|CommonNets(C, D)|}{min(|Pins(C)|, |Pins(D)|)} - \left(\alpha \cdot \frac{ClusterSize(C, D)}{AvgClusterSize}\right) \qquad (2.15)$$

The first component measures the strength of the attraction between the clusters, while the second component is the penalty for creating clusters with unbalanced sizes, whose influence is controlled by the scaling factor, α. The pairs of clusters with the highest closeness scores are merged until the desired number of clusters is reached. After two clusters are merged, the closeness scores of all the other affected clusters are updated to reflect the change.

Once all the clusters are constructed, the clustered netlist is partitioned multiple times, and the best solution is passed on to the flat partitioner for further refinement. The combined clustering and 2-level partitioning methodology leads to high quality partitioning results in a reasonable execution time.

In [Alpe97c], multi-level clustering is used to simplify the partitioning problem. The various levels of clustered netlists are passed to a hierarchical partitioning tool for repeated partitioning and decomposition of the clusters while maintaining the partitions from the previous higher level of the hierarchy.

The clustering at a given level merges 2 blocks (or clusters) at a time in the following manner: an unclustered block *w* is merged with a given block *v* if *w* has the highest connectivity to *v*, according to the cost function below [Alpe97c]:

$$conn(v, w) = \frac{1}{A(v) \cdot A(w)} \cdot \sum_{e \in \{e | (v \in e, w \in e)\}} \frac{1}{|e|} \quad (2.16)$$

where *A(v)* and *A(w)* are the respective areas of blocks *v* and *w*, and *e* represents each net common to both *v* and *w*. The (1 / |*e*|) component reflects that blocks connected to low fanout nets are preferred, and the (1 / (*A(v)* · *A(w)*)) term indicates that blocks with smaller areas are preferred for merging to maintain clusters with balanced sizes. If there is no suitable unclustered block *w*, then block *v* is assigned to its own separate cluster. Nets with fanout 10 and up are ignored for the purpose of calculating *conn(v,w)*, and the clustering at a given level terminates when a specified fraction of all blocks have been clustered.

2.4 Fast Compile Algorithms

While there exists a great deal of previous work on VLSI placement algorithms that can be applied towards FPGAs, and these algorithms succeed to varying degrees in minimizing the wiring area occupied by a circuit, very few of them have as their primary goal the minimization of run time. In this section, we describe some of the recent work dealing with fast compilation for FPGAs.

2.4.1 Lola

Gehring and Ludwig [Gehr98] describe a fast placement tool in the context of a set of integrated CAD tools for the Xilinx XC6200 FPGA architecture. It converts a Lola HDL specification [Wirt96] into an FPGA programming bitstream in the order of seconds. Their constructive and deterministic placement algorithm operates only on a hierarchical description of a circuit that contains regular subcircuits and is represented by parameterized templates. It takes user-specified position hints and proceeds in a bottom-up fashion to place the inner-most subcircuits, and then recursively places the larger array structures and expression trees. Simple heuristics for placing these structures are employed to place the regular bit-sliced designs that often occur in datapath circuits. The designer is permitted to manually intervene to provide hints and feedback to the

automatic placer that pre-places parts of the circuit. Since the circuit hierarchy is maintained, the user can easily modify the placement manually or provide further hints to constrain future iterations of the placer. Upon reaching a placement solution, the specific placement information for each subcircuit is passed on to every instance of that template.

The placement algorithm is of linear complexity and is fast - a circuit of 11,748 configurable logic blocks (CLBs) was placed in 33.5 seconds on a 166 MHz Intel Pentium. The Xilinx XC6264 is the target device. The authors admit that their strategy does not lead to dense layouts, and that for larger circuits, manual floorplanning may be required.

2.4.2 GAMA

Callahan et al. [Call98] combine fast placement with module mapping in their GAMA tool. It is used to synthesize bit-sliced datapath circuits quickly by treating the placement and mapping problems jointly as a tree covering problem. A dataflow graph representation of the circuit is split into trees, and a linear-time implementation of bottom-up dynamic programming is used to perform the simultaneous module mapping and relative module placement. While creating a tree covering for a particular module, a linear placement is formed by abutting the module with the best covers of its fanin trees. Modules within a subtree are placed contiguously, and the size of a module's fanin trees are used to estimate routing delays that form part of the cost function used to evaluate different covers and placements. Once placement within a tree is complete, a greedy heuristic is employed to perform the global placement of the trees that seeks to place trees on the critical path close to each other. After post-covering and post-placement local optimizations are made, each module is generated on demand, rather than copying it from a static library of modules with different widths, shapes, and orientations.

By opting to maintain the hierarchical datapath circuit structure, rather than flattening the design to a netlist of gates, the authors are able to exploit specialized features of their target FPGA device architectures, such as fast carry chains. They obtain good results when targeting the Xilinx XC4000 family and explore the trade-off between optimizing for area (minimum number of configurable logic blocks, CLBs, required) and for delay (minimizing critical path delay through the dataflow graph, or minimizing the number of CLBs while meeting a specific timing constraint).

The authors admit that only a subset of all possible linear orderings of optimized modules are evaluated, and so an optimal placement cannot be ensured. However, GAMA can be beneficial in situations where placement quality may be sacrificed for compilation time; it can provide a quick initial solution for further iterative refinement or when a fast estimate of placement cost is needed.

2.4.3 Fast Placement for FPGAs via Automatic Floorplanning

In [Tess98], compile-time efficient placement for FPGAs is approached using ASIC floorplanning techniques. By considering portions of the circuit being mapped to the FPGA as pre-placed and pre-routed macrocells, the compile times for large designs can be decreased from an hour to mere minutes, although there is both a severe area and circuit speed penalty. Using macros and floorplanning tends to yield a fast placement result, but following it up with some amount of annealing tends to smooth out any congestion and permits better device utilization. Furthermore, if inter-macro routing is not taken into account during the floorplanning stage, this can lead to longer routing times than those obtained by using classical flat place-and-route heuristics due to the increase in total wirelength.

For circuits with an explicit hierarchical structure, an iterative strategy using relative cell placement, cell rotation, and cell mirroring is used to achieve the goal of minimum interconnection wirelength. The method first involves finding optimized layouts for a set of macrocells that will be stored in a library database. Since each macrocell is small, this does not expend much time, and only needs to be done once. The circuit is then recursively bipartitioned using classical algorithms to create a slicing floorplan with minimum cut size, where each leaf in the binary slicing tree is a single macro. Then, floorplan sizing is performed using a bottom-up dynamic programming approach. All feasible shape combinations are evaluated in this traversal from the leaves to the root of the floorplan’s slicing tree, and those which consume too much area are pruned. A top-down slicing tree traversal is then performed, and at each level of the traversal, the combination of macrocells that fits the shape constraints laid out in the previous step is chosen such that wirelength is kept at a minimum. At the end of this stage, the floorplan contains a list of macro locations, shapes, and orientations that result in a macrocell placement on the minimum-sized square FPGA that can accommodate it.

2.4.4 Fast Routing and Difficulty Prediction

In [Swar98a], the routing phase of the fast FPGA compile is addressed. It is the complement to this work and together they form the Fast Compile Project at the University of Toronto. Swartz et al. present a fast routability-driven FPGA router, and later propose a fast timing-aware router in [Swar98b]. Their enhancements to the basic maze router include using a more aggressive, depth-first search technique to route nets from sources to sinks, the choice of target sinks when routing multi-terminal nets, routing the nets in decreasing order of fanout, and reducing problem complexity by dividing the FPGA into uniformly-sized bins. Another key contribution of this work is the feedback provided to the user of the predicted difficulty of a given routing problem. Given a placement and a target FPGA, the tool estimates the minimum number of tracks per channel required to route the circuit on the given FPGA from the bounding-box wirelength placement cost. It then compares this figure to the actual number of available tracks per channel on the device, and informs the user that either the circuit can be routed quickly, that it can be routed given more time, or that it is unroutable given the current target FPGA.

2.5 Summary

In this chapter, we provided definitions for the FPGA placement problem and the ultra-fast placement problem. We then reviewed some of the prior work done in general VLSI placement algorithms, focusing on simulated annealing-based heuristics. We also discussed previous efforts that used clustering to simplify and speed up placement and partitioning problems and the various cost functions employed to build clusters from a flat netlist. Finally, we presented some of the recent work accomplished in creating fast compilation tools for FPGAs. This provides the background information necessary to understand the design details of the ultra-fast placement algorithm, which is described in the next chapter.

Chapter 3

Ultra-Fast Placement Algorithm

In this chapter, the components of the ultra-fast placement algorithm are described, including the parameters that allow the exchange of wirelength for compile time. We then describe how we determined a stable set of these parameters that give us the best quality-time trade-off.

3.1 Overview of Approach

Recall that the placement problem for FPGAs begins with a technology-mapped netlist of logic blocks, I/O pads, and their interconnections. The output is an assignment of the blocks and pads to specific physical locations of the FPGA. To achieve ultra-high-speed placement for FPGAs, we build upon the clustering and hierarchical simulated annealing algorithm described in [Sun95] and the adaptive annealing schedule of [Betz97] [Betz98]. We integrate it into the infrastructure provided by VPR (the Versatile Place and Route tool presented in [Betz97]).

Figure 3.1 illustrates the framework for the algorithm [Sank99]. The first stage is a multi-level, bottom-up clustering of the logic blocks. Note that we do not incorporate I/O pads into the clusters of logic blocks, since they have restrictions on their placement. The clustering is parameterized as follows: a total of L different levels of clustering will be performed. At each level i, s_i blocks (or clusters) at the previous level are grouped into a cluster. If a circuit contains a total of N logic blocks, then after a single level of clustering (level 1), there are $\lceil N/s_1 \rceil$ clusters. These clusters can be grouped again to create a second level of clustering, with s_2 first-level clusters in each second-level cluster, giving $\lceil \lceil N/s_1 \rceil / s_2 \rceil$ clusters at the top level (level 2), and so on.

Once all the required clustering is done, placement must be performed at each level of the newly-formed hierarchy. We employ a two-step approach at each level: an initial constructive

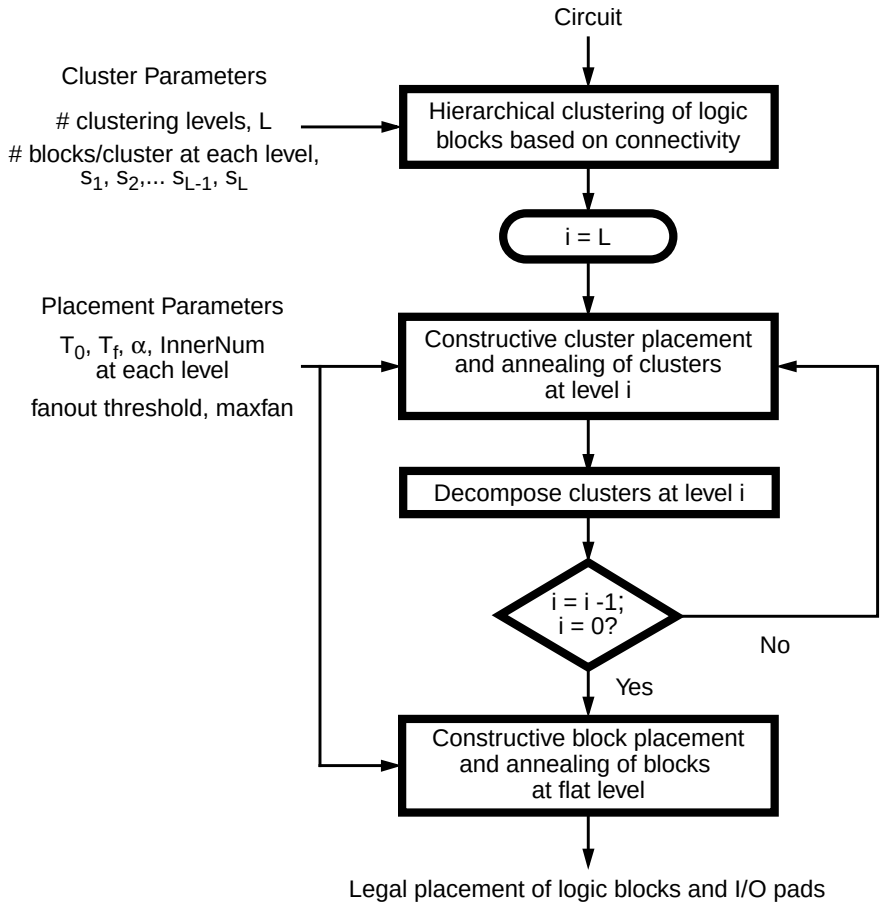

Figure 3.1: High-level view of fast placement algorithm.

placement followed by iterative improvement using simulated annealing. The parameters of the anneal are tuned to secure a good quality-time trade-off, as described below. Figure 3.2 illustrates an abstract view of the multi-level clustering and placement. Our goal is to achieve high-speed placement by quickly making good and fast global decisions at the higher levels of the hierarchy, and following this with iterative local improvement at each level. Our choices of algorithms are guided by the following objective: reduce the complexity of a large placement problem by dividing it into manageable portions, and then employ known heuristics that are simple, fast, and effective on each portion. As we have seen in Chapter 2, multi-level clustering has been used to simplify and speed up the solution of large placement and partitioning problems. This serves as the motivation behind using this approach to attack the fast placement problem for FPGAs.

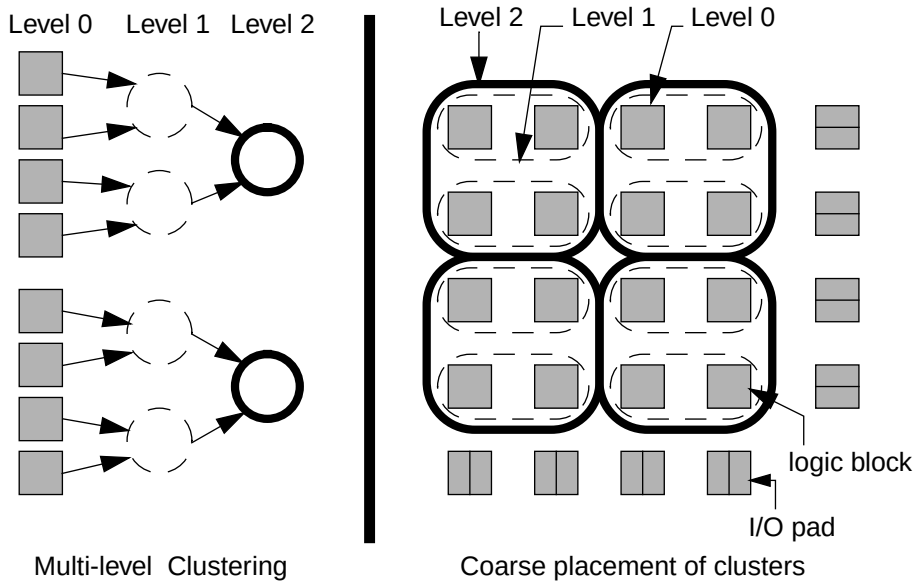

Figure 3.2: Abstract view of multi-level clustering and placement.

3.2 Multiple-Level Clustering

3.2.1 Description of Algorithm

The first step of the ultra-fast placement algorithm is a multi-level bottom-up clustering of logic blocks. The input to the clustering step is a netlist of N logic blocks and their interconnections, the number of clustering levels, L, and the cluster size at each level, s_1, s_2,... s_L. We restrict the cluster sizes (s_i) to be perfect squares (4, 16, 25, 64...) in order to simplify the grid resizing operations at the various levels of placement. The task is to create L separate netlists of clusters of logic blocks and their interconnections, where each block or lower-level cluster is assigned to a unique higher-level cluster exactly once, and each cluster $c_{i,k}$ (the kth cluster at level i) has at most s_i blocks or clusters from the previous level, i-1.

The clustering algorithm begins by randomly choosing a logic block as a seed, and assigning it to the first slot in a cluster. Each unclustered block connected to that seed is assigned a score that rates how much the block belongs to this cluster. This score, w_b, for each candidate block b has two components: (1) the number of connections between the candidate and the cluster being constructed, with each connection weighted by the fanout of the net on which it lies, as in [Sun95], and (2) the number of nets that would be completely absorbed if this candidate were

added to the current cluster. We say that a net is *absorbed* by a cluster if all the blocks on that net are contained within that single cluster. If we denote J to represent the set of nets shared between the candidate block b and the cluster c under construction, P_j as the set of pins on net $j \in J$, and A_{bc} as the set of nets absorbed by adding candidate block b to cluster c, then the score can be expressed as:

$$w_b = \sum_{j \in J} \frac{1}{|P_j| - 1} + |A_{bc}| \tag{3.1}$$

With this function, blocks on low-fanout nets and on nets that are about to be absorbed are preferred when building the clusters. The candidate block with the highest score is added to the next available cluster slot, and if the cluster is full, a new one is started with a new randomly selected seed block. This process is repeated until all the blocks are clustered. The result is a netlist of clusters with absorbed nets and redundant pins removed. We proceed in a similar manner to create further levels in the clustering hierarchy. The pseudo-code for the clustering algorithm is presented in Figure 3.3.

The number of clustering levels, L and the size of the clusters at each level, s_i can be varied to allow the trade-off between compile time and quality. As the size of the clusters increases, the placement problems become simpler because more details are hidden, but there is less accurate representation of the netlist and therefore lower quality may result. By focusing on the collapse of low-fanout nets via clustering, we are implying that blocks connected to those nets will be placed in close proximity after a high-quality flat placement. Furthermore, the removal of these nets from the global view of the circuit and the use of clustering to reduce the average fanout of the remaining non-absorbed nets will allow the placement phase to devote its attention at higher levels in the hierarchy to nets whose lengths are more difficult to minimize in a flat placement scheme, and do so in a much quicker fashion.

3.2.2 Clustering Example

Figure 3.4 provides an example to illustrate the clustering algorithm. The cluster under construction can hold up to four blocks, and three of its slots are filled with logic blocks *x, y,* and *z*. In order to fill the remaining available cluster slot, we examine the candidate blocks that share connections with the current cluster. Upon adding block *z* to the cluster, the most recent addition, scores are incrementally updated or assigned to blocks that are newly connected to the cluster and

```
Input: # of clustering levels, L; cluster size at each level, s_1 .. s_L
procedure cluster_blocks {
1       curr_level = 1;
2       Repeat {
3               Repeat {
4                       Start new cluster and flush bucket structure;
5                       Select random seed block;
6                       Repeat {
7                               for each candidate block b connected to seed {
8                                       Compute/incrementally update cluster score, w_b, from (3.1);
9                                       Store score in bucket structure;
10                                      if (w_b > w_best_block) {
11                                              best_block = b;
12                                      } /* end if */
13                              } /* end for */
14                              Add best_block to next free cluster slot;
15                              seed = best_block;
16                      } until (cluster is full || no more blocks at this level to cluster)
17              } until all logic blocks at this level are clustered
18              Revise netlist at this level -> remove absorbed nets and redundant pins;
19              curr_level++;  /* proceed to next level of clustering */
20      } until all L levels of clustering complete
} /* end cluster_blocks */
```

Figure 3.3: Pseudo-code for multi-level clustering algorithm.

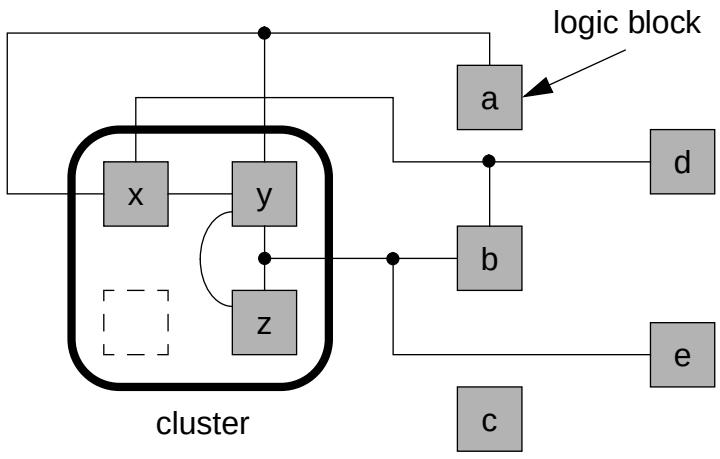

Figure 3.4: Clustering example.

those that are on nets about to be absorbed. The cost function in Equation (3.1) gives the scores in Table 3.1.

Table 3.1: Cluster scores for candidate blocks in example of Figure 3.4.

Block	Cluster Score
a	1.50
b	0.83
c	0
d	0.50
e	0.33

Note that based on its cluster score being the highest, block *a* would be chosen for the remaining free cluster slot over blocks *b, d, e,* and *c*, respectively. In fact, block *c* would not even be considered as a candidate since it does not share any connections with the cluster. The attractiveness of block *a* is apparent from its sharing a single 3-pin net with the cluster and that this net would be absorbed if block *a* was added to the cluster. If there are available cluster slots and none of the remaining unclustered candidate blocks share any connections with the cluster under construction, then a candidate is selected randomly and assigned to the next free slot.

Figure 3.5 illustrates how cluster size influences the amount of absorption of nets resulting from a single level of clustering for one circuit in our benchmark suite. Note that even for relatively small cluster sizes (between 2 and 10 logic blocks per cluster), a significant proportion of flat nets are collapsed (25-60%), and up to 80% of the nets from the original netlist are eliminated with a cluster size that is under 100 blocks. While the data shown in the graph was obtained from clustering a single circuit using different cluster sizes, and the results from different circuits are influenced by their fanout distribution, these trends are apparent in all the circuits.

3.2.3 Complexity of Clustering

The score assigned to any candidate block changes only when a net connected to that block is first connected to a cluster, or when that net is about to be absorbed (i.e. all but block *b* of the pins on net *j* are contained in cluster *c*, and the cluster has an available slot). The list of the best scores and associated candidates is maintained in a bucket-sorted data structure in order to perform fast updates [Corm90]. The bucket structure, which is sorted by score, only needs to be flushed when a cluster is full. Let *N* be the number of logic blocks, *K* be the number of nets on

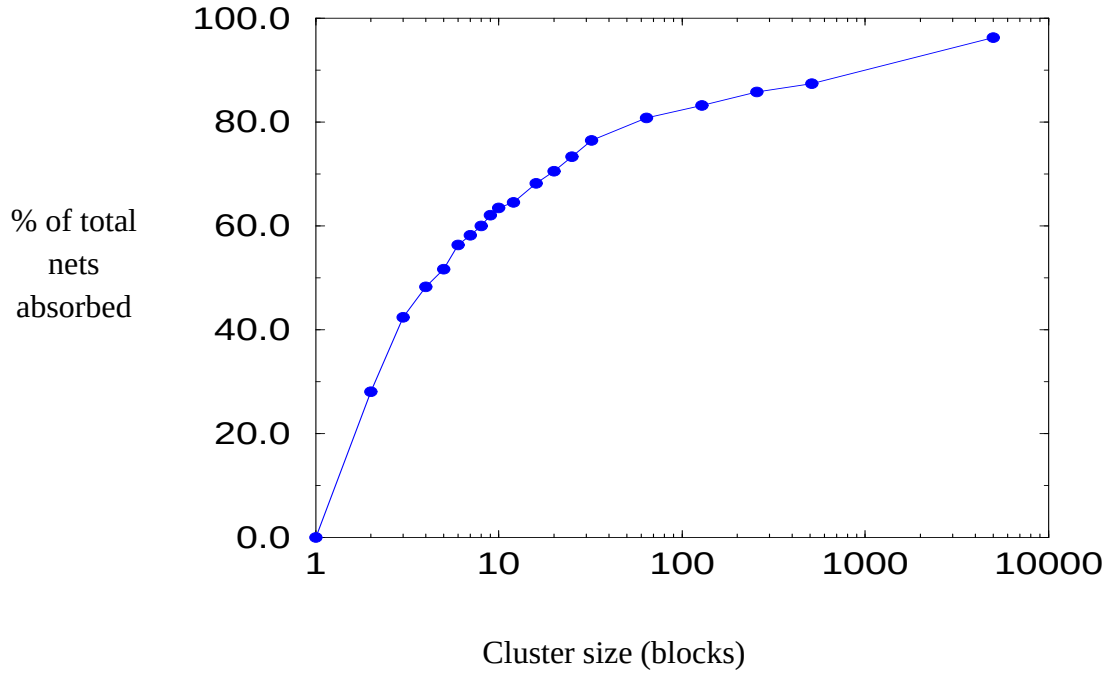

Figure 3.5: Graph of percentage of total flat nets absorbed versus cluster size for one level of clustering on MCNC circuit **frisc** (3692 blocks).

each logic block, f_{max} be the maximum fanout of any net in the circuit, and s be the size of the cluster. The complexity of the algorithm can be derived by observing that when generating the clusters, the algorithm must examine each of the N blocks once, each of the K nets connected to the block, and each of the other pins on those nets. This examination occurs either upon adding a block to a cluster or when a net is about to be absorbed. The complexity of the clustering algorithm is thus $O(N \cdot K \cdot f_{max})$. If we clip the value of f_{max} by restricting the clustering algorithm from examining nets above a certain fanout threshold, this is a linear-time algorithm. This bound is satisfied at higher levels of clustering as well, since N is scaled down by a factor of the cluster size (s), K is scaled up by at most a factor of s (and is often less than that), and f_{max} is likely to decrease. Practically speaking, the clustering is very fast: a 20,000 4-LUT circuit can be clustered into clusters of size 64 in 2.1 seconds on a 300MHz Sun UltraSPARC workstation.

3.3 Placement of Clusters at Each Level

Once we have constructed the hierarchy of clusters, placement must occur at each level. The placement algorithm consists of two steps: constructive placement followed by simulated-annealing-based iterative improvement.

3.3.1 Constructive Placement of Clusters

Given a netlist of clusters and their interconnections, we first perform a random placement of all the I/O pads in the circuit at the highest level of the hierarchy. This provides anchor points for the constructive placement of the clusters. Note that subsequent optimization steps will change this initial pad placement.

The constructive placement determines positions for three separate groups of clusters: (1) those connected to output pads, (2) those connected to input pads, and (3) those connected to other logic clusters. It computes, for each cluster in each group in succession, the arithmetic mean position of all the clusters and pads it is connected to that have already been placed. The cluster is then placed as close to this “center of gravity” as possible. The initial placement of the pads provides the guidance for this construction. We have found that this method provides a superior starting point for the subsequent iterative improvement step compared to using a simple random placement. Experiments also show that this placement results in a slightly better quality-time trade-off than an initial random placement. The pseudo-code for the constructive cluster placement step is shown in Figure 3.6.

At lower levels in the hierarchy, the same constructive approach is used, with three exceptions: (1) there is no initial pad placement -- pads are placed in the same way logic clusters are; (2) if a block has not yet been placed and its position is needed for the mean calculation, the geometric center of the higher-level cluster it is contained within is used as the position; (3) each of the cluster contents is placed as close to its calculated “center of gravity” while remaining within the prescribed cluster boundaries.

At each level of the hierarchy, it is possible to perform multiple iterations of the constructive placement algorithm prior to the iterative improvement step. The objective is to refine the placement using the same constructive algorithm but making use of the complete placement information available from the previous iteration. This helps to fix the problem that early in the first iteration, pads and clusters are placed without exact information on the location of the clusters to

```
Input: netlist of clusters, I/O pads, and interconnections, hierarchy level number
procedure place_clusters (int curr_level) {
1     calculate grid size for clusters at level curr_level;
2     if (curr_level == top level in hierarchy) {
3           sort clusters into list, sorted_clusters[ ], in following order:
                  clusters connected to output pads, input pads, and the rest;
4           do random pad placement;
5     }
6     else {  /* lower levels of hierarchy */
7           calculate boundaries induced by each level (curr_level+1) cluster;
8           sorted clusters[ ] = list of clusters and pads at level curr_level;
9     }  /* end if */
10    k = 0;
11    while not all clusters and pads have been placed {
12          curr_cluster = sorted_clusters[k];
13          total_x = total_y = total_connections = 0;
14          for each connection, conn, to curr_cluster {
15                if (curr_level != top level in hierarchy) && (conn is unplaced)
16                      use center of cluster at level curr_level+1;
17                if (conn has been assigned a legal location)
18                      total_x += conn.x; total_y += conn.y; total_connections++;
19          }  /* end for */
20          mean_x = total_x / total_connections;
21          mean_y = total_y / total_connections;
22          place curr_cluster as close to (mean_x, mean_y);
            /* without violating array or induced cluster boundaries */
23          label curr_cluster as placed; num_clusters_placed++;
24          k++;  /* examine next unplaced cluster or pad */
25    } /* end while */
} /* end place_clusters */
```

Figure 3.6: Pseudo-code for constructive cluster or intra-cluster placement.

which they are connected. Although multiple passes of the constructive placement algorithm tended to improve the wirelength slightly at each level in the hierarchy, there was no significant difference in final placement cost at each level after the simulated annealing-based iterative improvement phase, irrespective of the number of iterations performed by the constructive placer.

Table 3.2 shows a comparison between the constructive cluster placement algorithm and a quick and simple random placement at clustering levels below the top level. After the prescribed clustering was completed, a top-level placement was performed with our constructive placement algorithm. At all successively lower levels, the initial cluster placement was performed using the constructive algorithm, in one case, and a random placement algorithm, in the other. To ensure

that the comparison was fair, in both cases the follow-up iterative improvement phase at each level was the same. The data shown is the geometric mean of the normalized final placement cost and the associated mean overall run time across 20 benchmark circuits, using the different initial cluster placement algorithms. For 1 and 2 levels of clustering, there is a slight benefit in placement quality at the expense of a slight increase in run time when using the constructive placement algorithm. For 3 levels of clustering, the difference between the two algorithms appears to favor the random placement algorithm. Note that this is a comparison for the levels below the top level of the hierarchy; the constructive placement algorithm delivers a better quality-run time trade-off than a random placement of the top-level clusters.

Table 3.2: Constructive versus random initial cluster placement.

# Cluster Levels	Length of Follow-up Iterative Improvement Phase	Constructive		Random	
		Mean Run Time (s)	Geometric Mean Normalized Wirelength (20 circuit average)	Mean Run Time (s)	Geometric Mean Normalized Wirelength (20 circuit average)
1	short	7.36	1.567	6.10	1.589
	medium	13.09	1.250	11.77	1.265
	long	61.76	1.036	60.20	1.042
2	short	7.96	1.595	6.71	1.622
	medium	13.69	1.279	12.30	1.292
	long	61.78	1.048	58.87	1.065
3	short	9.77	1.581	8.13	1.594
	medium	15.28	1.266	13.80	1.262
	long	62.70	1.050	61.74	1.046

3.3.2 Simulated-Annealing-Based Iterative Improvement of Placement

Following the constructive placement of clusters and pads at any level in the hierarchy, we improve its quality using simulated annealing-based [Kirk83] [Sech85] iterative improvement before proceeding to the next lower level. Refer to Chapter 2 for a description of the basic simulated annealing method as it is applied to placement. We have adapted the annealing implementation in VPR described in [Betz97] [Betz98].

One important issue is whether or not to restrict the motion of blocks to remain within the cluster boundaries of the most recent cluster level. We have experimentally determined that at every level of the hierarchy it is *much* better to allow the blocks being placed to *move across* the cluster boundaries. This still means that the coarse placement from the previous level is useful; if the boundaries are strictly enforced, however, then placement quality suffers. To demonstrate this, an experiment was run on a set of 20 benchmark circuits comparing placing flat blocks within cluster boundaries to placing them across the boundaries. A single level of clustering was specified, with identical cluster sizes and identical top-level and flat-level placement parameters. In both cases, a geometric average run time of 14.04 seconds was obtained across the 20 circuits. However, the geometric average normalized placement cost was 1.23 when blocks were allowed to cross cluster boundaries, while a cost of 1.51 was obtained when the blocks were restricted to moving only within those boundaries.

The key parameters that control the quality-time trade-off for simulated annealing are:

- The starting temperature, T_0. This is a crucial parameter, because if the temperature is set too high, the subsequent annealing will destroy the placement structure developed at previous levels in the hierarchy. If it is set too low, then insufficient optimization will be performed. We employ three different mechanisms for determining T_0. The first is to employ the temperature “measurement” mechanism (simulated thermometer) suggested in [Rose90] -- here, the initial temperature is determined by finding the temperature at which the placement appears to be at equilibrium. The second is to set the initial temperature to zero (a “quench”, where only moves that improve the placement cost are accepted), and the third is to set the starting temperature to a fixed value, greater than zero. In the next section, we explore which of these approaches is most appropriate for different quality-time trade-off points.

  In Table 3.3, we show the effect of varying the starting temperature for the follow-up anneal at the flat level after a single level of clustering of the MCNC [Yang91] circuit clma. A constructive top-level placement, a follow-up top-level quench, and a constructive flat placement were performed. The starting temperature for the follow-up flat anneal was varied from 100 down to 0.025, the exit temperature was fixed at 0.01, and the number of moves attempted per temperature was adjusted so that the overall run times were comparable. The final placement costs obtained were compared to using the

simulated thermometer to determine the starting temperature and proceeding with an automatic flat anneal where the temperature update factor and exit temperature are computed dynamically. The usefulness of the simulated thermometer is apparent when compared to the quality-time trade-off obtained using pre-specified starting temperatures that do not take into account the state of the current placement. The run times are similar, but the placement with lowest cost is generated by the anneal that uses the thermometer. Note that as the starting temperature is reduced from 100 to 0.1, hill-climbing is still accomplished, but in smaller amounts in the same amount of execution time. However, at very low temperatures (< 0.1), the annealer is caught in local minima that are difficult to escape, and there are not enough temperature steps for significant optimization. The thermometer permits a nice trade-off between these two ranges of temperatures, and is computed dynamically using the current placement of the circuit in question at any level in the hierarchy, hence it is more robust and adaptive than a static choice of starting temperature.

Table 3.3: Effect of starting temperature calculation on annealing for MCNC circuit **clma**.

T_0	Run Time (s)	Normalized Placement Cost
100	23.52	2.29
10	22.04	2.22
1	21.92	1.97
0.1	21.76	1.22
0.025	21.64	1.35
thermometer (0.089)	21.87	1.21

- The number of “moves” per temperature, called “*InnerNum*.” The basic annealing algorithm of VPR [Betz97] makes $InnerNum \cdot N_{blocks}^{4/3}$ moves at each temperature, where N_{blocks} is the total number of blocks and pads. The parameter *InnerNum* determines how much work is done per temperature.
- The temperature update factor, α. This factor is the amount by which the temperature is reduced between iterations of the main annealer loop ($T_{new} = \alpha \cdot T_{old}$). A lower value for α results in a faster anneal, but also a reduction in quality. VPR [Betz97] automatically

adjusts α as described in Section 2.2.3; we have found that squaring the automatic α increases the speed with which the algorithm converges with little reduction in quality.

- The exit criterion -- what causes the annealing to stop -- is either a pre-specified temperature at which the annealing terminates (T_f) or when one of the following two conditions are met: (i) the temperature is less than 1% of the average cost per net, or (ii) the average cost over the last three temperatures is unchanged.

We have identified three types of schedules that permit us to explore the quality-time space thoroughly: (1) an aggressive, dynamic, adaptive schedule with automatic calculations for T_0, T_f, and α; (2) a quench (all moves made at temperature = 0), where no hill-climbing is permitted; (3) a manually-specified schedule where the values of T_0, T_f and α are fixed. Schedule (1) is an anneal tailored to the current placement of the circuit, whatever its level of granularity, schedule (2) is a greedy heuristic, and schedule (3) is a short, fixed anneal. In all three cases, we can trade quality for compile time by varying the *InnerNum* parameter.

3.3.3 Fanout

Another enhancement that we implement to speed up the placement is to ignore nets with large fanout. This is useful because a high-fanout net will likely cover much of the FPGA and so it is harder to reduce its wirelength. By ignoring nets above a certain fanout threshold (called *maxfan* in our tool), the placement problem is further simplified. If we set the threshold too low, however, we may lack enough information to create a good placement. Note that both the clustering and placement steps ignore the nets above this threshold, *maxfan*.

To illustrate how varying the fanout threshold impacts the amount of information remaining in the netlist and the subsequent placement quality, we conducted a set of experiments on the MCNC [Yang91] circuit clma. The value of *maxfan* was varied from 1 to 10000. For each of these fanout thresholds, the nets with fanout greater than *maxfan* were removed from the netlist, and the percentage of total flat pins that remained in the circuit was recorded. Then, a high-quality simulated annealing-based placement tool, VPR [Betz97] [Betz98], was run on the simplified netlist. After the anneal terminated, the ignored nets were reinstated, and the cost was compared to that obtained when the original circuit was placed (with no nets ignored). In Figure 3.7, the percentage of total pins remaining and the percentage increase in VPR placement cost are shown as functions of the fanout threshold, *maxfan*, for this circuit. Clearly, when all nets with more than 2 pins are ignored (*maxfan* = 1), there is not enough information left to perform a good placement (only

30% of the pins remain), and the quality degradation is severe (nearly 3 times the wirelength). However, when the fanout threshold is raised to 10 (nets with 11 pins and less are kept), over 50% of the pins in the netlist are intact, and the resulting annealed placement has a less than 30% increase in wirelength compared to annealing the original version of clma.

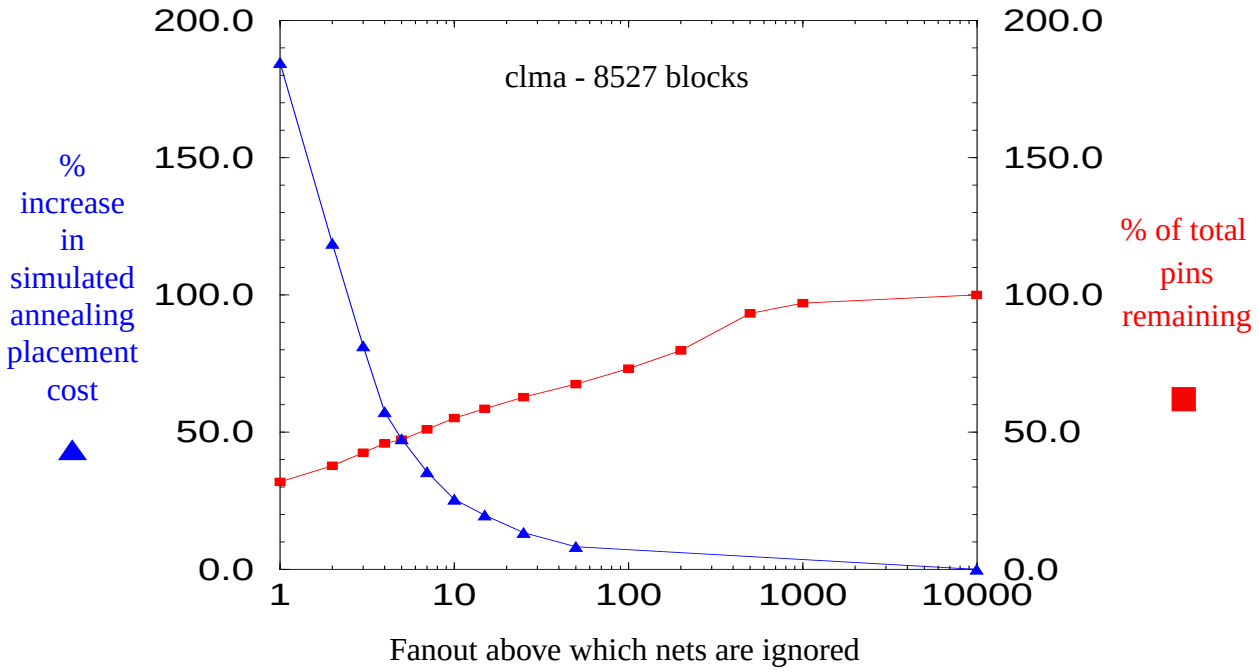

Figure 3.7: Graphs of placement cost degradation and percentage of total flat pins remaining after nets above threshold ignored, each versus fanout threshold for MCNC circuit **clma**.

3.3.4 Complexity of Placement

At any level in the hierarchy, the initial constructive placement algorithm has worst-case time complexity $O(N_{blocks} \cdot K \cdot f_{max})$, with N_{blocks} logic blocks and pads, K pins per block, and a maximum fanout of f_{max} for any net in the circuit. Just as with the clustering algorithm, this is because we must examine each block (or cluster) exactly once, examine each net connected to that block, and examine every other block on that net. Furthermore, by examining only those nets below a certain fanout threshold, we can ensure that it remains a linear-time algorithm. The experiments described in the previous section and Figure 3.7, in which nets above a certain fanout threshold are ignored, indicate that, practically, it is possible to provide an even tighter bound on the complexity of the constructive placement algorithm.

Assume there are N logic blocks and $(PI+PO)$ pads in the circuit, and that we choose a uniform cluster size of s at each level of the hierarchy. For the follow-up simulated annealing algo-

rithm, we explore at each level i (i = 0...L) at most $InnerNum \cdot ((N/s^i)+PI+PO)^{4/3}$ configurations per temperature. The intelligent starting temperature calculation and aggressive adaptive annealing schedule typically ensure that we do not search through many temperatures per level in the clustering hierarchy. This means that the annealing algorithm's worst-case time complexity is bounded by O($N_{blocks}^{4/3}$) and is typically less than that.

3.4 Determination of the Quality-Time Envelope Parameters

In this section, we describe the experiments used to identify the set of parameters for the ultra-fast placement tool and choose those parameters. There are two sets of parameters: those that control the clustering, and those that control the iterative-improvement of the placement. Our goal is to determine the parameters that lead to the best quality-time trade-off for our tool, which we call the *envelope* parameters. Please note that the details of the actual FPGA architecture and the other parts of the CAD flow are given in Chapter 4.

3.4.1 Cluster Parameter Experiments

The key parameters of the multiple-level clustering approach are the number of clustering levels (L) and the cluster sizes at each level (s_1 ... s_L). We first explored a single level of clustering with L = 1. To determine the cluster size value (s_1) that provides the best quality-time trade-off, we ran the placement tool on a set of benchmark circuits and varied the cluster size from 4 to 4096 by powers of 4. For the subsequent iterative improvement placement, many different annealing parameters were run in order to determine the complete quality-time trade-off possibilities. For example, Figure 3.8 is a plot of the geometric mean normalized placement wirelength (with respect to the best possible placement obtained by VPR [Betz97]) versus the geometric mean run time, across a set of 20 benchmark circuits. In that figure, the clustering size s_1 was set to 64.

We performed similar experiments and generated the same curve for values of s_1 = 4, 16, 64, 256, 1024 and 4096, and determined that the values of 64 and 16 resulted in the best (lowest cost) quality-time trade-off curve. Figure 3.9 shows the comparison of the quality-time curves for each value of s_1, and we chose to use 64 as our 1-level cluster size in the placement parameter experiments that follow. This results in netlists with fewer clusters that are larger in size compared to a 1-level cluster size of 16.

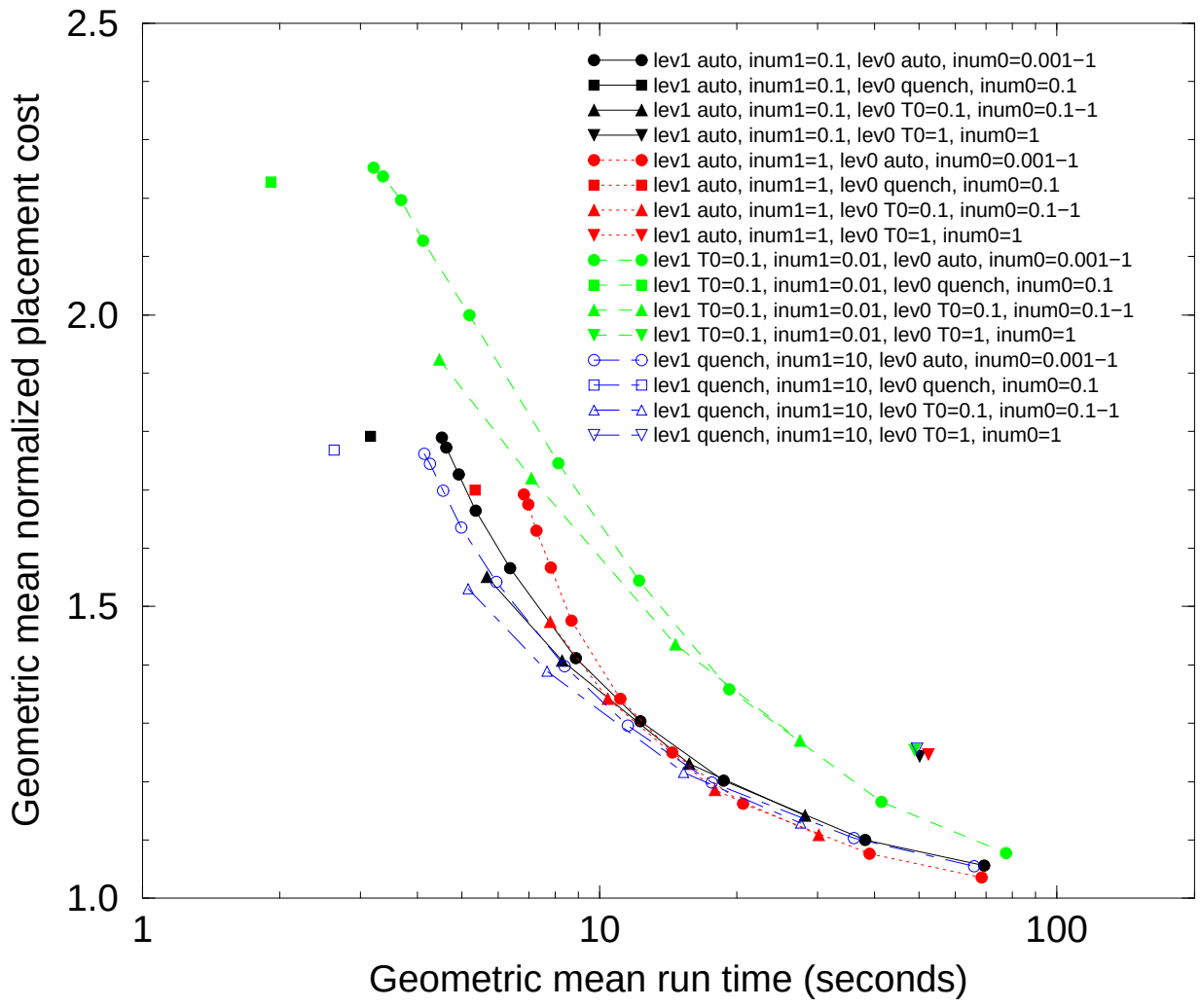

Figure 3.8: Placement quality-time plot (20 circuit average) for ultra-fast placement tool using different combinations of annealing schedules on 1-level, size-64 clustered circuits.

We performed similar studies for L = 2 and 3 levels of clustering. These studies are problematic as there are many more parameters to explore: for L = 2, the values of s_1 and s_2 must both be set; for L = 3, the values of s_1, s_2 and s_3 must all be specified. Furthermore, annealing parameters must be specified at each of those levels. The experiments demonstrated that for L = 2, the values of s_1 = 64 and s_2 = 4 were found to work best, and in a few cases, the quality-run time trade-off was superior to the best of the L = 1 envelope. For L = 3, the values for (s_1,s_2,s_3) of (64,4,4), (64,16,4) and (256,4,4) were all found to behave about the same, but all of these settings yielded results that were no better than those obtained across L = 1 and 2. This may be due to the sizes of the large circuits in our benchmark suite; after 2 levels of clustering, the circuits have

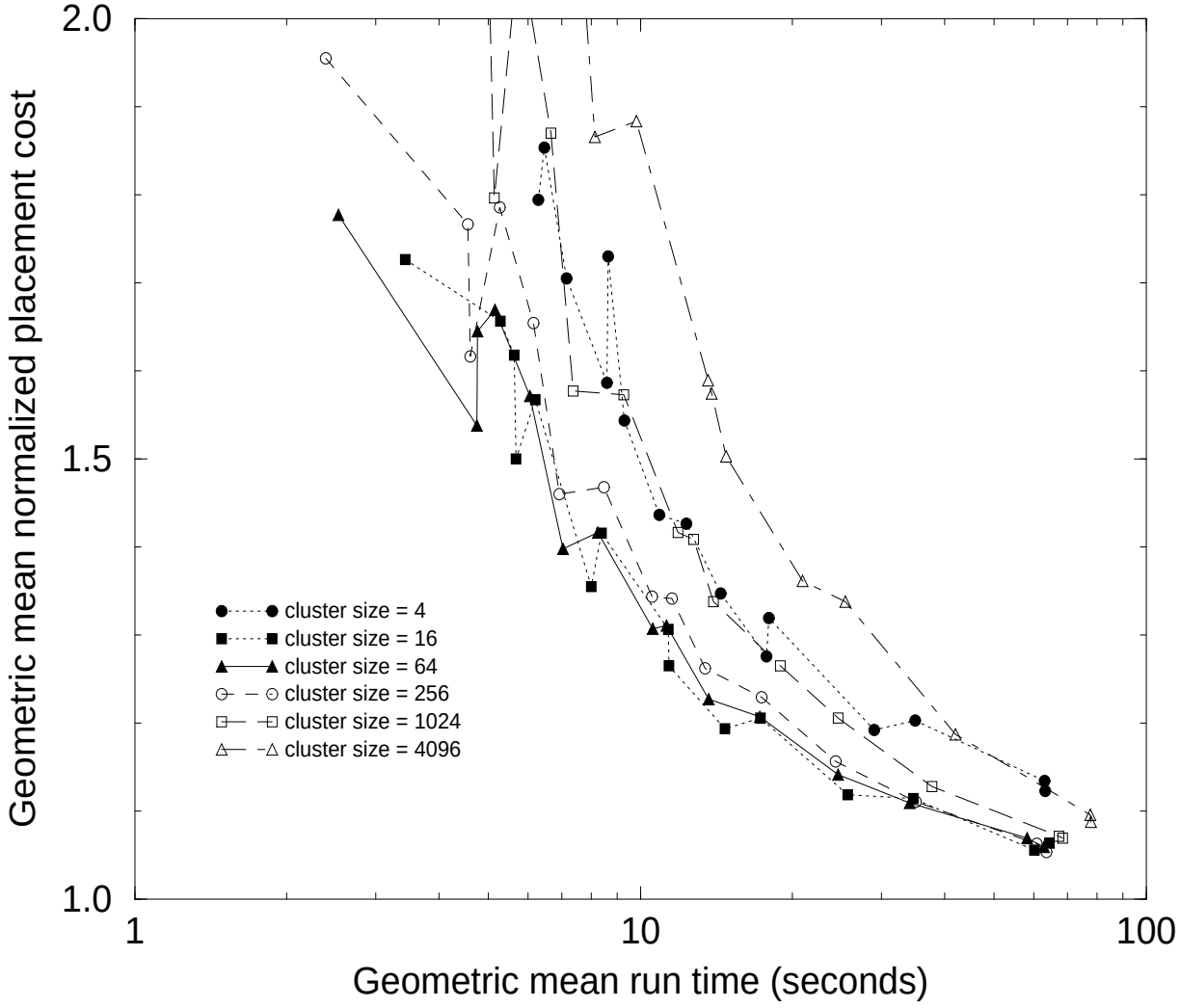

Figure 3.9: Placement quality-time curves (20 circuit average) for ultra-fast placement tool using a sample of annealing parameters and varying 1-level cluster sizes from 4 to 4096.

already been transformed into a few large clusters (tens of clusters, each with 256 total flat blocks). So, an additional level of clustering does little to further simplify the placement problem, and may even cost both time (because of the extra processing at level 3) and area (an additional level of grid resizing must be performed, which may adversely affect the grid size at the flat level).

3.4.2 Placement Parameter Experiments

The next set of parameters to tune is the set of simulated annealing parameters described in Section 3.3.2. Recall that we settled on the set of three types of schedules described in Section 3.3.2: (1) an automatic anneal using a simulated thermometer to compute T_0, dynamically calcu-

lated values for T_f and α, and variable *InnerNum*; (2) a quench with variable *InnerNum*; (3) a fixed anneal with $T_0 = 0.1$, $T_f = 0.01$, $\alpha = 0.8$, and a variable *InnerNum*. We explored the combinations of these schedules at the clustered and flat levels of the hierarchy, for circuits clustered with $L = 1$ and $s_1 = 64$ blocks per cluster. The scatter plot of geometric mean normalized placement cost versus geometric mean run time across 20 circuits is given in Figure 3.8, and note the complexity of the various combinations of schedules. For short run times, the envelope is comprised of a quench (schedule 2) at the top level with *InnerNum* = 10, and the short, fixed annealing schedule (schedule 3) at the flat level with *InnerNum* of 0.1 to 0.5. For longer run times, the envelope consists of the automatic anneal (schedule 1) at the top level with *InnerNum* = 1 and the automatic anneal at the flat level with *InnerNum* from 0.2 to 1. In each case, though, it is evident that there are alternative schedules that come reasonably close to providing the same quality-time trade-off as the envelope.

Similar combinations of annealing schedules were attempted for 2 and 3-level clustered circuits. In the case of circuits with $L = 2$ levels of clustering and cluster sizes ($s_1 = 64$, $s_2 = 4$), the envelope parameters leading to short run times consist of a quench at the top level (level 2), a quench at level 1 (both with *InnerNum* = 10), and a fixed anneal at the flat level with *InnerNum* from 0.01 to 1. For medium run times, the 2-level envelope consists of a quench at the top level with *InnerNum* = 10, a short fixed anneal at level 1 with *InnerNum* = 1, and at the flat level, either an automatic anneal (*InnerNum* from 0.1 to 0.5) or a short fixed anneal (*InnerNum* from 0.2 to 1). For longer run times, the 2-level envelope consists of fixed anneals at level 2 and level 1 (both with *InnerNum* = 1) and a flat automatic anneal (*InnerNum* = 1), or a top-level automatic anneal (*InnerNum* = 1), a level 1 fixed anneal (*InnerNum* = 1), and a flat automatic anneal (*InnerNum* from 0.5 to 1).

The placement quality-time envelope formed by the placement parameters for 3-level clustered circuits is no better than the 1-level and 2-level envelopes. Examples of such 3-level annealing schedules ($L = 3$, $s_1 = 64$, $s_2 = 4$, $s_3 = 4$) are: a top-level (level 3) quench (*InnerNum* = 1), a fixed anneal at level 2 and level 1 (*InnerNum* = 1) and a flat fixed anneal (*InnerNum* from 0.5 to 1); a top-level fixed anneal (*InnerNum* = 0.1), a fixed anneal at level 2 and level 1 (*InnerNum* = 1), and a flat automatic anneal (*InnerNum* = 0.5). Once again, although these sets of placement parameters resulted in the best quality-time trade-off for our ultra-fast tool, there are alternative schedules that achieve results that are approximately as good.

In order to determine the best value of the fanout threshold, *maxfan* (the value of fanout above which the nets are ignored), we performed an experiment with $L = 1$ and $s_1 = 64$ and varied the fanout threshold. Figure 3.10 is a scatter plot of quality versus run time, averaged over 20 circuits, for various values of fanout threshold and annealing schedules. The circular dots represent the quality when no nets are ignored, and the other points show the quality when more nets are ignored -- for fanout thresholds ranging from 1000 to 1. It is evident that excessively low fanout thresholds eliminate far too much placement information from the circuit, hence the area degradation is huge. However, when nets with fanout over 100 are ignored, we save a few seconds of placement time with almost no degradation in quality.

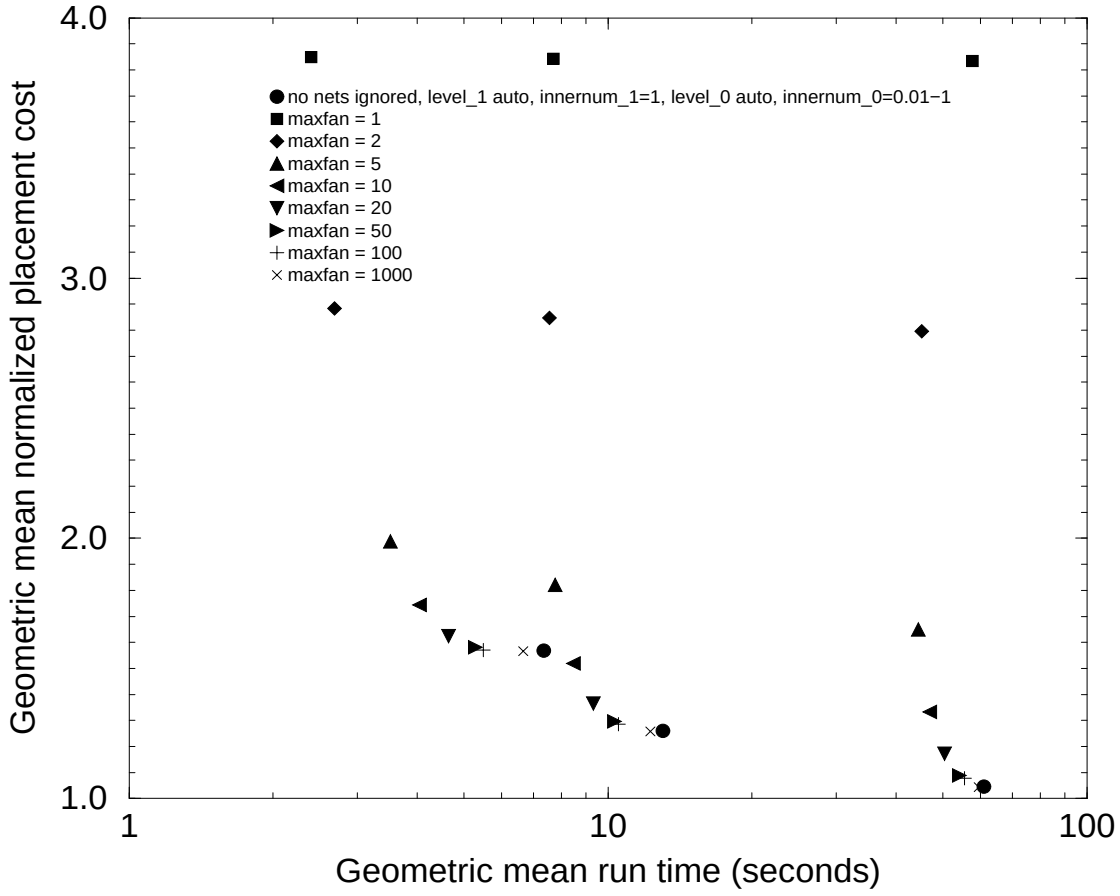

Figure 3.10: Placement quality-time plot (20 circuit average) for ultra-fast placement tool using different fanout thresholds above which nets are ignored on circuits with fixed cluster and placement parameters.

3.5 Summary

In this chapter, the details of the ultra-fast placement algorithm were described. The algorithm begins with a bottom-up, multiple-level clustering phase to simplify the placement problem. It is followed by a two-step hierarchical placement algorithm: at each level of the hierarchy, an initial constructive placement of the clusters takes place, followed by a simulated-annealing-based refinement. The algorithms were shown to have linear time complexity if nets above a certain fanout threshold were ignored. The key tunable parameters of both the clustering (number of cluster levels, cluster sizes) and placement (starting temperature, exit temperature, temperature update factor, number of moves per temperature, fanout threshold) algorithms were presented. The search for the envelope parameters that provide the best quality-time trade-off for the ultra-fast tool was described, and the specific clustering and placement parameters that form the envelope were furnished.

In the next chapter, we will establish the baseline placement results to which we will compare the placements produced by our ultra-fast tool. This will be followed by a comprehensive evaluation of the quality-time performance of the ultra-fast tool and a description and results of the types of prediction information that are conveyed by the tool.

Chapter 4

Experimental Results

In this chapter, we compare the new ultra-fast placement tool to an existing and known high-quality placement tool, VPR [Betz97]. We also discuss the fast prediction of high-quality wirelength and the fast prediction of the placement quality versus run time relationship of a given circuit. We first describe the FPGA architecture used in the experimental comparisons and the overall CAD flow.

4.1 Target FPGA Architecture

For our placement experiments, we use an island-style FPGA with a logic block that contains a single 4-input lookup table (4-LUT) and a single D flip-flop. This simple and most basic FPGA architecture is illustrated in Figure 4.1. Each block has 6 pins: 4 inputs, 1 output, and 1 clock. We will assume the FPGA has dedicated resources for routing the clock, reset, and other global nets. We also assume an I/O pad pitch-to-logic block ratio of 2.

4.2 Benchmark Circuits and CAD Flow

We have collected 20 large circuits from a variety of sources: 14 of the circuits originate from the MCNC suite [Yang91], one comes from the RAW suite [Babb97], one is a synthetic circuit generated by GEN [Hutt97], and the remaining four are designs created for the Transmogrifier-2 Rapid Prototyping System [Lewi97] at the University of Toronto [Ye99] [Hame98]

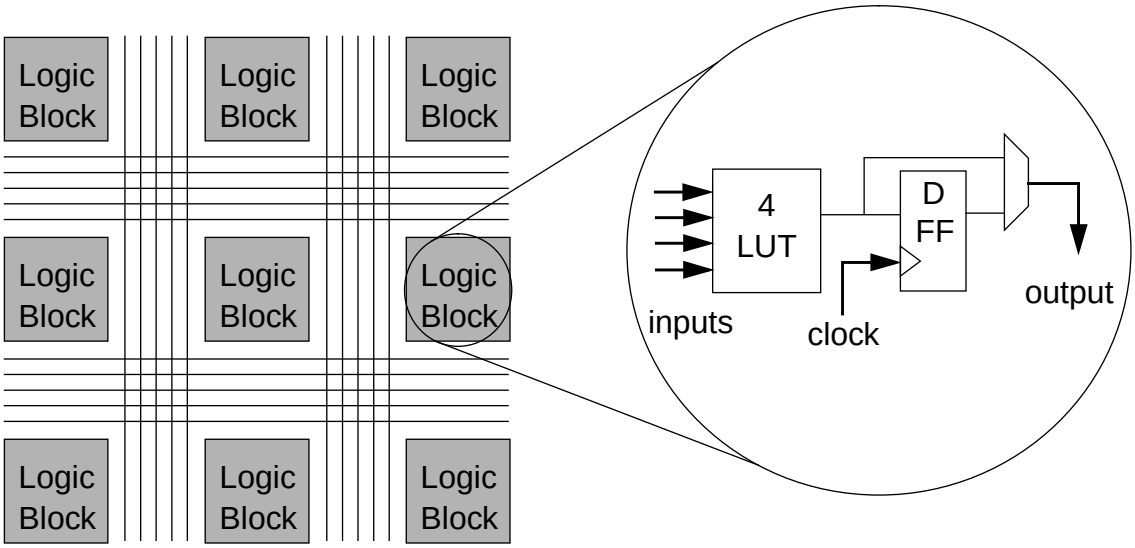

Figure 4.1: Island-style FPGA architecture and basic logic block contents.

[Leve98]. Each circuit was optimized using SIS [Sent92], and technology mapped into 4-LUTs using Flowmap and Flowpack [Cong94]. VPACK [Betz97] was used to pack the netlists of 4-LUTs and flip-flops into basic logic blocks. The sizes of the 20 benchmark circuits range from 3000 to 20,000 logic blocks.

We have implemented our fast placement tool within the framework of VPR. Since it is a good estimator of wiring area, we use the bounding box wirelength of all nets in the circuit to compare the quality of placement of each circuit from each tool. We measure only the time used to perform clustering and placement, and do not include the initial input file reading time and parsing (this is no more than 5 seconds for the largest circuit). All experiments are run on a 300 MHz Sun UltraSPARC workstation.

4.3 Basis of Comparison

We use the pure simulated annealing-based VPR as the basis for comparison to our new placement algorithm. In order to compare to the quality-time trade-off curve for VPR, we needed to vary the schedule parameters for VPR itself, in a manner similar to that described in Chapter 3 for our tool, to determine how well we can do with respect to placement quality and time using VPR alone.

To obtain the envelope of the quality-time curve for VPR, we varied each of the key simulated annealing parameters -- initial temperature (T_0), exit temperature (T_f), temperature update factor (α), and scaling factor for the number of moves to attempt per temperature (*InnerNum*). We used the three types of schedules described in Section 3.3.2: (1) an automatic annealing schedule (T_0, T_f, and α calculated dynamically and adjusted depending upon the quality of the placement) with variable *InnerNum*; (2) a quench (greedy heuristic) with variable *InnerNum*; (3) a fixed annealing schedule, in which we either sweep T_0, keeping T_f, α, and *InnerNum* constant, or sweep α, keeping T_0, T_f, and *InnerNum* constant.

We ran each unique annealing schedule on all 20 circuits, recorded the run time and wirelength, and normalized the wirelength for each run on a given circuit to that achieved by VPR when run under its "-fast" option on that same circuit. This specific VPR option is similar to its default parameters that are tuned to generate high-quality placements, except that one-tenth of the configurations are explored at each temperature (the scaling factor *InnerNum* is 1 under "-fast", and 10 by default). Typically, this increases the placement cost by at most 10%, but with a factor of 10 speedup in placement time. Essentially, it is a very high quality placement that is obtained in a reasonable amount of time. The normalized wirelengths and run times were then averaged geometrically across all 20 circuits to give a single (placement time, placement cost) pair for each distinct annealing schedule. It is from these experiments that we determined the envelope of the best VPR annealer parameters to specify across all 20 circuits.

The envelope containing the annealing schedules that produced the best quality-time tradeoff consisted of parts of three types of schedules with variable *InnerNum* (between 0 and 100): a quench, an anneal with $T_0 = 1$, $T_f = 0.01$, and $\alpha = 0.8$, and an automatic anneal with dynamically-updated T_0, T_f, and α. Figure 4.2 illustrates the geometric mean normalized placement cost (bounding-box wirelength) versus geometric mean run time across all 20 benchmark circuits for the three schedules that form the quality-time envelope for VPR.

There is not much difference in wirelength and run time among the schedules for extremely short run times (< 3 seconds). We observe that for run times in the 10 to 100 second range, there is ample room for improvement; an average of 80-100% extra wiring area is likely unacceptable to a circuit designer even within 10 seconds of placement time. The figure also shows that as more

time is expended on placement and more configurations are evaluated, the benefit of probabilistic hill climbing becomes readily apparent. The curve representing a greedy annealing schedule (quench) shows that placement solutions in this range cannot escape local minima, and it diverges from those schedules that accept some bad moves in order to reach global minima.

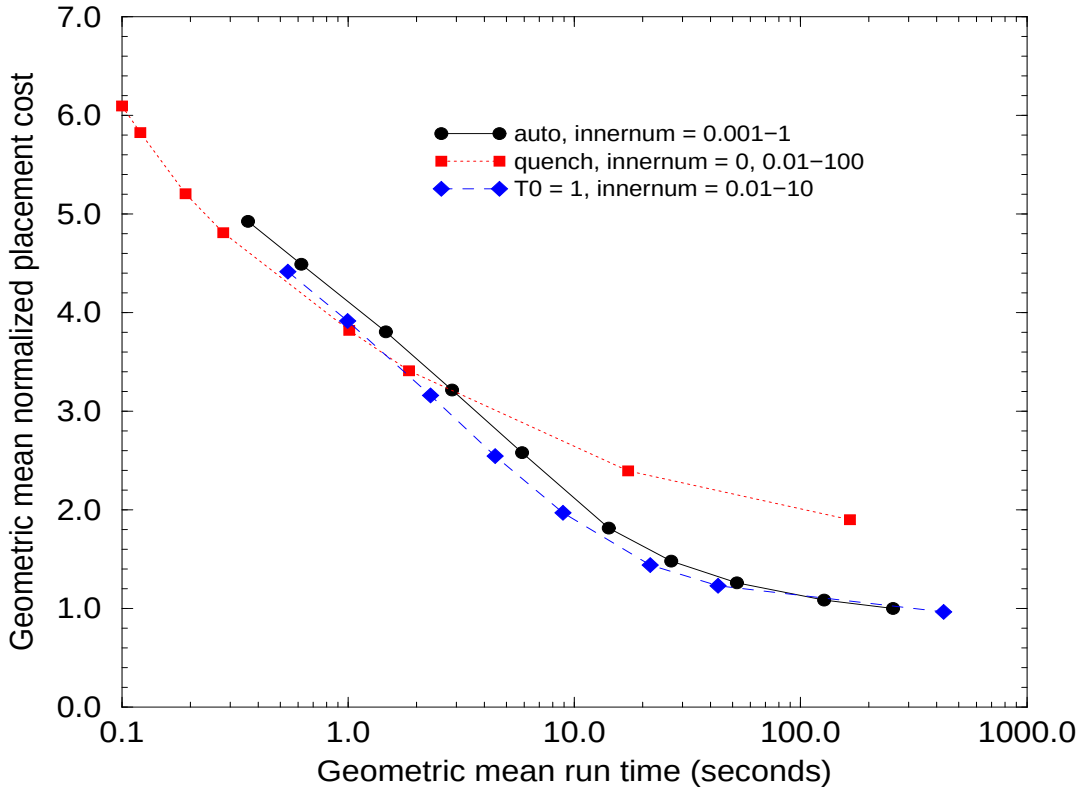

Figure 4.2: VPR placement quality-time trade-off (20 circuit average) using annealing schedules that form the envelope.

4.4 Comparisons Between Ultra-Fast Algorithm and VPR

A head-to-head comparison between the ultra-fast placement tool and VPR is possible by first running each set of placement parameters that lies on the envelope of the respective tool on every circuit in the benchmark suite. Then, the placement quality results are normalized to those obtained by running VPR under its “-fast” option (described above), and the geometric mean placement cost and mean run time are calculated for each set of parameters across all circuits.

Figure 4.3 is a plot of both the best VPR quality-time envelope and the new ultra-fast placement tool quality-time envelope. It indicates that the ultra-fast placement tool has a clear advantage for both short run times (10 seconds or less) and medium run times (from 10 to 100 seconds). In 10 seconds, our placement tool requires only 30% more wirelength on average (than the best

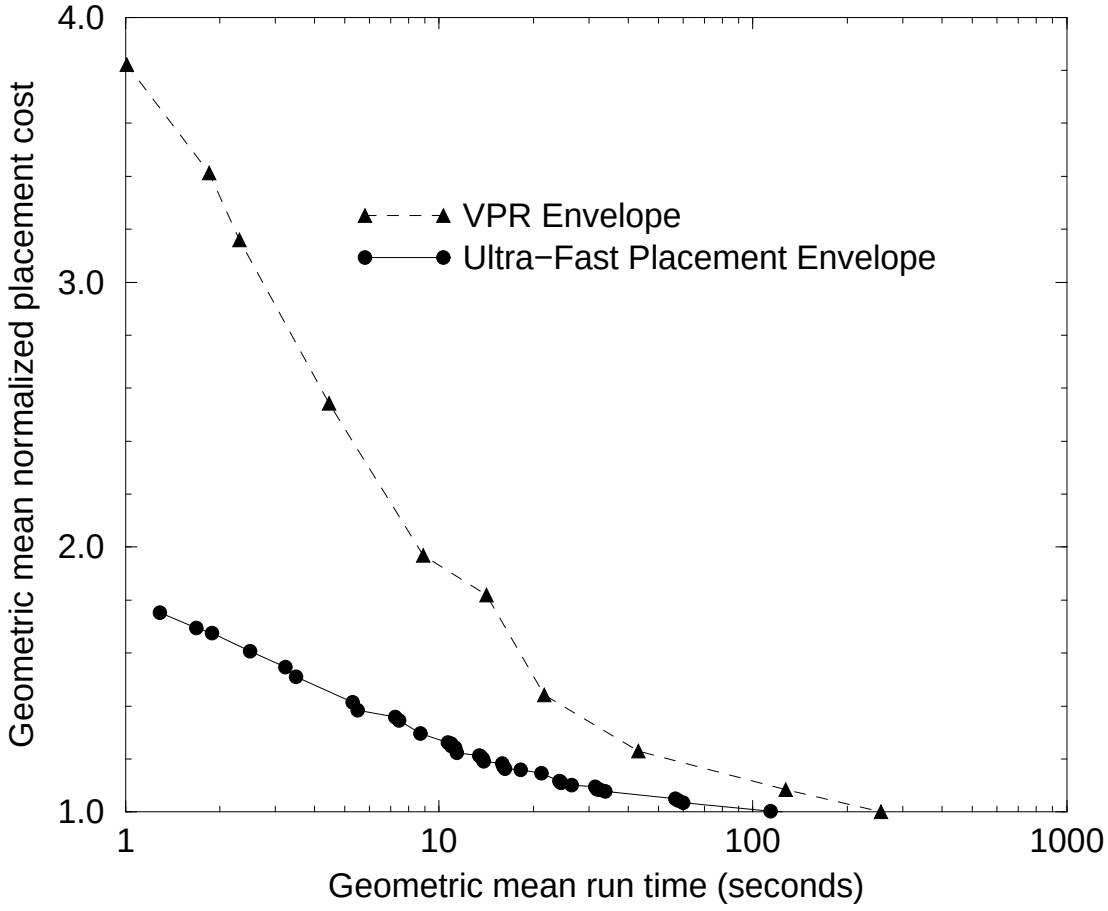

Figure 4.3: Placement quality-time envelope curves (20 circuit average) for VPR and new ultra-fast placement tool.

possible placement using VPR), while VPR requires at least 80% more wirelength on average. In just 3.5 seconds, our ultra-fast tool generates placements requiring approximately 50% more wirelength on average, whereas placements produced by VPR in the same amount of time require between 2.5 and 3 times the wirelength on average. Furthermore, while VPR can achieve an average area penalty of 10% in over 100 seconds, our placement tool can attain this level in less than 30 seconds. If we allowed our placement tool to run without a compile time restriction, it would produce placements that would be very nearly what VPR can achieve, since both tools are based

on similar implementations of simulated annealing. This is evident from the graph: within 60 seconds on average, the ultra-fast placement tool yields an average wirelength that is within 5% of that obtained via VPR's high-quality anneal. Within 115 seconds on average, the ultra-fast tool generates placements that are of the same quality as the high-quality anneals from VPR. Figure 4.3 also demonstrates that by manipulating the parameters of the fast placement tool, we can realize a smooth trade-off between placement quality and execution time.

Figure 4.4 shows the envelope curves from the two placement tools and highlights the point at which the curves intersect, which is at approximately 250 seconds, the average amount of time required by VPR to produce high-quality placements of the circuits in the benchmark suite. Figure 4.5 shows the envelope curves from the two placement tools using a linear scale for the time axis, focusing on short and medium run times.

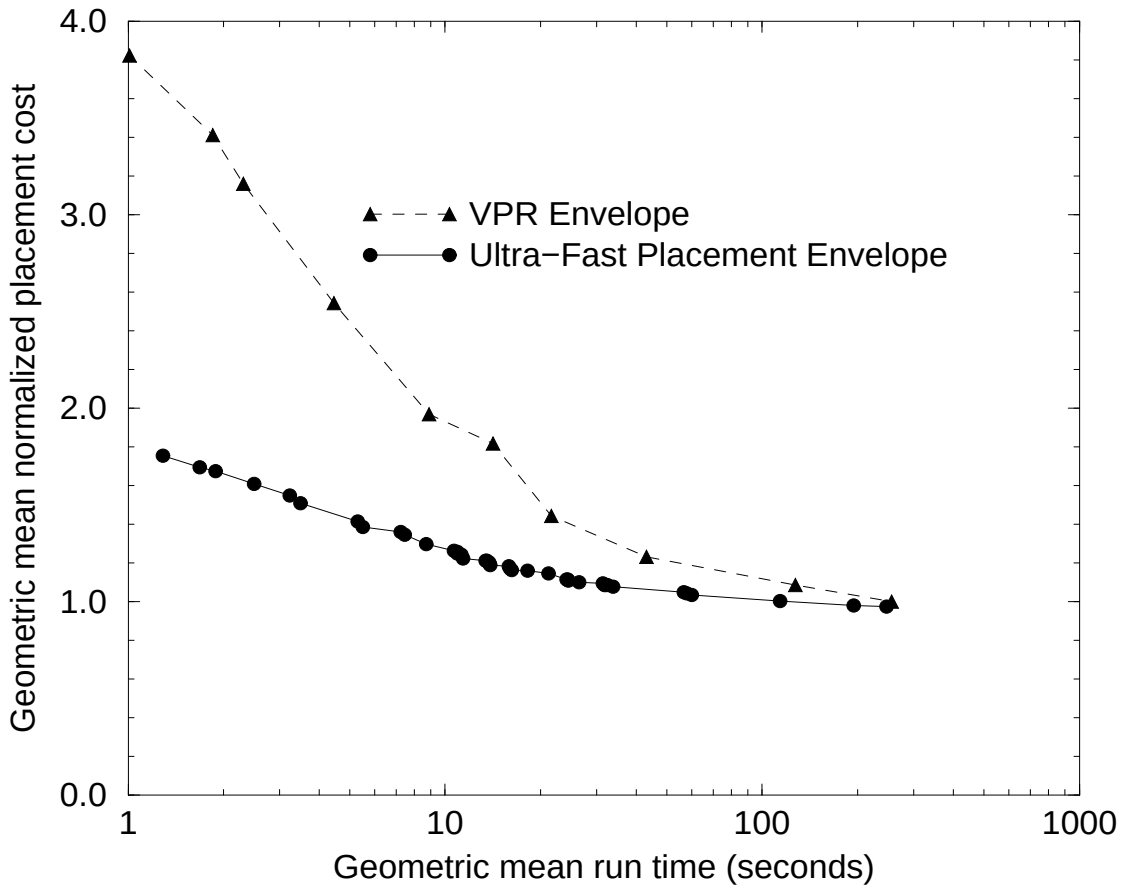

Figure 4.4: Placement quality-time envelope curves (20 circuit average) for VPR and new ultra-fast placement tool, highlighting the point at which the curves meet.

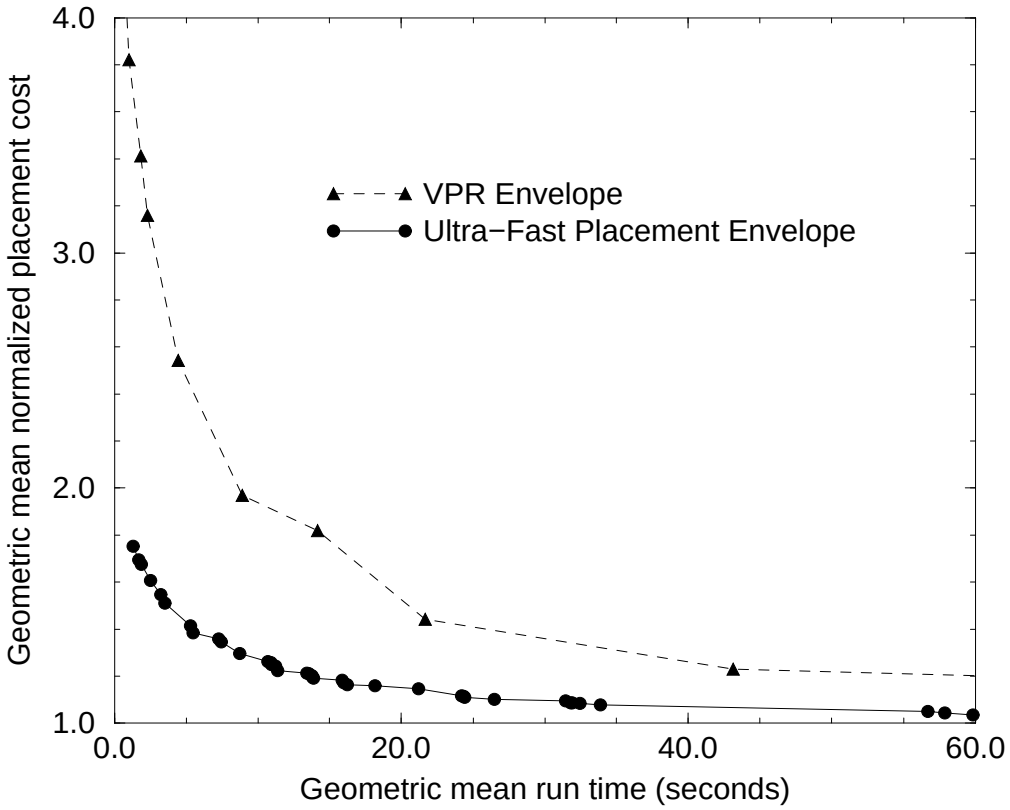

Figure 4.5: Placement quality versus time envelope curves (20 circuit average) for VPR and new ultra-fast placement tool (linear scale).

Table 4.1 provides a more detailed comparison between VPR and the ultra-fast placement tool with one particular set of parameters: $L = 2$ levels of clustering with cluster sizes $s_1 = 64$ and $s_2 = 4$, with the top-level and level-1 annealing schedules being quenches (with *InnerNum* = 10), a flat anneal with $T_0 = 0.1$ (*InnerNum* = 0.5), and nets above fanout 100 ignored (*maxfan* = 100). From Figure 4.3, we note that the mean run time for this schedule across our 20-circuit benchmark suite is 11.4 seconds with a mean normalized placement cost of 1.22. It is difficult to find directly comparable run times between the two tools; we then select a schedule from the VPR envelope that has a run time that is as close as possible (mean run time = 14.2 seconds; mean normalized placement cost = 1.82). The columns of Table 4.1 give the circuit name, the circuit size in number of logic blocks and nets, the run time and normalized placement cost obtained using our fast tool, the comparable data using VPR, and the percentage reduction in placement cost due to the ultra-fast tool compared to VPR. The table shows that the ultra-fast placer wins in a compari-

son with VPR for every circuit in our suite, posting a superior wirelength in a significantly shorter run time. Note that for this particular set of ultra-fast placement parameters, the reduction in wirelength compared to VPR ranges from 13% to 51%, with an average reduction of 33% in 20% less time.

Table 4.2 and Table 4.3 provide additional comparisons between our ultra-fast placement tool and VPR for very short and somewhat longer run times, respectively. In both cases, the ultra-fast placer provides lower wirelength in the same amount of placement time or less for every circuit in the test suite. Naturally, as more time is devoted to placing the circuits, the difference in wirelength results between the ultra-fast tool and VPR becomes smaller. For the short run times (geometric mean run time of 5.5 seconds for the ultra-fast tool), the average reduction in wirelength compared to VPR (8.9 seconds per run on average) was approximately 30% in 37% less time. For the longer run times (geometric mean run time of 24.4 seconds for the ultra-fast tool), the average reduction in wirelength compared to VPR (26.6 seconds per run on average) was over 18% in 7% less time.

The true measure of quality of a given placement is whether or not it can be successfully routed on the target FPGA. Although we have not attempted to route any of the ultra-fast placements, [Swar98b] has shown that wirelength and routability correlate extremely well. Therefore, we are satisfied that our ultra-fast placements are superior to those produced by VPR, based solely on wirelength for the range of compile times of interest.

Table 4.1: Comparison between ultra-fast placement tool and VPR for 20 benchmark circuits. One set of placement parameters was employed for each tool such that their run times were close and they were part of the quality-time envelope for their respective tools.

Circuit	# Logic Blocks	# Nets	Ultra-Fast Placement		VPR		Ultra-Fast % Reduction in Placement Cost
			Run Time (s)	Normalized Placement Cost	Run Time (s)	Normalized Placement Cost	
clma	8383	8444	21.71	1.20	29.79	1.83	34.4
spla	3690	3706	6.37	1.22	7.26	1.63	25.2
s38584.1	6447	6484	14.55	1.29	18.35	2.33	44.6
s38417	6406	6434	13.33	1.22	16.88	1.87	34.8
frisc	3556	3575	6.15	1.21	7.04	1.63	25.8
pdc	4575	4591	8.43	1.20	10.35	1.52	21.1
ex1010	4598	4608	7.69	1.23	10.53	1.67	26.3
elliptic	3604	3734	6.05	1.15	7.16	1.60	28.1
beast20k	19600	20000	108.34	1.16	128.10	1.34	13.4
bubble sort	12293	12311	41.08	1.29	53.65	2.14	39.7
fir16	6975	6994	16.31	1.32	20.86	2.19	39.7
iir16	3739	3773	6.93	1.15	7.57	2.16	46.8
mac64	4307	4374	8.94	1.19	10.19	1.69	29.6
ochip64	4083	4101	6.85	1.13	10.63	2.30	50.9
ralu32	3662	3184	5.96	1.25	6.69	1.66	24.7
spsdes	3363	3366	5.22	1.21	6.47	1.70	28.8
des_fm	4786	4791	9.25	1.34	13.25	1.69	20.7
des_sis	5351	5356	11.12	1.24	14.14	1.67	25.7
wood	7432	7524	17.54	1.24	22.15	2.00	38.0
marb	5535	5639	11.61	1.26	13.72	2.15	41.4
Geometric Average			11.37	1.22	14.17	1.82	33.0
Arithmetic Average			16.67	1.22	20.74	1.84	33.7

Table 4.2: Comparison between ultra-fast placement tool and VPR across 20 circuits for very short run times.

Circuit	# Logic Blocks	# Nets	Ultra-Fast Placement		VPR		Ultra-Fast % Reduction in Placement Cost
			Run Time (s)	Normalized Placement Cost	Run Time (s)	Normalized Placement Cost	
clma	8383	8444	10.13	1.31	16.95	1.84	28.8
spla	3690	3706	3.21	1.33	5.40	1.71	22.2
s38584.1	6447	6484	6.82	1.46	12.14	2.45	40.4
s38417	6406	6434	6.08	1.35	10.81	2.00	32.5
frisc	3556	3575	3.19	1.35	4.45	1.72	21.5
pdc	4575	4591	4.35	1.31	6.50	1.60	18.1
ex1010	4598	4608	3.81	1.47	6.28	1.70	13.5
elliptic	3604	3734	3.08	1.25	5.12	1.66	24.7
beast20k	19600	20000	55.75	1.30	76.59	1.65	21.2
bubble sort	12293	12311	18.39	1.49	32.33	2.25	33.8
fir16	6975	6994	7.63	1.48	13.60	2.38	37.8
iir16	3739	3773	3.32	1.31	5.07	2.36	44.5
mac64	4307	4374	4.54	1.38	6.32	1.97	29.9
ochip64	4083	4101	3.18	1.42	6.32	2.59	45.2
ralu32	3662	3184	2.92	1.37	4.16	1.81	24.3
spsdes	3363	3366	2.53	1.35	4.11	1.80	25.0
des_fm	4786	4791	4.36	1.55	6.78	1.82	14.8
des_sis	5351	5356	5.17	1.47	8.15	1.83	19.7
wood	7432	7524	8.11	1.36	15.19	2.21	38.5
marb	5535	5639	5.51	1.42	8.87	2.50	43.2
Geometric Average			5.49	1.39	8.90	1.97	29.4
Arithmetic Average			8.10	1.39	12.76	1.99	30.2

Table 4.3: Comparison between ultra-fast placement tool and VPR across 20 circuits for longer run times.

Circuit	# Logic Blocks	# Nets	Ultra-Fast Placement		VPR		Ultra-Fast % Reduction in Placement Cost
			Run Time (s)	Normalized Placement Cost	Run Time (s)	Normalized Placement Cost	
clma	8383	8444	51.78	1.07	54.90	1.31	18.3
spla	3690	3706	14.09	1.11	15.26	1.26	11.9
s38584.1	6447	6484	35.58	1.16	34.34	1.43	18.9
s38417	6406	6434	28.88	1.10	33.97	1.34	17.9
frisc	3556	3575	13.12	1.12	13.82	1.35	17.0
pdc	4575	4591	19.08	1.07	20.48	1.29	17.1
ex1010	4598	4608	17.87	1.08	19.60	1.15	6.1
elliptic	3604	3734	14.54	1.09	13.82	1.35	19.3
beast20k	19600	20000	212.87	1.06	221.51	1.06	0
bubble sort	12293	12311	85.82	1.17	98.61	1.45	19.3
fir16	6975	6994	35.25	1.13	40.96	1.44	21.5
iir16	3739	3773	12.86	1.08	15.03	1.59	32.1
mac64	4307	4374	17.01	1.06	18.97	1.27	16.5
ochip64	4083	4101	15.98	1.00	17.03	1.63	38.7
ralu32	3662	3184	11.71	1.12	12.95	1.34	16.4
spsdes	3363	3366	12.07	1.10	12.68	1.34	17.9
des_fm	4786	4791	17.31	1.22	21.26	1.29	5.4
des_sis	5351	5356	20.86	1.15	24.54	1.34	14.2
wood	7432	7524	43.55	1.17	41.09	1.52	23.0
marb	5535	5639	23.04	1.16	27.17	1.69	31.4
Geometric Average			24.41	1.11	26.59	1.36	18.4
Arithmetic Average			35.16	1.11	37.90	1.37	19.0

4.5 Wirelength Estimation and Accuracy

One way to use a fast placement tool, even if the user is not interested in sacrificing any final circuit quality, is to use it as a routability estimator for a given netlist. Swartz et al. [Swar98a] show how to predict if a circuit will route on a given FPGA, given the wirelength of the placement of a circuit and the number of tracks per channel in the target FPGA. The drawback of their approach is that the placement must be known. We propose that our fast placement algorithm be used to obtain very fast and accurate estimates of the final *best* placement wirelength. The idea is that we can run the fast placement tool in one of its very fastest modes, measure the wirelength of that placement, and then estimate the best attainable wirelength by decreasing the wirelength by the typical amount that the fast mode is usually worse than the best mode. The quality of the result depends on the consistency of difference in wirelength between the fast mode and the best mode. This can be measured by determining how much the normalized placement cost for each circuit, in the fast mode, varies from the mean normalized placement cost across all circuits.

Figure 4.6 is a plot of the average difference of each circuit's normalized wirelength from the mean over all circuits versus different run times of the ultra-fast placement tool obtained from the quality-time envelope parameters. (To obtain this graph, we calculate the absolute difference between the geometric mean normalized placement cost across all circuits and the actual normalized placement cost for each of the 20 benchmarks for each set of fast placement parameters. We then compute the arithmetic mean of these differences (and call it mean absolute error) and plot it versus the geometric mean run time that was obtained for the set of circuits for this set of parameters.)

Figure 4.6 shows that, as we would expect, longer compile times produce more accurate wirelength estimates. Impressively, even placements in short run times result in accurate estimates -- for example, an average 10 second run time results in an average absolute error in normalized placement cost of less than 5%.

We can therefore use the fast placement wirelength as an accurate estimator of the final best placement wirelength.

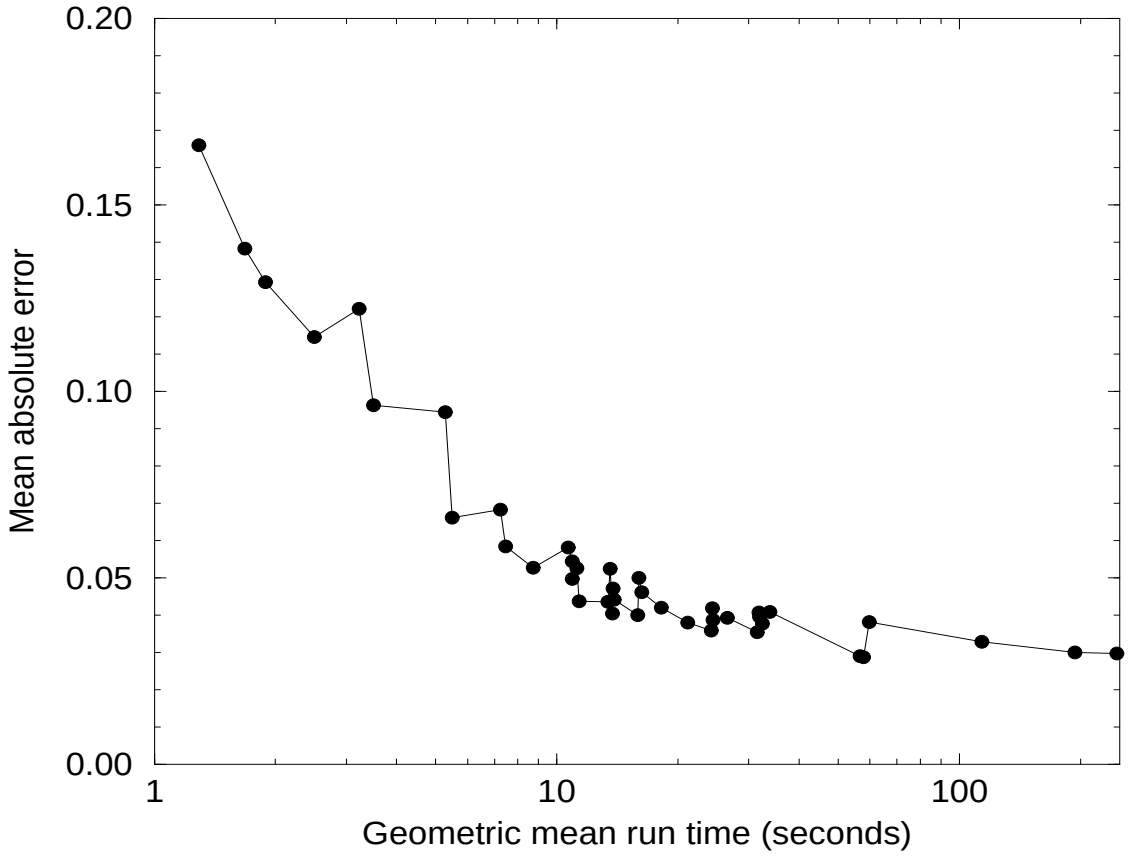

Figure 4.6: Mean absolute difference in wirelength (between mean wirelength and individual circuit results) versus mean run time for parameters forming ultra-fast placement tool envelope.

Table 4.4 illustrates an example of fast wirelength estimation for each of the circuits in our benchmark suite. We used the same set of ultra-fast placement parameters as that used to generate the data in Table 4.1, and recorded both the run time and raw wirelength result in each case. From the envelope curve in Figure 4.3, we know the mean normalized wirelength for this set of parameters across all circuits to be 1.22, or 22% larger than the highest-quality wirelength attainable by VPR. Figure 4.6 indicates that the mean absolute error for that set of parameters is 0.044 (4.4%).

Our pessimistic prediction of high quality wirelength can be written as:

$$Wirelength_{predicted} = \beta \cdot Wirelength_{ultra-fast} \quad (4.1)$$

$$\beta = \frac{1}{\overline{Wirelength_{normalized}} - \overline{AbsoluteError}} \quad (4.2)$$

The predicted high-quality wirelength for a given circuit is proportional to the wirelength obtained from the ultra-fast placement tool. The scaling factor, β, is composed of the difference between the normalized wirelength for the set of ultra-fast parameters chosen (geometrically averaged across all circuits) and the previously described mean absolute error in normalized wirelength for the same set of placement parameters.

For the example in Table 4.4, Predicted Wirelength = Ultra-Fast Wirelength / (1.22-0.044). We use this formula to compute a wirelength estimate for each circuit based on the fast placement wirelength result, and compare it to the known high-quality wirelength for each circuit from VPR. For 16 of the circuits, our pessimistic estimate is between 0.89% and 13.75% higher than the actual high-quality wirelength, and in only two cases is the error greater than 10%. In four cases, the estimator was not pessimistic enough, predicting a wirelength that was between 1.71% and 3.93% less than the actual high-quality wirelength. Overall, the average absolute error of the wirelength estimator was 4.91% for the set of placement parameters that yielded a mean run time of just over 11 seconds.

Another example of fast wirelength prediction for each circuit in the benchmark suite is provided in Table 4.5. Here, the placement data from Table 4.2 was used. The average ultra-fast placement time was just 5.5 seconds, the mean normalized wirelength was 1.39, and the mean absolute error was 0.066 (from Figure 4.6). The predicted wirelengths were between 0.94% and 17.20% higher than the actual VPR high-quality wirelengths, with six estimates over 10%. There were five instances where the pessimistic estimate underpredicted the actual wirelength, with none greater than 5%. Naturally, with such a short average run time, the predictions based on the ultra-fast placements tend to be less accurate than those obtained during the longer placement times in Table 4.4, but the average absolute error is still merely 6%.

Table 4.4: Quality of wirelength prediction capability of ultra-fast placement tool using placement data from Table 4.1 (mean run time = 11.4 seconds).

Circuit	Run Time (s)	Ultra-Fast Placement Wirelength	Predicted High-Quality Wirelength	VPR Actual High-Quality Wirelength	% Error
clma	21.71	1786	1514	1491	+1.55
spla	6.37	763	646	625	+3.37
s38584.1	14.55	901	763	696	+9.70
s38417	13.33	883	748	726	+3.12
frisc	6.15	685	580	566	+2.58
pdc	8.43	1096	929	917	+1.35
ex1010	7.69	843	715	688	+3.84
elliptic	6.05	588	499	513	-2.75
beast20k	108.34	7522	6374	6485	-1.71
bubble sort	41.08	1632	1383	1262	+9.57
fir16	16.31	1108	939	841	+11.62
iir16	6.93	464	393	404	-2.63
mac64	8.94	660	560	555	+0.89
ochip64	6.85	350	297	309	-3.93
ralu32	5.96	506	429	405	+5.85
spsdes	5.22	527	447	434	+2.88
des_fm	9.25	857	727	639	+13.75
des_sis	11.12	826	700	665	+5.33
wood	17.54	1085	920	873	+5.32
marb	11.61	617	523	492	+6.39
Arithmetic Average Absolute Error					4.91

Table 4.5: Quality of wirelength prediction capability of ultra-fast placement tool using placement data from Table 4.2 (mean run time = 5.5 seconds).

Circuit	Run Time (s)	Ultra-Fast Placement Wirelength	Predicted High-Quality Wirelength	VPR Actual High-Quality Wirelength	% Error
clma	10.13	1958	1485	1491	-0.40
spla	3.21	832	631	625	+0.94
s38584.1	6.82	1018	772	696	+10.88
s38417	6.08	981	744	726	+2.52
frisc	3.19	763	578	566	+2.18
pdc	4.35	1202	911	917	-0.63
ex1010	3.81	1014	769	688	+11.76
elliptic	3.08	643	488	513	-4.87
beast20k	55.75	8446	6404	6485	-1.25
bubble sort	18.39	1881	1426	1262	+12.99
fir16	7.63	1248	946	841	+12.53
iir16	3.32	527	400	404	-0.93
mac64	4.54	766	581	555	+4.70
ochip64	3.18	438	332	309	+7.60
ralu32	2.92	557	422	405	+4.20
spsdes	2.53	584	443	434	+1.98
des_fm	4.36	987	749	639	+17.20
des_sis	5.17	976	740	665	+11.29
wood	8.11	1188	901	873	+3.16
marb	5.51	698	530	492	+7.69
Arithmetic Average Absolute Error					5.98

4.6 Practical Usage of Ultra-Fast Placement

We have presented results obtained from running an ultra-fast placement tool against a known pure simulated-annealing-based placement tool, and shown how it can be used for fast high-quality wirelength estimation. In this section, we discuss how such a placement package can be used in a practical setting, specify the interface to the user, and describe what the tool does when the user wants a circuit to be placed quickly.

In order for ultra-fast placement to be accessible to the user, the tool should provide two key features:

- Immediate feedback to the user upon reading in the circuit to be placed; this feedback consists of the expected placement quality versus compile time curve displayed in a window on the user's screen.
- Given either a compile time restriction *or* a wirelength restriction, specified by the user at the command-line, the tool should automatically determine the appropriate parameters (clustering and placement) and execute the ultra-fast placement algorithm.

The predictor of the quality-time envelope curve for a given circuit was developed using the fast placement data in Table 4.1, the 20-circuit average quality-time envelope, and the size of the circuits in the benchmark suite, in logic blocks. Adapting the average quality-time envelope to generate the envelope for a specific circuit requires more than just a simple scaling based on circuit size.

First, the ultra-fast placement times of all 20 benchmark circuits for the chosen set of placement parameters are plotted as a function of circuit size. A curve fit is then performed. Using the fitted function and the size of the circuit to be placed, the estimated run time for the envelope parameters invoked in Table 4.1 is calculated for the circuit in question. We call this the estimated 10-second run time. Then, to determine each remaining estimated run time on the envelope for the given circuit, we simply scale this "10-second run time" point on the estimated envelope by the ratio between the corresponding run time on the average envelope and the "10-second run time" point on the average envelope. In each case, the corresponding estimated normalized wirelength is taken to be the geometric mean normalized wirelength obtained across all circuits for the same set of envelope parameters. Examples of this prediction scheme are shown for three different circuits in Figure 4.7, Figure 4.8, and Figure 4.9, along with the actual fast placer quality-time envelopes.

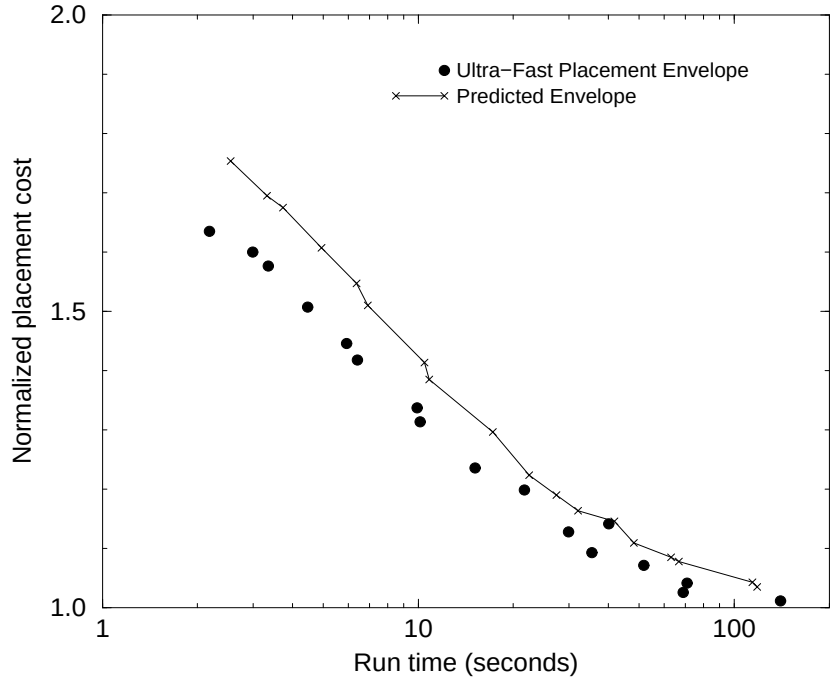

Figure 4.7: Comparison of predicted ultra-fast placement quality versus time envelope with actual envelope for MCNC circuit **clma** (8383 logic blocks).

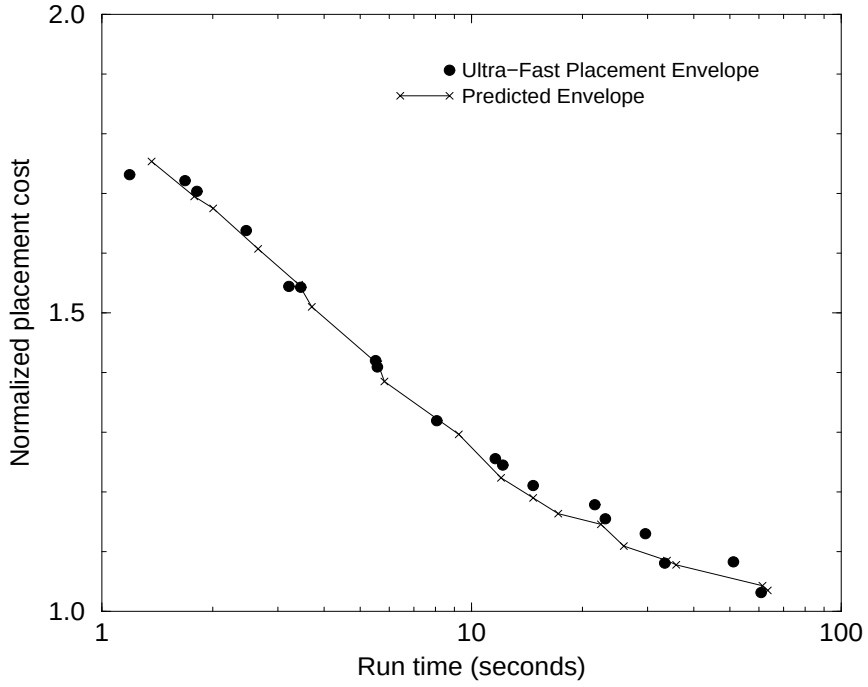

Figure 4.8: Comparison of predicted ultra-fast placement envelope with actual envelope for circuit **marb** (5535 logic blocks).

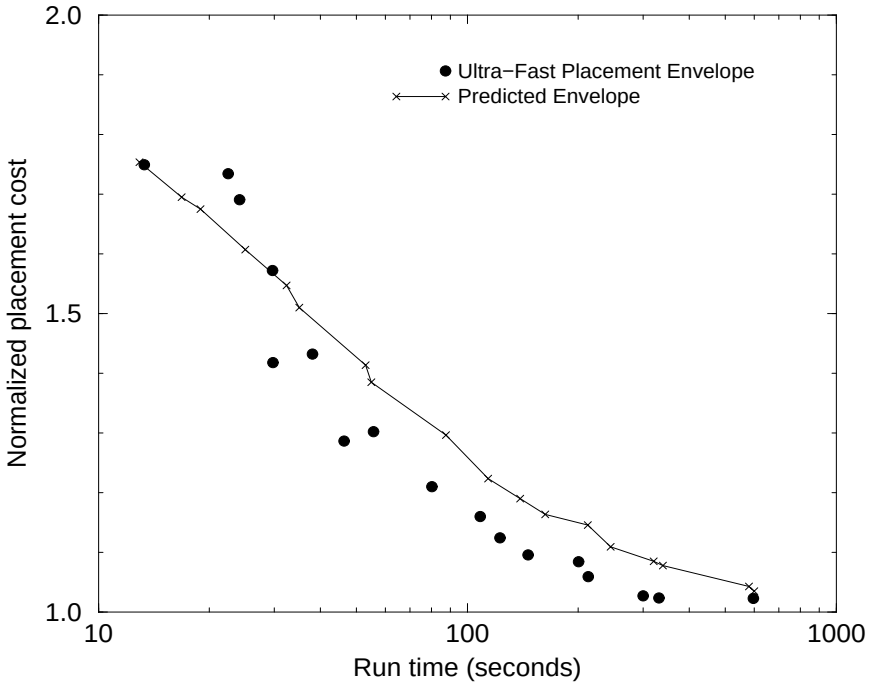

Figure 4.9: Comparison of predicted ultra-fast placement envelope with actual envelope for synthetic circuit **beast20k** (19600 logic blocks).

Note that the prediction scheme tracks the actual fast placer envelope quite closely in each case. This is a simplistic scheme based only on circuit size, and one that performs some amount of analysis of the circuit structure (fanout distribution) should perform even better.

Once the predicted quality-time envelope has been calculated and presented to the user (via a pop-up window), this information is stored to assist in the automatic generation of fast placement parameters based on the circuit size and either a given area or compile time restriction. For a compile time restriction, a search is performed of the estimated envelope run times to determine the closest schedule to the desired compile time, and the associated clustering and placement parameters are retrieved via table lookup. Then, a simple scaling of the *InnerNum* parameter (number of moves per temperature) at the flat level is done, based on the difference between the estimated envelope run time and the desired run time restriction. With all the fast placer parameters now specified, the algorithm is invoked, and an ultra-fast placement is produced. For an area restriction, a similar procedure is followed, except that the area restrictions are first translated into compile time restrictions based on the predicted quality-time envelope.

4.7 Summary

In this chapter, we showed how the envelope curve for the quality-time trade-off for a high-quality simulated-annealing-based placement tool, VPR, was determined. This was used as the metric of comparison for our ultra-fast placer for both quality and time, and also for fast prediction of high-quality wirelength. We then showed a detailed comparison of the ultra-fast placement tool with VPR, and the results indicate that in the compile time regions of interest, the ultra-fast placer provides superior wirelength in a shorter time for every circuit in the benchmark suite. We further described the method to perform fast high-quality wirelength prediction, and presented some impressive results using this scheme. Finally, in order to properly allow the user to interact with the ultra-fast placement tool, two features are incorporated into the tool and discussed -- prediction of the quality-time relationship for an unknown circuit given its size and automatic generation of fast placer parameters given either a compile time or quality restriction for that circuit.

Chapter 5

Conclusions and Future Work

5.1 Conclusions and Contributions

We have observed that when mapping circuits to an FPGA, the time devoted to placement and routing dominates the synthesis process. With advances in process technology, FPGA device capacities will continue to grow, and the size and complexity of circuits being mapped to them will increase accordingly. Therefore, existing CAD algorithms will need to adapt to ensure that the placement and routing times for such circuits and devices will not overwhelm the user by taking hours or days to complete. It is our belief that some FPGA users are willing to give up quality of the mapped circuit -- accepting a circuit that occupies more area on a given FPGA or requires a larger FPGA -- in exchange for obtaining that result very quickly.

The first contribution of this work was the exploration of the parameter space at our disposal through an existing pure simulated-annealing-based tool, VPR [Betz97] [Betz98], that is known to produce high-quality placements. In the course of our exploration of this parameter space that defines all possible annealing schedules (starting temperature, exit temperature, temperature reduction factor, and number of moves to attempt at each temperature), we showed the best quality-time trade-off that is achievable. While high-quality placements are attainable in a few hundred seconds, a designer must be willing to accept 80-100% more area on average for a given large circuit if a placement is desired within 10 seconds.

We restricted our focus in this thesis to the placement phase of layout synthesis for FPGA circuits and demonstrated that an ultra-fast placement algorithm based on multiple-level clustering, constructive placement, and simulated-annealing-based refinement works well in relation to an existing high-quality pure simulated annealing placement tool. It provides superior area results

across a set of large benchmark circuits (those containing between 3000 and 20,000 total logic and I/O blocks) compared to VPR when both tools are instructed to take approximately the same amount of time to generate a placement. For example, in 10 seconds on a 300 MHz Sun UltraSPARC, our ultra-fast tool can achieve an average area penalty of less than 30% (compared to high-quality placements that require on average over 250 seconds to produce), while the best that VPR can achieve is an 80% area penalty. A placement of a 100,000-gate circuit is produced by our tool in 10 seconds that is only 31% worse than a high-quality placement from VPR that requires 524 seconds; our ultra-fast tool achieves this level of placement quality 5 times faster than VPR. Furthermore, it takes VPR approximately 100 seconds to achieve an average area penalty of 10%, but the ultra-fast tool can attain the same level in less than 30 seconds. If we have no compile time restrictions, then our algorithm produces placements that approach the same quality as VPR; within 60 seconds on average, placements are produced with an average wirelength that is within 5% of that obtained using VPR.

The placement algorithm allows the user to smoothly trade quality of placement (bounding-box wirelength, a good estimator of wiring area) for compile time. We explored the space covered by these parameters to find the best quality-time envelope and showed that its envelope is significantly better than that possible with the pure simulated annealing formulation of VPR. We used this envelope to quickly and successfully predict the quality versus run time relationship of a specific circuit that has not yet been placed, given only the size of the circuit in logic blocks. This predicted envelope was returned to the user as feedback, and was used to automatically generate the required placement parameters to meet a user-specified compile time or area restriction.

We also showed that the ultra-fast placement tool can be used as a fast estimator of the final high-quality wirelength that is achieved when the pure simulated annealing placement tool is tuned to achieve minimum wirelength with no restriction on compile time. This fast wirelength prediction scheme is successful with a mean absolute error of 6% between the estimated and actual high-quality wirelength over the set of large benchmark circuits, in an average run time of merely 5.5 seconds on a 300 MHz Sun UltraSPARC.

5.2 Future Work

This thesis has provided the first exploration of very fast, flat placement, be it for FPGAs or as part of automated layout synthesis packages for gate arrays or standard cells. However, there are many areas within this topic to explore more thoroughly.

First, a fast placement tool that is timing-driven should be developed. Further improvements to the fast placement tool should include exploiting both the hierarchy that is inherent in the circuit structure, as well as the topology of the underlying routing architecture of the target FPGA.

In the future, it would be interesting and beneficial to explore a fast quadratic-programming-based placement algorithm or one based on top-down min-cut partitioning, and determine their quality-time trade-off relationships. Both of these techniques are based on popular placement algorithms that have been implemented in CAD tools, and it would be interesting to compare their performance and predictive ability with our tool.

Another interesting area to pursue is the refinement and integration of the fast high-quality wirelength estimator with the difficulty predictor provided by an existing fast router [Swar98a]. It would also be fruitful to extend the prediction scheme to incorporate fast estimation of circuit speed and power dissipation as well. Finally, a complete fast synthesis package should be on the road map of any fast compile project, integrating the existing work in fast placement and routing with fast technology mapping, fast partitioning, and perhaps even fast logic optimization and fast high-level synthesis, all of which should trade quality of result for compile time.

References

[Alpe97a] C. J. Alpert, T. F. Chan, D. J. -H. Huang, A. B. Kahng, I. L. Markov, P. Mulet, and K. Yan, "Faster Minimization of Linear Wirelength for Global Placement," *ACM Symposium on Physical Design*, 1997, pp. 4-11.

[Alpe97b] C. J. Alpert, T. Chan, D. J. -H. Huang, I. Markov, and K. Yan, "Quadratic Placement Revisited," *ACM/IEEE Design Automation Conference*, 1997, pp. 752-757.

[Alpe97c] C. J. Alpert, J. -H. Huang, and A. Kahng, "Multilevel Circuit Partitioning," *Proc. ACM/IEEE Design Automation Conference*, 1997, pp. 530-533.

[Babb97] J. Babb, M. Frank, E. Waingold, R. Barua, M. Taylor, J. Kim, S. Devabhaktuni, P. Finch, and A. Agarwal, "The RAW Benchmark Suite: Computation Structures for General Purpose Computing," *Proc. IEEE Symposium on FPGAs for Custom Computing Machines*, 1997, pp. 161-171.

[Betz97] V. Betz and J. Rose, "VPR: A New Packing, Placement and Routing Tool for FPGA Research," *Proc. Intl. Workshop on Field Programmable Logic and Applications*, 1997, pp. 213-222.

[Betz97b] V. Betz, "The FPGA Place-and-Route Challenge," 1997. (*Available from http://www.eecg.toronto.edu/~vaughn/challenge/challenge.html*).

[Betz98] V. Betz, "Architecture and CAD for Speed and Area Optimization of FPGAs," *Ph.D. Thesis*, University of Toronto, Department of Electrical and Computer Engineering, 1998.

[Brow92] S. D. Brown, R. J. Francis, J. Rose, and Z. G. Vranesic, *Field-Programmable Gate Arrays*, Norwell, MA: Kluwer Academic Publishers, 1992.

[Call98] T. J. Callahan, P. Chong, A. DeHon, and J. Wawrzynek, "Fast Module Mapping and Placement for Datapaths in FPGAs," *Proc. 6th ACM/SIGDA Intl. Symposium on FPGAs*, 1998, pp. 123-132.

[Chen94] C. Cheng, "RISA: Accurate and Efficient Placement Routability Modeling," *Proc. Intl. Conference on Computer-Aided Design*, 1994, pp. 690-695.

[Cong94] J. Cong and Y. Ding, "Flowmap: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table Based FPGA Designs," *IEEE Transactions on Computer-Aided Design*, Jan. 1994, pp. 1-12.

[Corm90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, Cambridge, MA: MIT Press, 1990.

[Doll91] K. Doll, F. Johannes, and G. Sigl, "DOMINO: Deterministic Placement Improvement with Hill-Climbing Capabilities," *Proc. VLSI*, 1991, pp. 91-100.

[Dunl85] A. E. Dunlop and B. W. Kernighan, "A Procedure for Placement of Standard-Cell VLSI Circuits," *IEEE Transactions on Computer-Aided Design*, vol. 4, no. 1, Jan. 1985, pp. 92-98.

[Gehr98] S. W. Gehring and S. H. -M. Ludwig, "Fast Integrated Tools for Circuit Design with FPGAs," *Proc. 6th ACM/SIGDA Intl. Symposium on FPGAs*, 1998, pp. 133-139.

[Hage97] L. W. Hagen and A. B. Kahng, "Combining Problem Reduction and Adaptive Multistart: A New Technique for Superior Iterative Partitioning," *IEEE Transactions on Computer-Aided Design*, vol. 16, no. 7, July 1997, pp. 709-717.

[Hame98] I. Hamer, "Implementation of DES on Transmogrifier-2a," *Personal Communication*, 1998.

[Hana72] M. Hanan and J. M. Kurtzberg, "Placement Techniques," in *Design Automation of Digital Systems, Volume 1: Theory and Techniques*, M. A. Breuer, Ed., Englewood Cliffs, NJ: Prentice-Hall, 1972, pp. 213-281.

[Huan97] D. Huang and A. Kahng, "Partitioning-Based Standard-Cell Global Placement with an Exact Objective," *ACM Symposium on Physical Design*, 1997, pp. 18-25.

[Huan86] M. Huang, F. Romeo, and A. Sangiovanni-Vincentelli, "An Efficient General Cooling Schedule for Simulated Annealing," *Proc. Intl. Conference on Computer-Aided Design*, 1986, pp. 381-384.

[Hutt97] M. Hutton, J. Rose, and D. Corneil, "Generation of Synthetic Sequential Benchmark Circuits," *Proc. 5th ACM/SIGDA Intl. Symposium on FPGAs*, 1997, pp. 149-155.

[Kary97] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel Hypergraph Partitioning: Application in VLSI Domain," *Proc. ACM/IEEE Design Automation Conference*, 1997, pp. 526-529.

[Kirk83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by Simulated Annealing," *Science*, vol. 220, no. 4598, May 13, 1983, pp. 671-680.

[Klei91] J. M. Kleinhans, G. Sigl, F. M. Johannes, and K. J. Antreich, "GORDIAN: VLSI Placement by Quadratic Programming and Slicing Optimization," *IEEE Transactions on Computer-Aided Design*, vol. 10, no. 3, Mar. 1991, pp. 356-365.

[Lam88] J. Lam and J. Delosme, "Performance of a New Annealing Schedule," *Proc. ACM/IEEE Design Automation Conference*, 1988, pp. 306-311.

[Leng90] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*, Chichester: John Wiley & Sons, 1990.

[Leve98] P. Leventis, "Using edif2blif Version 1.0," University of Toronto, Department of Electrical and Computer Engineering, 1998. (*Available for download from http://www.eecg.toronto.edu/~leventi/edif2blif/edif2blif.html*).

[Lewi97] D. M. Lewis, D. R. Galloway, M. van Ierssel, J. Rose, and P. Chow, "The Transmogrifier-2: A 1 Million Gate Rapid Prototyping System," *Proc. 5th ACM/SIGDA Intl. Symposium on FPGAs*, 1997, pp. 53-61.

[Quic98] Quickturn Design Systems Inc., *The Mercury Design Verification System*, 1998. (*Available from http://www.quickturn.com*).

[Rose90] J. Rose, W. Klebsch, and J. Wolf, “Temperature Measurement and Equilibrium Dynamics of Simulated Annealing Placements,” *IEEE Transactions on Computer-Aided Design*, vol. 9, no. 3, Mar. 1990, pp. 253-259.

[Rose97] J. Rose and D. Hill, “Architecture and Physical Design Challenges for One-Million Gate FPGAs and Beyond,” *Proc. 5th ACM/SIGDA Intl. Symposium on FPGAs*, 1997, pp. 129-132.

[Roy93] K. Roy and C. Sechen, “A Timing Driven N-Way Chip and Multi-Chip Partitioner,” *Proc. Intl. Conference on Computer-Aided Design*, 1993, pp. 240-247.

[Sank99] Y. Sankar and J. Rose, “Trading Quality for Compile Time: Ultra-Fast Placement for FPGAs,” *to appear in Proc. 7th ACM/SIGDA Intl. Symposium on FPGAs*, 1999.

[Sarr97] M. Sarrafzadeh and M. Wang, “NRG: Global and Detailed Placement,” *Proc. Intl. Conference on Computer-Aided Design*, 1997, pp. 532-537.

[Sato97] S. Sato, “Simulated Quenching: A New Placement Method for Module Generation,” *Proc. Intl. Conference on Computer-Aided Design*, 1997, pp. 538-541.

[Sech85] C. Sechen and A. Sangiovanni-Vincentelli, “The TimberWolf Placement and Routing Package,” *IEEE Journal of Solid-State Circuits*, vol. 20, no. 2, Apr. 1985, pp. 510-522.

[Sech88] C. Sechen, *VLSI Placement and Global Routing Using Simulated Annealing*, Norwell, MA: Kluwer Academic Publishers, 1988.

[Sent92] E. M. Sentovich et al., “SIS: A System for Sequential Circuit Analysis,” *Tech. Report No. UCB/ERL M92/41*, University of California, Berkeley, 1992.

[Shah91] K. Shahookar and P. Mazumder, “VLSI Cell Placement Techniques,” *ACM Computing Surveys*, vol. 23, no. 2, June 1991, pp. 143-220.

[Shin93] H. Shin and C. Kim, “A Simple Yet Effective Technique for Partitioning,” *IEEE Transactions on VLSI Systems*, vol. 1, no. 3, Sept. 1993, pp. 380-386.

[Sigl91] G. Sigl, K. Doll, and F. M. Johannes, “Analytical Placement: A Linear or a Quadratic Objective Function?,” *Proc. ACM/IEEE Design Automation Conference*, 1991, pp. 427-432.

[Sun95] W. Sun and C. Sechen, “Efficient and Effective Placement for Very Large Circuits,” *IEEE Transactions on Computer-Aided Design*, vol. 14, no. 3, Mar. 1995, pp. 349-359.

[Swar98a] J. S. Swartz, V. Betz, and J. Rose, “A Fast Routability-Driven Router for FPGAs,” *Proc. 6th ACM/SIGDA Intl. Symposium on FPGAs*, 1998, pp. 140-149.

[Swar98b] J. S. Swartz, “A High-Speed Timing-Aware Router for FPGAs,” *M.A.Sc. Thesis*, University of Toronto, Department of Electrical and Computer Engineering, 1998.

[Swar90] W. Swartz and C. Sechen, “New Algorithms for Placement and Routing of Macro Cells,” *Proc. Intl. Conference on Computer-Aided Design*, 1990, pp. 336-339.

[Tess98] R. Tessier, “Fast Place and Route Approaches for FPGAs,” *Ph.D. Thesis*, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 1998.

[Wirt96] N. Wirth, “The Language Lola and Programmable Devices in Teaching Digital Circuit Design,” *Proc. 2nd Intl. Andrei Ershov Memorial Conference*, 1996.

[Xili98] Xilinx Corporation, *The Xilinx Foundation Series 1.4*, 1998. (*Available from http://www.xilinx.com*).

[Yang91] S. Yang, “Logic Synthesis and Optimization Benchmarks, Version 3.0,” *Tech. Report*, Microelectronics Centre of North Carolina, 1991.

[Ye99] A. Ye, “Procedural Texture Mapping on FPGAs,” *M.A.Sc. Thesis*, University of Toronto, Department of Electrical and Computer Engineering, 1999.