Modeling of a hardware VLSI placement system: Accelerating the Simulated Annealing algorithm

William Merle Batts Jr.

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation
Modeling of a Hardware VLSI Placement System: 
Accelerating the Simulated Annealing Algorithm

by

William Merle Batts Jr.

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of 
Master of Science in Computer Engineering

Supervised by

Visiting Assistant Professor of Computer Engineering Dr. Marcin Łukowiak
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
July 2005

Approved By:

Marcin Łukowiak
Dr. Marcin Łukowiak
Visiting Assistant Professor of Computer Engineering
Primary Adviser

Stanisław Radziszowski
Dr. Stanisław Radziszowski
Professor of Computer Science

Greg P. Semeraro
Dr. Greg Semeraro
Assistant Professor of Computer Engineering
Thesis Release Permission Form

Rochester Institute of Technology
Kate Gleason College of Engineering

Title: Modeling of a Hardware VLSI Placement System: Accelerating the Simulated Annealing Algorithm

I, William Merle Batts Jr., hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or part.

William Merle Batts Jr.

Date: 7-29-85
Dedication

For all of my friends and family, especially Denise, whose unwavering support made this possible.
I'd like to thank my adviser Dr. Marcin Łukowiak for his guidance, wisdom and unending patience. I'd also like to thank my committee members Dr. Stanisław Radziszowski and Dr. Greg Semeraro, whose input and time given to this work is greatly appreciated.
Abstract

An essential step in the automation of electronic design is the placement of the physical components on the target semiconductor die. The placement step presents the opportunity to reduce costs in terms of wire length and performance degradation; however it is compute intensive and is NP-complete in terms of obtaining an optimal solution. As designs have grown in complexity and gate count, obtaining an optimal solution is not feasible due to time to market constraints or sheer compute effort required. Heuristic algorithms allow for efficient but sub-optimal designs to be produced with a reduction in processing time. A widely used algorithm is Simulated Annealing (SA).

The goal of this work was to develop a model that would enable an analysis into the feasibility of developing a hardware accelerated placement system which uses SA at its core. The SA heuristic was analyzed for possible improvements in efficiency with focus given to targeting the system for hardware. A solution implementing parallel computing with specialized hardware configurations inside a field programmable gate array (FPGA) was investigated as having the possibility to improve the efficiency of the SA-based algorithm. All supporting subsystems were also described for a hardware accelerated model.

A large speedup was analytically shown from both accelerating the critical path of the SA algorithm as well as novel methods of improving SA's efficiency. As data throughput requirements were not included in this work, the results presented may be optimistic for an overall system speedup. However, the results clearly show that future work is warranted in studying the concept of a hardware accelerated placement system.
## Contents

Dedication ................................................................. iii
Acknowledgments ......................................................... iv
Abstract ................................................................. v
Glossary ................................................................. xii

1 Introduction ............................................................. 1

2 Motivation ............................................................... 3

3 Background ............................................................. 5
  3.1 IC Design Flow ....................................................... 5
     3.1.1 Concept - Research & Development ....................... 6
     3.1.2 Concept - High Level Design ............................... 6
     3.1.3 Design Entry ................................................. 7
     3.1.4 Synthesis ...................................................... 7
     3.1.5 Placement & Routing ....................................... 8
     3.1.6 Physical Verification & Simulation ....................... 9
     3.1.7 Fabrication ................................................... 10
  3.2 Electronic Design Automation ................................... 10
     3.2.1 Logical Description ....................................... 11
     3.2.2 Structural Description ................................... 15
  3.3 Library Exchange Format/Design Exchange Format .......... 16
     3.3.1 LEF Syntax ................................................ 16
     3.3.2 DEF Syntax ............................................... 18
  3.4 Physical Design Automation ..................................... 20
     3.4.1 Placement ................................................... 21
     3.4.2 Routing ..................................................... 23
3.4.3 Back Annotation .................................................. 24

4 Placement ............................................................... 25
4.1 Exhaustive Search ................................................. 27
4.2 Generalized Hill Climbing/Local Search ................. 28
4.3 Min-Cut ............................................................. 29
4.4 Genetic Algorithms ............................................... 30
4.5 Tabu Search ........................................................ 30

5 Simulated Annealing ................................................. 32
5.1 Physical Model ...................................................... 32
5.2 Application to Combinatorial Optimization ............. 33
5.3 SA & Placement ..................................................... 33
5.4 SA Research ........................................................ 37
  5.4.1 Parallel SA ....................................................... 38
  5.4.2 Hardware Assisted SA ....................................... 39
  5.4.3 Greedy Mixed Perturbations .............................. 40
  5.4.4 Multiple Heuristic Combination ......................... 41

6 Software Implementation ........................................... 43
6.1 Software Revisions and Lessons Learned ............... 43
  6.1.1 Experimental Code ........................................... 43
  6.1.2 Initial LEF/DEF Placements .............................. 45
  6.1.3 Benchmarks ..................................................... 46
6.2 Logical Modules .................................................... 46
  6.2.1 LEF/DEF Reader & Writer ................................. 47
  6.2.2 Design Perturbation ......................................... 48
  6.2.3 Wirelength Estimator ...................................... 49
  6.2.4 Overlap Calculation ....................................... 50
  6.2.5 Move Acceptance .......................................... 51
  6.2.6 Design Update .............................................. 51
  6.2.7 Temperature Update .................................... 52
6.3 Temperature Schedule ........................................... 52
6.4 Benchmark Circuits ............................................... 54
6.5 Characterization & Optimization .......................... 55
  6.5.1 Benchmark Visualizations ................................. 59
# Hardware Model

7.1 Wire Length Estimator ............................................. 61
7.2 Overlap Detector .................................................... 61
7.3 Move Acceptance Logic ............................................ 63
7.4 Interfaces ............................................................ 65
    7.4.1 Datapath Control ............................................ 65
    7.4.2 Component Supply ........................................... 66
    7.4.3 Design Update & Algorithm Management ............... 67

# Method of Investigation ........................................ 70

# Results ................................................................ 72

9.1 Software Implementation ....................................... 72
9.2 Hardware Model .................................................... 72
9.3 Speedup Justification ............................................ 74

# Conclusions .......................................................... 77

10.1 Discussion .......................................................... 77
10.2 Future Work ........................................................ 78
    10.2.1 Software Implementation ................................. 79
    10.2.2 Control Unit .................................................. 79
    10.2.3 Data Management .......................................... 79
10.3 In Closing .......................................................... 80

# Bibliography .......................................................... 81
# List of Figures

3.1 Simple IC Design Flow ........................................... 5
3.2 One-bit Adder Example VHDL Description .......................... 12
3.3 One-bit Adder Logical Schematic ................................ 12
3.4 LEF Syntax for SIZE-only MACRO Statement ....................... 17
3.5 Example LEF Definition of a One-Bit Adder’s Components ....... 17
3.6 Bounding Box Wirelength Estimation and Overlap Penalty ....... 18
3.7 DEF Syntax for Simple Net listing ................................ 19
3.8 Example DEF Definition of a One-Bit Full Adder ................. 20
3.9 One-bit Adder Initial Random Placement .......................... 21
3.10 One-bit Adder Final Placement .................................. 22
3.11 Example DEF Definition of a One-Bit Full Adder ............... 23

4.1 Series of Neighboring Solutions Containing a Local Minimum .... 29

5.1 Molecules’ Movements per Temperature Region .................. 32
5.2 Simulated Annealing Placement Flow Chart ....................... 34
5.3 Bounding Box Wirelength Estimation and Overlap Penalty ....... 35
5.4 Characteristics Justifying FPGA Application ..................... 40

6.1 Initial Software Algorithm Organization .......................... 44
6.2 Software Module Interaction ..................................... 47
6.3 Sample Temperature Schedule .................................... 53
6.4 Software SA Placement Time Profile .............................. 56
6.5 Software SA Placement Cost vs. Iteration ......................... 58
6.6 Initial and Final Placements of Benchmark ibm05 ................ 60

7.1 Top Level RTL Schematic ........................................ 62
7.2 Wirelength Estimator RTL Schematic ............................. 64
7.3 Overlap Detector RTL Schematic ................................ 65
7.4 Move Acceptance Logic RTL Schematic ........................... 66
7.5 Overall System Model ............................................ 67
8.1 Test Method
List of Tables

6.1 Benchmark ibm01 – ibm04 Information ........................................ 55
6.2 Benchmark ibm01 – ibm04 Time Profiles (% total) ......................... 55
6.3 Cost Delta Function Time Analysis ............................................... 57
6.4 Average Cost Delta Function Composite Analysis ........................ 57

7.1 Top Level Hardware Input Interface ............................................ 63
7.2 Top Level Hardware Output Interface ......................................... 64

9.1 Time Comparison to a Mature Placement Tool .............................. 73
9.2 Die Grid Points per Benchmark .................................................. 74
9.3 FPGA Clock Cycle Requirement per Calculation ......................... 75
9.4 Speedup Analysis ................................................................. 75
Glossary

A

ASIC Application Specific Integrated Circuit – An IC which has been constructed to perform a limited range of functions very efficiently, usually faster than a GPP. Its functionality cannot be changed after it has been manufactured.

C

CAD Computer Aided Design – Design work managed and enhanced by the use of computer technology.

component A unit which provides some type of functionality to a hardware design and may be combined with other units to implement more complicated behavior. An adder is a common example of a component, providing the ability to add two values with the ability to be combined into a larger design, such as a multiplier.

D

DEF Design Exchange Format – An ASCII text based format which defines a design’s specific organization in terms of instances of components and interconnections. Used in conjunction with LEF. A community project organized by the Silicon Integration Initiative (SI2).

die The semiconductor target on which integrated circuits are constructed.
<table>
<thead>
<tr>
<th>E</th>
<th>EDA</th>
<th>Electronic Design Automation – A CAD technique allowing the design flow of electronic devices to become more manageable, reducing associated overhead.</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>FPGA</td>
<td>Field Programmable Gate Array – A digital processing device which has the ability to be programmed after it is manufactured allowing its functionality to be changed.</td>
</tr>
<tr>
<td></td>
<td>FSM</td>
<td>Finite State Machine – Logic which consists of a finite number of states with transitions and outputs defined by the current state and possibly the value of inputs.</td>
</tr>
<tr>
<td>G</td>
<td>GPP</td>
<td>General Purpose Processor – A processor which implements functionality through groups of instructions and generic functional units rather than specialized data structures.</td>
</tr>
<tr>
<td>H</td>
<td>HPWL</td>
<td>Half Perimeter Wire Length – A method of approximating a net’s interconnection wirelength by fully enclosing it in a minimal bounding box and taking half of the perimeter.</td>
</tr>
<tr>
<td>I</td>
<td>IC</td>
<td>Integrated Circuit – A general term for an electronic device which contains active semiconductor switches and possibly passive devices such as resistors or capacitors.</td>
</tr>
</tbody>
</table>
IP  Intellectual Property – Original work that is at its core, intangible; an algorithm is an example of this.

L

layer  The fundamental construct of a semiconductor device. The combination of different types of layers are used to implement functionality. Normally metal layers are used for interconnection while N+, P+ and polysilicon are used to create transistors.

LEF  Library Exchange Format – An ASCII text based format which defines the specific IC technology and components used to implement a design. A community project organized by the Silicon Integration Initiative (SI2).

LUT  Look-up Table – A construct used to implement logic functionality. Instead of implementing direct logic, these are programmable memories.

N

net  A connection between the ports of two or more components or I/O pads, can be envisioned as wires connecting the ports of components.

netlist  The collection of nets and components which describe a specific design.

P

PDA  Physical Design Automation – A CAD technique allowing the design flow of electronic devices to become less complicated in terms of satisfying design rule checks, electrical properties, physical organization, etc.

port  The point on a component to which a net connects.
**R**

**routing**  The act of physically defining all nets’ connections on the target semiconductor die. Usually performed with metal layers.

**S**

**SA**  Simulated Annealing – A stochastic heuristic which uses a cooling material as a model in order to solve combinatorial problems.

**V**

**via**  The constructs within a semiconductor which serve to interconnect layers.
Chapter 1

Introduction

As time to market pressures and integrated circuit (IC) design complexity increase, reduction of time for any step in the design flow may provide an advantage, technically and economically. An integral step in the IC design flow is placement in which components targeted to define a device’s functionality are logically placed on the semiconductor die. Efficient placements are desirable in that operation is improved by reducing delays, parasitic losses and if considered during placement, other cost factors [30]. The act of finding a placement in terms of a global optimum is an NP-complete combinatorial optimization problem [36] giving that it is not feasible to approach the placement of a large design with a brute force method [21]. In order to reduce computational requirements when performing placements heuristics are often employed [2] [33] [40]; these algorithms do not produce optimal results but do produce acceptable solutions (satisfying design constraints) while reducing computing time. For example, there are over 200,000 placement solutions for a nine component design on a three by three grid, all which must be evaluated in order to find an optimal solution. Using a heuristic method one can reduce the number of evaluations to hundreds – a clear savings of computing effort. One such heuristic method used in placement is Simulated Annealing (SA) which, as its name implies, is modeled after the cooling of metal and the behavior its molecules exhibit [1]. The focus of this work is to analyze, optimize and accelerate the Simulated Annealing heuristic with respect to its use implementing a placement algorithm.

This document is organized as follows; Chapter 2 provides a motivation for this work
by defining its place in current IC design flows and exploring prior methods of accelerating placement heuristics. Chapter 3 provides the background of electronic design automation and IC design flow. Chapter 4 describes in depth and compares algorithms which can be used for placement. Chapter 5 provides an analytical description of the Simulated Annealing algorithm and previous research looking to improve its speed as a placement heuristic. Chapter 6 describes the software implementation of a placement tool used to characterize the timing and analyze the critical path of the Simulated Annealing algorithm. Chapter 7 describes the hardware model generated using observations from the software implementation. Chapter 8 gives the analytical process used to validate the hardware model’s increase in performance. Chapter 9 details the results of the investigation while Chapter 10 states conclusions drawn from the investigation results and provides future direction for this work.
Chapter 2

Motivation

It would be difficult to argue that any technique or method allowing for a decrease in the development time of a product would not be desirable. This is especially true for the electronics industry which has no foreseeable slowdown in innovation and development [35]. Many goods currently ship with IC devices providing specialized capabilities which are subject to a process that may be the critical path in the product development cycle. To allow further growth in the complexity of these devices without imposing restrictive development time overhead, accelerated methods at the core of the design flow should be sought [41].

As the gate count increases (in the above mentioned devices) so does the associated development time in terms of computational requirements. Computer aided design (CAD) of electronic devices and targeting them to physical fabrication (commonly known as Electronic Design Automation (EDA) and Physical Design Automation (PDA), respectively) has reduced this time to a great extent but improvement is required as designs grow. Current EDA and PDA tools automatically satisfy the various requirements particular to fabrication processes while optimizing the design where possible (Mentor Graphics Design Architect, Mach TA and Calibre [24]; Synopsis Galaxy, DesignWare and Discovery [39]; Cadence Encounter [10]). The placement process is compute intensive [30] [36] and represents a significant amount of time in the design flow. As with any of the steps in the design flow, it may be revisited several times to further optimize the design or to fix problems encountered by following steps. Placement acceleration would be desirable to any group implementing
this step within a development path. Simulated Annealing [21] is a very common algorithm used to implement a placement tool [2] [32] [33] [40], accelerating this algorithm therefore has been the center of many studies [11] [14] [18] [23].

Prior work in this field has firstly focused on improving the Simulated Annealing algorithm through analyzing and modifying the perturbation types and cost functions [14] [18] [19] [28]. Other work has looked to parallel implementation of the Simulated Annealing algorithm purely in software to produce a speedup [1] [11]. The serial nature of the algorithm does not directly lend itself to this approach though parallel implementation has been shown to be successful [8] [23]. Other approaches of speeding up the Simulated Annealing algorithm have focused on hybrid implementations using other search methods as an augmentation [18] [20]. Hardware implementations have looked to specialized data and processing structures designed to be implemented in large matrices of execution elements on FPGAs [16] [41].

Analyzing the above prior work, it seems a hardware accelerated parallel processing approach would have the ability to provide a substantial speedup. Taking direction from [8] [11] [23] and [41], this work seeks to first characterize the pure software placement tool using SA as its core heuristic to identify the critical path in the data flow. Having this information, a tailored datapath can be developed which would provide some amount of speedup over the software tool. Considering hardware interface requirements (memory access, data structures, etc.) it also seems feasible that the parallel operation of multiple identical datapaths would produce near-linear speedups to a point. Applying knowledge of the behavior of the SA algorithm when applied to placement also provides novel approaches to a hardware accelerated approach.
Chapter 3

Background

3.1 IC Design Flow

Implementing an electronic design in an integrated circuit is by no means a trivial task, a number of steps occur between the conception of the idea to the delivery of the packaged IC. Each of the steps in the design process are highly correlated with others [30]; the process is usually not executed linearly. Figure 3.1 only gives an overview of the IC design flow steps. This figure is very simplified and only gives a general outline of the entire process as each step contains several underlying steps. As shown, the process may reverse direction to optimize the design or fix errors before resuming its original course.

![Figure 3.1: Simple IC Design Flow](image)

Figure 3.1: Simple IC Design Flow
Of the following steps, this thesis focuses on placement. Here the components of the physical design are arranged in an attempt to produce an optimized layout. All other steps are outside the scope of this work but are important in that together they define a semiconductor design flow.

### 3.1.1 Concept - Research & Development

The initial phase of design involves the analysis of the original idea, refinement and research. Some initial steps are to determine usefulness, profitability, feasibility, which target technology is used, and overall project goals/requirements. Any of the steps here will impact the rest of the process, an example of this would be the target technology. If high performance and volume is a project goal, application specific IC (ASIC) design may be targeted whereas if cost is a limiting factor or the design is to be produced in small volumes or for prototyping, less expensive standard cell or FPGA techniques may be utilized. Each decision here leads the project down different design flows, this must be carefully considered in this step.

### 3.1.2 Concept - High Level Design

Here the overall system architecture is defined and subsystems appear which may also be further broken down into smaller components. The high level design is taken from the results of the previous step and represented in modeling tools. At this point, either proprietary or target technology vendor supplied libraries may be utilized to reduce duplicate design effort as subsystems of the design may be readily available in these libraries as common components. Early insights to optimizations can be discovered in this step such as data bottlenecks, processing requirements and requirement variances.
3.1.3 Design Entry

This step begins the use of EDA CAD tools and generates the logical representation of the design. Two traditional types of design entries are typically used, schematic capture and text based modeling languages with the latter being more popular for large designs. Schematic capture involves using a GUI to represent components and connections. The designer creates a diagram to define a system in which the tool then creates an intermediate representation to be passed on to the next step. Modeling languages offer portability and self documentation wherein hardware designs are represented through the use of source code. Two very common languages are VHDL [13] and Verilog HDL [12], these are widely used to define hardware systems and to simulate a design before being mapped to a target technology. Others languages such as SystemC [26] look to fill in the design flow gaps that the two aforementioned languages leave open by not allowing overall systems to be modeled in great detail.

Many IDEs allow for both the concurrent use of schematic entry and a modeling language in order to leverage the strengths of both methods (the high-level design can be viewed in schematics and the low-level components can be viewed in HDL). Using such a tool allows for any changes in one method to update the other, maintaining coherency across all views.

A designer can find optimizations in this step through intelligent construction; a good designer will produce efficient, correct source code. Some tools used in this step are Mentor Graphics DesignArchitect [24], OrCAD PSpice [10] and Xilinx IDE (Integrated Design Environment) [42], Mentor Graphics ModelSim [24] and Aldec Active-HDL [5] for model language and/or schematic capture development and simulation.

3.1.4 Synthesis

The designs entered in the previous step are now translated into library or custom components. Here the designs are transformed into the description of physical entities having
size, shape, connection ports, and electrical characteristics. This is the first step in placing the design onto the target semiconductor die.

Each component has ports which serve to move signals to and from an exterior connection while groups of ports usually from different components may be connected together to form nets. It is the unique combination of components and connections which gives one design different characteristics from another.

The choice of technology used greatly influences this step. If an FPGA or standard cell library is used, the designer may use a tool to convert the design into its physical form in which a component list and a net list will be generated representing the project in circuit form. If full custom ASIC technology is chosen, another designer will have to create components from the output of the previous step either from scratch or from a generic form component. Some tools used here are Mentor Graphics’ Design Compiler [24] and Cadence’s BuildGates [10]. At this point the CAD tools in use move from EDA to PDA as the logical representation of the design is complete, further steps deal with applying the logical design to the physical process chosen in the concept stages.

3.1.5 Placement & Routing

The components from the previous step are physically applied to the floor of the target semiconductor die and the physical design begins to take shape. Here the placement of each component generated in the previous step with respect to every other component becomes important in order to minimize wiring delays and congestion as well as to minimize the target die size. As the focus of this work involves algorithms at the core of this step, more about placement will be expanded upon later. Routing (instantiating the connections defined by all nets on the semiconductor die) is of obvious concern as again wire lengths should be optimized to improve performance. Most designs are automatically placed and routed for any project of appreciable size. Direct manipulation by the designer is sometimes warranted but this is limited to small areas which require attention. Efficiently routing a large design by hand would be overwhelming for anything beyond a small project.
Placement and routing may be performed as two independent sub-steps or as a single integrated step. Performed as independent sub-steps, routing requirements must be considered in the placement step as to provide a routable design. The possibility of having to partially re-place the design exists as it is possible that some aspect of the initial placement will create problems in routing. The integrated place and route step may possibly suffer from unacceptably long run times as the search space created by combining both steps is much larger than either step by itself. With current semiconductor processes offering full routing over the components, where all interconnections exist above the transistor layers, independence of the two steps becomes more reasonable.

3.1.6 Physical Verification & Simulation

At this stage a physical representation of the project is complete and all timing and electrical characteristics of the system can be known allowing for an accurate simulation to take place. This is known as post-place and route simulation and makes use of parasitics extraction which uses the geometries of each transistor to fully specify a very detailed model. Previous simulations could not account for these values (due to being specific to physical construction) and were either ignored or estimated. Here the system can be measured to ensure the physical design will meet criteria set forth in the preceding stages.

Now having the physical characteristics of the target, post layout verification such as design rule checks and layout versus schematic can be performed to make sure the layout does not violate any fabrication rules and behaves as the designers intend, respectfully. Any mistakes here will likely send the project back to the place and route stage or worse, the synthesis stage if a major fault is discovered. It is possible that the fundamental design would require modification at which point there is no choice but to re-design around the problem and re-enter the correction in the logical representation. This obviously gives that care should be taken up to this point to ensure correctness. If the design passes all tests it will then be sent to fabrication in order to be produced, tested, packaged and delivered.
3.1.7 Fabrication

If the design is targeted to an ASIC, the physically defined project is created in a semiconductor foundry first as whole wafers then individual dies and finally packaged dies ready for use. In order to efficiently produce ICs a built-in self test can be included in the design to allow dies to be tested before packaging. In this way if a die fails its self test it can be discarded before packaging, saving time and money. If an FPGA is the target of the design, the FPGA is programmed using the bitstream generated by the design suite targeted toward the particular FPGA used. Typically, the manufacturer of the FPGA provides a software package to take a design from a concept to the finished, programmed FPGA without reliance on third-party tools, though third-party suites exist that replace this functionality [5]. The finished product is then marketed and sold or included in a larger project depending on its purpose.

3.2 Electronic Design Automation

Electronic Design Automation is a CAD technology aimed at managing the requirements of working with designs targeted to work inside of electronic technologies (custom IC, standard cell, FPGA). The term EDA is usually an umbrella term applied to all CAD technologies used to manage designs from ideas to silicon; EDA applies to all CAD tools used before application to the physical process and physical design automation (PDA), discussed below, involves all CAD tools used to manage a design after this point.

EDA is widely employed as a method to ensure that a group’s intellectual property (IP) is properly utilized by allowing modularization of designs and creation of proprietary libraries. Furthermore and more importantly, EDA tools allow for reuse of previously created IP, reducing duplicate effort. This organizational function of EDA tools is not its primary focus; EDA tools allow one to easily navigate overwhelmingly large designs with relative ease. Designs have grown in size both in subsystem hierarchies and pure transistor count to the point where one is no longer able to manage them purely by hand. This is not a new
revelation, in 1965 Intel co-founder Gordon Moore stated that IC transistor count will double every two years. In 1971 the Intel 4004 had a transistor count of 2,300, by 1982 with the Intel 286 the count had risen to over 130,000. Known as Moore’s Law, this prediction has held true and is foreseen to do so [35].

EDA’s primary purpose is to provide a method of translating a design input description to a logic description, i.e., combinations of basic logic functions such as AND, OR, XOR. As previously mentioned, the input type could be an integrated development environment (IDE) based schematic capture where a design is “drawn” in terms of visual elements or a text based language such as VHDL. The EDA package will take these inputs and create generic logical descriptions which may be optimized using libraries either previously created and archived by developers or provided by the target technology supplier.

The strength of EDA CAD technology is being able to simply represent, navigate and test large designs. EDA tools provide the intelligence to optimize designs with previously and specifically developed components. This saves duplicate development time allowing a group to build upon previous work and provides superior implementations of systems without requiring intimate knowledge of the target technology, optimal logical function implementations or excessive interaction with the designer.

3.2.1 Logical Description

The logical description represents the first step inside an EDA CAD tool toward the realization of a design as a fully functional electronic device. Taking as a very small example the generation of a one-bit full adder one can show the translation from input to logic (usually the full adder is an atomic element of an electronic design but for this example it can be decomposed). From fundamentals, a one-bit full adder is given as

\[ S_o = A \oplus B \oplus C_i \]  \hspace{1cm} (3.1)

\[ C_o = ((A * B) + ((A + B) * C_i)) \]  \hspace{1cm} (3.2)
where $A$, $B$ and $C_i$ represent the adder’s inputs and $S_o$ and $C_o$ represent the adder’s sum output and carry output, respectively.

Having its behavior defined, the design must be entered using one of the aforementioned methods; the core VHDL is presented in Figure 3.2 while a schematic capture is given by Figure 3.3. The syntax for the VHDL statements assigns the logic value of the

$$
S_o \leftarrow A \text{ xor } B \text{ xor } C_i;
C_o \leftarrow ((A \text{ and } B) \text{ or } (A \text{ or } B) \text{ and } C_i);
$$

Figure 3.2: One-bit Adder Example VHDL Description

function on the right hand side of the signal assignment operator (left arrow) to the output signal on the left. A VHDL compiler will then analyze the file containing these statements and assemble a logical description which can be visualized by Figure 3.3 which also would be the input of a schematic capture EDA tool.

The EDA tools now having the logical representation of the design can determine how

![One-bit Adder Logical Schematic](image)

Figure 3.3: One-bit Adder Logical Schematic

the system will act given certain inputs, in other words, the design can be tested and examined for correct behavior. Usually, this is the first time the design is tested with actual inputs and usually begins at the module level such that given known interfaces, individual designers can independently create modules that will produce a working system after integration. Normally, along with, and possibly before, generating a system’s module, a
testbench is generated that contains test inputs with known outputs such that fast go/no-go tests can be executed during development reducing the code/test/debug cycle's time. For something as simple as the one-bit adder, an exhaustive test set would be used to check all possible combinations of inputs whereas for development purposes in a much larger design a limited test set would be used for spot checks during development. A larger, more complete test would then be executed to ensure that the module will correctly function before integration into the system.

The basic building block used to implement a function within a design is known as a logic gate; within the example figure xor, and, and or are all instances of gates which implement basic logic functions. The elements in this example can also be called components, a structure which implements some amount of functionality. In this case the components implement basic functionality, these may implement more complex functions and it is up to the designer of the library to define their granularity. In synthesizing an arbitrary function it may be more efficient to implement a multiplexer-based look-up table (LUT) rather than pure logic as shown here. A synthesis tool will perform forward and reverse elimination in order to determine this; it's operating theory is outside the scope of this thesis. Here the tool traverses the boundary between logical and structural descriptions in mapping the logic to components. It is likely that unless an optimized library component exists for the example one-bit adder and depending on the level of optimization, it would be instantiated in a LUT.

To be defined as a component, interconnections must be able to be made to/from other components. To specify the points where these connections are made ports are used. A port is, as its name implies, a path which passes from outside the component to the functional elements inside. A port may have properties which specify the direction of the logical data flow in order to allow an EDA tool to determine a component is used correctly. The interconnections which are made between ports of components are known as nets. Typically, a net only connects to one port on a component and between a limited number of components, but may theoretically connect any number of nets of any number of components.
The electrical properties of the target technology limit the number of input ports an output port can drive. This value is known as fan-out. Conversely, there are real-world limitations on the number of outputs one input can support, this is not usually encountered as multiple drivers of an input are avoided (or at least advised against during EDA processing). This value is known as fan-in and is, along with fan-out, calculated for each net and verified not to exceed limits as defined by the target technology. In the example above, the nets can be identified as N1, N2, N3, N4, A, B, Ci, So and Co, the input nets A and B experience a fan-out of three each, the input net Ci experiences a fan-out of two and all internal nets experience a fanout of one. The output nets So and Co will experience fan-outs determined by the full adder's place in a larger design. As with function to component mapping, a synthesis tool will also consider these loading values when selecting a particular implementation. It is possible that a faster or smaller implementation may violate a loading constraint which would then require buffer(s) to remedy. This solution may increase a signal's latency (having to pass through the additional buffer(s)) thereby possibly decreasing the maximum operating speed. Selecting an implementation which is less efficient but acceptable in terms of satisfying loading constraints may be a better solution. Since no additional buffers are required, signal latency may be reduced and the maximum operating speed may be higher than the more efficient implementation. The synthesis tool takes this into account when determining which particular components are used in function mapping.

The collection of components and interconnections is known as a design's netlist, this is unique to each design and subsequently defines its structure and behavior. However, different netlists may define the same behavior through a different structure, this is the fundamental principle of optimization, discussed below.

Having translated the one-bit adder into a combinatorial logic function, the EDA tool can then examine the design in order to optimize it with pre-defined and very efficient library implementations. Although a one-bit full adder can be entered in this fashion, most standard libraries have provisions to use the ‘+’ operator to represent this function and
highly optimized implementations that run much faster than the straightforward definition.

### 3.2.2 Structural Description

Having the logical description which describes the design purely as a function of input to output behavior, an EDA tool can then perform synthesis to generate a list of components and connections which implements the desired logical behavior known as the structural description. From the one-bit adder example above, using fundamental gates the behavior would then be implemented using two each of AND, OR and exclusive-OR components. In reality, this would not be the case, to make full use of EDA's abilities, one would allow the CAD tool decide how to implement the full adder's logical function. The resulting implementation would then not be our explicit definition in VHDL but something defined in a target library given for the final technology implementation. Commonly, a technology vendor will supply basic libraries for their products along with premium libraries which may perform better than the basic libraries. The basic library would be free and a licensing fee would be paid for the premium offering; the developer would then have a jump start on development having components which are already optimized for the particular final technology.

A design can be implemented in many different ways which all produce the same logical behavior, however, one implementation may be superior over another due to requiring less components or running at a faster clock speed. As with many of the steps in the IC design flow, computational effort to implementation optimization trade-offs have to be made, as a function of processing time versus component count and operating frequency. A design will typically have a number of timing constraints as a function of the number of inputs and outputs. These constraints generally involve the processing and production of signals with respect to the system clock (which may require a minimum frequency as a constraint) or other signals. Knowing the intended functions and organization of the structural description, the static timing of all paths can be calculated and compared against the required constraints. Optimization then proceeds to satisfy any constraints which have been violated.
by the structural implementation. Once all constraints have been satisfied no further optimization is required the design flow can then proceed to applying the structural design to the physical implementation.

3.3 Library Exchange Format/Design Exchange Format

The Library Exchange and Design Exchange Formats (LEF/DEF) are ASCII text files which are capable of describing a library of components with the technology in which they are implemented and a specific design, respectively. OpenEDA, sponsored by the Silicon Integration Initiative (SI2), maintains the LEF/DEF formats as a community project meaning that anyone is able to request a license to the standards references and sources such as in this work [25]. SI2 is an organization of electronics, EDA and semiconductor technology vendors committed to reducing cost and increasing productivity within integrated silicon systems. The purpose of the LEF/DEF project is to create an open standard format which technologies, libraries and individual designs can be exchanged between organizations using tool sets from different vendors with no translation issues. Most EDA technologies use the GDSII [10] format to represent the physical design and proprietary file formats to represent library and netlist information. The GDSII format is used to transmit a design to a fabrication facility such that it can be constructed in the target physical technology; GDSII is being replaced by the OASIS [34] standard which offers higher density and 64-bit values. LEF/DEF seeks to give an open option to these binary, proprietary formats [25].

3.3.1 LEF Syntax

It is not within the scope of this work to discuss the entire LEF format though it is useful to describe the constructs used. The LEF format has the ability to describe technologies and components as they would be used to implement a design directly on the semiconductor die. This includes information describing all layers available within the semiconductor
technology, vias to interconnect layers and components which describe a library.

As stated above, the LEF file describes both the components and the technology in which a library is implemented, to this end, the LEF portion of a design may then be broken into two separate files. If this is the case, the technology portion of the LEF description must be read first in order to understand how the components are constructed and if they violate any design rules. It is an option to combine both technology and component descriptions, however, just as if two separate files are used, the technology section must be defined first. This allows for a reduction of redundant data as one file can be used to contain the technology description which multiple library files may reference.

Figure 3.5 gives the LEF description of the library used to implement the one-bit adder given in Figure 3.3. As shown, only one type of LEF statement is used to describe the components, the MACRO statement. This statement has many sub-statements defining all properties of the component such as port locations, construction, electrical behavior, etc. The sub-statement which is used here is SIZE which defines the minimum bounding box which completely covers all elements of the component. Figure 3.4 gives the syntax of a LEF MACRO statement containing only a SIZE sub-statement. Values of interest lie in

MACRO macroname ; SIZE width BY height ; END macroname ;

Figure 3.4: LEF Syntax for SIZE-only MACRO Statement

MACRO xor.2 ; SIZE 1000 BY 1000 ; END xor.2 ;
MACRO and.2 ; SIZE 1000 BY 1000 ; END and.2 ;
MACRO or.2 ; SIZE 1000 BY 1000 ; END or.2 ;

Figure 3.5: Example LEF Definition of a One-Bit Adder’s Components

the width and height values of the SIZE statement which defines the physical box which must be placed onto the semiconductor die. As not all components are perfect rectangles there are constructs to define which portions of the bounding box are not obstructed but for the purposes of this work, these values are ignored. It is a design rule violation to overlap any portions of two components, doing so will result in a design which will be unable to
correctly function. To prevent this, these bounding boxes are used as the boundaries which define if an overlap exists between two components as shown in Figure 3.6.

![Figure 3.6: Bounding Box Wirelength Estimation and Overlap Penalty](image)

### 3.3.2 DEF Syntax

As with the case of the LEF format, it is not necessary to discuss the DEF format in its entirety. The DEF format uses the information given in the LEF file(s) as its reference to define a specific design in terms of instances of library components and interconnections. The DEF format has the provisions to define all aspects related to the component level of a physical design such as instances of components, their position, orientation, the size of the target die, all connections between all instances in the design, etc., and relates closest to the focus of this thesis.

Specifically, this work is looking at individual instances, or components, and their relationship to all other components to which they are connected, known via the netlist. The sample syntax of the DEF format is given below in Figure 3.7, the format clearly shows instantiations of components and provisions to declare their positions on the die with respect to their “Southwest”, or bottom left corner. The DEF file is both the primary input and output of the placement step, before placement components may be roughly placed or
locked by the designer. With this, the DEF file serves as a guide file such that the placement algorithm is not forced to organize large subsections of a design such as memory blocks, arithmetic units, etc. Also clearly shown is the netlist given by each net's name, the component members and the respective ports connected to the net. Having the essentials to physically define a design, instances of components, their positions and interconnections, one can then proceed to translating a logical design given by EDA tools to a physical device using PDA tools. The example circuit, the one-bit adder, used throughout this document

```plaintext
DESIGN design_name;
TECHNOLOGY technology_name;
DIEAREA (die_SW.corner_coordinates_x.y) (die_NE.corner_coordinates_x.y);
COMPONENTS numcomps;
  - instance_name library.component + FIXED SW_x.y | PLACED SW_x.y | UNPLACED;
  ...
END COMPONENTS
NETS numnets;
  - netname (component1 port) (component2 port) ...;
  ...
END NETS
END DESIGN
```

Figure 3.7: DEF Syntax for Simple Net listing

is represented in a DEF file in Figure 3.8, below. One is able to see that the design uses two instances each of the three library components and that each instance has no initial placement position. Also given are the dimensions of the target die which has an area of 36 million square units; the area required by the design is only six million square units (one million square units per six components). The netlist is clearly shown with all nets having more than one connection (the output nets have only one connection and are not pertinent in this example) and their member components. Normally, for a full physical description, the input/output (I/O) pads of the target die would also be defined and placed as fixed components either by the designer or the PDA tool. The input and output nets would then include these I/O components which would then be considered part of the placement problem.
DESIGN onebitadder;
TECHNOLOGY tsmc035;

DIEAREA ( -3000 -3000 ) ( 3000 3000 );

COMPONENTS 6;
- xor1 xor_2 + UNPLACED;
- xor2 xor_2 + UNPLACED;
- and1 and_2 + UNPLACED;
- and2 and_2 + UNPLACED;
- or1 or_2 + UNPLACED;
- or2 or_2 + UNPLACED;
END COMPONENTS

NETS 7;
- A ( xor1 A ) ( and1 A ) ( or1 A );
- B ( xor1 B ) ( and1 B ) ( or1 B );
- Ci ( xor2 B ) ( and2 B );
- N1 ( xor1 Y ) ( xor2 A );
- N2 ( and1 Y ) ( or2 A );
- N3 ( or1 Y ) ( and2 A );
- N4 ( and2 Y ) ( or2 B );

END NETS
END DESIGN

Figure 3.8: Example DEF Definition of a One-Bit Full Adder

3.4 Physical Design Automation

Up to this point in the development flow the design itself has been treated as a logical entity, one performing some function with outputs affected by its state and inputs. This has been broken into subroutines and assigned to elements consisting of combinations of basic logic gates. These have been instantiated in components which by themselves do not comprise a complete design, however, as a hierarchy which has an organization a full description is achieved. Thus the physical instantiation of the design begins, having the library of components and the specifics of the semiconductor technology from the LEF file(s) and the logical function translated into individual instances of the library components along with
the full list of interconnections from the DEF file, the design can be treated as a tangible item. With this comes the physical manipulation of the components and interconnections in order to produce a usable design.

3.4.1 Placement

As placement will be covered in much more detail later, only the basic operation is described here. Figure 3.9 gives the initial placement of the example carried throughout this document, the one-bit adder. Clearly shown are the six components comprising the example design. In this example, in order to reduce the interconnection length of this design it is necessary to place the components as close together as possible. In general there are other considerations that may make ultra-compact placements undesirable (for example, power density or wiring congestion), here for simplicity and clarity, they are not considered. Figure 3.10 gives the design after placement has occurred, by comparing to the previous placement one can see that the components are placed near optimally with respect to interconnect wire length; the software Simulated Annealing algorithm (presented in Chapter 6) could not find the best positions for components xor2 and and2. Another notable feature is that there is no overlap between any of the components, a strong requirement for a desirable placement. If one were to ignore overlap during placement, the algorithm
would undoubtedly find the naïve solution in which all components are placed on top of each other. One reason that the optimal placement was not discovered is that the die area is much larger than the design requires, unnecessarily increasing the search space.

In this example with a die size of 8000 by 8000 units and each component 1000 by 1000 units in size, Equation 4.3 can be used to find the size of the search space (placement points taken to the power of the number of components). Given there are six components which must be placed, the total number of solutions is $1.38 \times 10^{46}$. Reducing the die width and height by half brings the search space down to $5.3 \times 10^{41}$ possible solutions while still allowing room for the entire design without overlap. Clearly, enormous search spaces exist for even the simplest of placement problems.

As the placement algorithm performs an action that is recorded, the DEF file is modified to reflect the new positions of the components on the die. The following component statements now replace the statements in the initial file given in Figure 3.8. Here, the coordinate values of the position of each component’s Southwest corner is given along with the

```plaintext
- xor1 xor.2 + PLACED ( 0 1000 ) ;
- xor2 xor.2 + PLACED ( -50 -50 ) ;
- and1 and.2 + PLACED ( 2000 1000 ) ;
- and2 and.2 + PLACED ( 1000 -50 ) ;
- or1 or.2 + PLACED ( 1000 1000 ) ;
- or2 or.2 + PLACED ( 2000 0 ) ;
```

Figure 3.11: Example DEF Definition of a One-Bit Full Adder
keyword PLACED which indicates it has been intentionally placed in that location but may be moved by hand or algorithm. After the placement step the design’s layout is then complete with respect to all components’ positions, the interconnections must now be routed.

3.4.2 Routing

Routing is the PDA step that generates the physical interconnections between components given by the netlist in the DEF file and the placement from the last step, respectively. Routing, like all EDA/PDA steps, has many challenges which must be overcome or mitigated in order to produce a properly functioning device. Some considerations which must be taken into account are wiring congestion, wire capacitance/inductance coupling, antenna effects, etc. It is likely that some of these considerations may be unresolvable or unacceptable in the routing step resulting in another round of placement in order to remove the problem. In this case, most of the design will remain fixed and only the problem area will be modified. The placement and routing steps will then iterate until a routable placement is generated.

After the routing step is complete the routing tool updates the DEF file with information indicating the physical layout of the routing. This includes specifying layers used, locations of vias and the shape of the wires implementing the netlist. With this, the physical design process has come to a point where the system is able to be fabricated in the semiconductor technology which for it was originally targeted. This is rarely the final step as further verification and testing is performed to ensure the steps of physically creating the design has not introduced errors with respect to the device’s electrical properties.

An interesting consequence of increasing transistor speeds is that wire (signal transmission) delays have begun to become larger than logic delays. This gives that the insertion of buffers may result in faster circuits [7] in certain situations. This is important to the EDA/PDA community in that tools will have to take this into consideration during synthesis, placement, and routing. If a long interconnection absolutely has to be constructed, its delay may possibly be mitigated through the insertion of a buffer. The design would then have to be analyzed with this buffer in place to determine if there is a net benefit to
its presence. If included, this additional component would then have to be inserted into the design, possibly after the first placement attempt. If further placement attempts eliminate the long interconnect, this buffer may then pose a performance reduction giving that it should be removed. Furthermore, placement of the buffer with respect to the location on the interconnect determines its effectiveness, a placement tool would then have to take this into consideration. As logic delays continue to become less than wire delays, the EDA/PDA community will have to integrate the management of performance increasing buffers accordingly.

3.4.3 Back Annotation

This step involves extracting all electrical characteristics of the final translated, placed and routed design in order to allow for very precise simulation. Previous simulations could only approximate the electrical properties of the entire design not knowing the physical geometries of the device. Now, having the device and it's physical layout, the simulator can account for previously unknown factors which may affect the performance or even the correctness of the design in order to ensure that the final product when packaged and inserted into a circuit will function as originally envisioned during the first steps of the IC design flow. If any problems are found here the design may be sent back to previous steps, discussed above, to enter modifications which will hopefully correct these problems without introducing others. The final design is translated into a format which a fabricator can understand and transmitted such that the device may be realized.
Chapter 4

Placement

Placement involves arranging components inside of the floorplan of the die or FPGA targeted for the construction of the integrated circuit implementing the original design. This can be thought of as analogous to arranging an office’s floorplan (target die) with different size offices (components) in order to set up the floor in the most efficient way possible. Efficiency has multiple variables each of which must be considered when determining an acceptable arrangement. Making sure that the employees of the office are situated close to others which whom they will have the most contact with is a definite goal, however, making sure the office holds as many employees as possible should also be considered. Allowing room for walkways and other essential constructs is a restriction on both the number of workspaces which can be included and the way they can be placed on the floor. These comparisons are directly applicable to placing an integrated circuit; both the proximity of interacting components and the density of their placement are of high importance. Additionally, allowing room for structures such as wires must also be considered while arranging the components, if placed too tightly these will have no room or the semiconductor may overheat thus crippling the circuit much in the same way an office with no walkways or which has employees packed against each other could not function.

Determining a perfect placement solution is an NP-complete problem [30] giving that for any design of considerable size finding a complete solution requires overwhelming computational effort. For a design with $n$ components, there exists a search space on the order
the number of placement grid points taken to the power \( n \) \([30]\). With time to market, development tool and other administrative concerns, simply allowing a machine to exhaustively operate on trial placements in order to find an optimal solution over a matter of months or investing in cutting edge hardware to reduce the compute time to days are not acceptable methods of CAD assistance. More elegant methods of placement providing very good but not perfect solutions allow for a reduction in computing effort; the cost of a non-optimal solution is deemed acceptable for its payoff in reduced placement time.

The act of placing components onto a die is an example of a combinatorial optimization problem \([36]\); the general formulation is introduced here. Placement is a minimization problem meaning that one placement which has lower cost than another is more desirable and a placement which has the least cost is considered as the optimum. Combinatorial optimization may also exist as maximization problems in which the highest cost is considered as the optimum. A specific problem instance, or a unique component set and netlist, can be formalized as a pair \((S,f)\) where \( S \) represents the finite set of all possible solutions, or placements, and \( f \) gives the cost function by which individual solutions in set \( S \) may be compared to one another. The cost function mapping is defined as

\[
 f : S \rightarrow \mathbb{R}
\]

which is to say the cost function produces a real value for individual solutions from a given set. As stated above, placement is a minimization problem; this gives that a solution is sought that satisfies

\[
 f(i_{\text{opt}}) \leq f(i), \forall i \in S
\]

A maximization problem uses the formula given above with the inequality reversed and both minimization and maximization problems use the term optimal to represent the best solution in set \( S \), \( i_{\text{opt}} \) or set of best solutions \( S_{\text{opt}} \). The globally optimum solution \( i_{\text{opt}} \) is either a minimal or maximal solution depending on the problem type, minimization or maximization, respectively \([1]\).

The cost function focuses on the parameter(s) which make one solution more desirable
than another by evaluating said parameter(s) in a way to meaningfully represent the solution with respect to others. Cost functions are usually unique to the problems in which they are implemented, generally making the comparison of two different optimization problems using the same cost function meaningless if not impossible. Costs can include any number of parameters which are pertinent to finding an optimal solution. As this determines a best solution, implementation of the cost function is a very important portion of the combinatorial optimization problem. Along with actually determining the optimal solution using a cost function, finding the function's representation which will give a specific solution from the search space as an optimal cost is also an NP-complete problem. However, analysis of the problem at hand will provide very good guidelines as how to define the cost function, i.e., minimizing wire length for placement.

4.1 Exhaustive Search

The most trivial method that can be used to find the best placement solution is an exhaustive search involving computing the cost of each individual placement inside of the search space and selecting the optimal solution(s), as stated above, an NP-complete operation. Using the naïve approach, the quantity of placement solutions is proportional to the number of placement points on the target die taken to the power of the number of components. Specifically, the number of solutions is given by

\[ \prod_{n=0}^{i} (X_{points_n} \times Y_{points_n}) \] (4.3)

where \( i \) is the number of components and \( X_{points} \) and \( Y_{points} \) are the number of points in which component \( n \) may be placed in the \( x \)-axis and \( y \)-axis, respectively. As this method is very inefficient for larger designs, it is rarely used. For trivial designs (tens of components), this method will provide the best output for computing effort invested, guaranteeing the optimum solution. In general, this method is not recommended.
4.2 Generalized Hill Climbing/Local Search

This method is a modification of the exhaustive search in which an intermediate solution is kept only if its respective cost is more desirable than the one from which it was generated. Though this method is a derivative of exhaustive search, it has striking resemblance to Simulated Annealing [1]. The concept of a neighborhood structure is introduced here to facilitate generation of new solutions to be compared against the original.

A neighborhood is the set of solutions created by moving away from the current solution by one "step". A step is defined in the same manner as the cost function, dependent on which parameters are pertinent to the problem and usually independent from other problems. For each solution \( i \in S \) a set \( S_i \subset S \) of solutions that are close to \( i \) by one step, \( S_i \) is known as the neighborhood of \( i \) and any solution \( j \in S_i \) is known as a neighboring solution to \( i \).

The algorithm usually begins by generating a random solution, computing its cost and generating a neighborhood from the initial solution. The neighborhood is then searched for a better solution as compared to that which the search neighborhood was generated from. If a better solution is found, a new neighborhood is generated from this and the process is repeated. The process continues to iterate until a neighborhood is generated which contains no neighboring solutions with a better cost than the prior solution.

Here, the algorithm only moves along the hill if the given path will take it to a more desirable solution – lower cost in this case. The major downfalls of this method are that it is highly dependent on the initial placement and that it is highly susceptible to being caught in local minima. Being purely greedy, the algorithm will consider a local minimum to be the best solution when more desirable solutions may exist which are only reached through first increasing the solution's cost. Figure 4.1 displays this graphically. A local minimum solution \( i \) is defined as

\[
f(i) \leq f(j), \forall j \in S_i
\]  

(4.4)
or $i$ is a solution which has a lower cost than any other solution in its neighborhood.

It is very possible that a poor intermediate solution will be selected such that better solutions will never be encountered, this can be avoided by starting the algorithm with a large number of initial solutions. As more initial solutions are used, the probability that a global optimum will be found asymptotically approaches unity [1]. Given this, generation and search of initial solutions is easily parallelizable if independent neighborhoods can be guaranteed, i.e., duplicate search efforts can be eliminated. Mixing this method with other methods, possibly as a final greedy step, may yield better results more efficiently than if used alone [36].

### 4.3 Min-Cut

The min-cut method is a recursive partitioning method which uses the principle that if a floorplan is cut in half, the fewer wires that it cuts, the more efficient the placement. The min-cut operation is performed on the sub-levels of the first cut and so on until only one component is left at the lowest level. There are several problems with this method in its purest form including loss of information from one level to the next; however, techniques
can be applied allowing the algorithm's efficiency to be increased. With respect to placement this method is best used to quickly converge on a solution used either as a final result or as a starting point for another algorithm [29] [36].

4.4 Genetic Algorithms

These algorithms take their form from nature and evolution, that is to say a population is formed, breeding occurs and the members who are most fit for their purpose survive to pass good traits of the species along to the future. In its application to computing several possible, usually random, solutions to the problem at hand are placed in the population and they are allowed to "breed" and "mutate" or otherwise create new members which have characteristics common to both progenitors and randomly introduced, respectively. The population is evaluated and the members least fit for their purpose or the most inefficient solutions are eliminated thus removing the undesirable traits from the solution pool. Ideally the good traits, or the best partial solutions, will continue into the future and combine together to produce a member with superior traits, or optimal solution. Many implementation details have not been mentioned here as this area of computational study is very rich and not within the scope of work, however, this algorithm type can be powerful when applied to certain optimization problems [15] [27]. Using this class of algorithms combined with Simulated Annealing and others has shown some amount of success [20].

4.5 Tabu Search

This search method is often used to augment other methods such as local search, described above [20]. The main concept here is to penalize moves that would take the algorithm to points in the search space which have already been visited. A history of moves is maintained in a finite, first in first out queue, if a restricted move is encountered it is rejected or at least not unconditionally accepted. The next move is then entered into the list and the
move which is oldest in the queue is removed allowing it to again be freely considered in the next solution.

This is a robust and simple amendment to many search methods and may be used in particular stages of the search to refine its behavior. As an example, this method can be applied to the local search to allow it to escape local minima by allowing the algorithm to try solutions that may be detrimental to the cost of the intermediate solution but allowable as long as the particular solution point has not been visited in the tabu history.
Chapter 5

Simulated Annealing

5.1 Physical Model

Annealing is a mechanical process in which material is slowly cooled allowing the molecules to arrange themselves in such a way that the material is less strained thereby making it more stable. If materials such as glass or metal are cooled too quickly its constituent molecules will be under high stress lending it to failure (breaking) if further thermal or physical shocks are encountered. Slowing the cooling of the material allows each molecule to move into a place it feels most comfortable, i.e., less stress. As the material is kept at a high temperature the molecules are able to move around quite freely thus reducing stress on a large scale, indeed if the material is made too hot it will move into the liquid state allowing free movement of the molecules. As the material is cooled the molecules are not able to move around as freely but still move limited distances reducing stress in regional areas. The re-

![Figure 5.1: Molecules’ Movements per Temperature Region](image-url)
suit is a material with significantly less internal stress and resistant to failure due to external shock. If one equates molecules to components and the substance to the overall design of an electronic circuit, Simulated Annealing can be applied to efficiently place the system onto the target die.

5.2 Application to Combinatorial Optimization

Simulated Annealing (SA) is a stochastic algorithm, or one that implements random elements at its core. As a Genetic Algorithm attempts to model evolution as a way to select an optimal solution, Simulated Annealing looks to the model of molecules in a heated mass and the way they behave as they cool to form a structured solid. The aim of the algorithm is to reduce the energy of the system through a slow cooling. As applied to placement, system energy is measured in the inefficiency (cost) of the placement; a poor placement will cause a system to have higher energy. This analogy is drawn from molecules in the cooling mass to components in the placement; a quickly cooled mass is quite fragile as a poorly placed design is inefficient due to the molecules and components, respectfully, being arranged in such a way that they experience internal tensions amongst each other as they try to move to regions which would lower their energy. After the material is cooled, the molecules (components) are frozen in these non-optimal positions, resulting in overall fragility of the system.

5.3 SA & Placement

VLSI placement in general consists of rectilinear components being targeted onto a rectangular or square die area in such a way that the interconnect wire length is minimized. Components are moved about the die in the x and y planes to generate a placement which has an interconnect wire length associated with it. In general, components are free to move to any location on the die and the interconnect wire length is calculated by measuring and
summing the length of wire used to connect each net, or connection of a group of ports [30]. The general case stated here is cumbersome at best to use inside of the iterating SA algorithm, therefore some refinements of the model are used in order to increase its efficiency.

The algorithm is generalized by Figure 5.2. The initial placement may simply be given as a random placement for each component able to be moved, i.e., not fixed, or a more
intelligent mechanism may be used to initially place the design in order to allow the core algorithm to work more efficiently. The cost function is usually comprised of several parameters each measuring a different aspect of the current solution. A very simple and widely used cost function parameter is the interconnect wire length of a placement solution [2] [32] [33], this can be easily approximated using the bounding box method [36]. This wire length estimation method draws a bounding box around all ports in a given net, half the perimeter of this box is taken as the net’s interconnect length approximation. The half-perimeter wire length (HPWL) estimation for minimally routed two and three port nets gives an exact value [33]. The sum of all HPWL, i.e., all nets, gives a value of the approximate interconnect wire length for a placement solution. Another cost function parameter widely used is component overlap; as design rules do not allow components to overlap each other, any instance of such should be considered as a penalty [33]. In the example below (Figure 5.3), the shaded components are members of a net; the minimum box that can enclose them has a perimeter of 56 units and half that length, 28 units which is taken as the HPWL approximation of the wire length required to interconnect this net. Also shown is a component that overlaps others, the total overlapping area is 11 sq-units which is multiplied by a penalty factor and added directly to the wiring length approximation.

Removing overlap during the iterative process is desirable since allowing an accu-

Figure 5.3: Bounding Box Wirelength Estimation and Overlap Penalty
mulation during the placement can yield invalid results and require more time in another round of placement. Any instance of overlap does not automatically exclude a solution as this may only be an intermediate step, the new placement solution may overlap two components although this could be in exchange for large savings in total interconnect wire length. It should be noted that allowing overlap during intermediate solutions must be balanced in such a way that the algorithm will produce usable results. In order to implement more specific placement models additional cost function parameters can be used. In row based standard cell placement it is desirable to keep row sizes from growing too large in order to yield a placement which is able to fit on the target die. A parameter measures the individual row sizes and increases the cost of the placement solution if one or more dominate the maximum lengths. Depending on the desired characteristics of a placement solution, the cost function can be tailored with many parameters to ideally produce the optimal solution.

Perturbing the placement to generate a new solution can be performed in many ways and different methods can be used concurrently in the algorithm. All methods generate a new solution in the neighborhood of the original and their respective costs can be calculated. This is a direct tie-back to the local search method described above. The simplest way to generate a new placement is to move one random component from one position to another random position while another fairly simple change is to swap two random components’ positions. Changing a component’s orientation will move its ports resulting in small changes to interconnect lengths of the nets of which the component is a member. This type of move is usually only performed when no other types of perturbations yield a new solution.

The key to applying Simulated Annealing to placement is the use of a cooling schedule which the algorithm follows. As the algorithm’s greediness is inversely proportional to the system’s temperature, moves may be accepted that actually allow an increase in a placement’s cost. Enough time must be spent in the upper and lower temperatures to allow first a quick arrangement of the system and a final localized arrangement, respectively. If too much time is spent in the upper temperatures, processing time will be wasted as many
inefficient intermediate solutions will be accepted, if too much time is spent in the lower temperatures, processing time will again be wasted due to a tight restriction on accepted moves.

Specifically, the cooling schedule can follow any function but it typically employs two slopes, one steep slope for the extreme high and low temperatures and a smaller slope for the intermediate temperatures where the most beneficial placement changes will be made. As stated above, changes resulting in a reduced cost will always be accepted but that is not to say that changes resulting in an elevated cost will always be rejected. Temperature has an effect on the probability of a cost inducing change being accepted with the specific form

\[ e^{-\left(\frac{\Delta C}{T}\right)} \]  (5.1)

where \( \Delta C \) is the positive cost change due to the new placement and \( T \) is the current system temperature given by the cooling schedule. This function is combined with a random value generator, if the randomly generated value is greater than the temperature function result the new placement is accepted. It is easy to see that for very large temperatures almost any change will be accepted while as the temperature is reduced the chance that a positive cost change will also be accepted is reduced. Each temperature step may contain several placement perturbations in the algorithm, adjusting this number is one of the refinements that may be made to deliver a more computationally efficient placement.

### 5.4 SA Research

Having the ability to perform multiple objective optimization [19], SA readily lends itself to cell placement which may have many variables determining the overall quality of the solution. With this, much research has been performed to otherwise increase SA’s efficiency; SA is very compute intensive and has the characteristic of wasting some of this effort on calculations that do not contribute to the overall solution due to rejected moves.

Inside of this work there are two main areas of focus, accelerating the critical path of the serial SA algorithm and parallelizing the serial SA algorithm in general, both inside of
an FPGA co-processor. These two areas are reviewed along with two other interesting concepts, namely randomly involving greedy moves and the combination of the SA algorithm with other optimization algorithms.

5.4.1 Parallel SA

Parallelization of the SA algorithm has been the main focus of academic research with respect to increasing performance and can be further refined into sub-categories [11]. The most common method of parallelizing the SA algorithm involves allowing each processing entity to make independent moves while manipulating common data. This has the obvious downfall in that error may be introduced if data common to multiple calculations is not coherent due to changes made by one of the processing elements. Two methods of dealing with this problem are ensuring that calculations will be performed on independent data and utilizing some error tolerance procedure [14], possibly to the effect of improving overall runtime at the cost of the placement’s quality [31]. Another method of acceleration is the parallelization of individual moves where one logical SA program runs with each of the individual SA steps performed on separate processors. This is not as common as the communication to computation ratio make this approach only feasible on shared-memory architectures [23]. An interesting approach is the use of Markov chains [11] which is the parallel, concurrent manipulation of duplicate and independent memory-spaces with the periodic exchange of desirable solution characteristics. This can be differentiated from the concept above by noting that Markov chains replicate the SA program and solution memory per processor where the concurrent, parallel technique above operates on the same memory-space. [8] introduces synchronization periods to the concurrent common-data parallelization model to limit the amount of error accumulation and to assist with convergence but is not an implementation of Markov chains; the synchronization is a method of beginning moves with all logical SA processes accessing data guaranteed to be valid.

This thesis focuses on implementing parallel moves while operating on a common memory space using a hardware accelerated, not parallelized, cost calculation and move
evaluation path. Parallelization of the SA steps was not considered due to focusing on optimizing the critical path of the serial SA algorithm and implementing a tightly coupled parallel moves concept but is inadvertently partially achieved through the FPGA implementation, specifically, the HPWL estimation and overlap calculations are performed in parallel. Though Markov chains show linear speedup characteristics, duplicating the solution space memory and processing paths across multiple instantiations may not be feasible on FPGA architectures due to gate count and memory limitations. In this work, only the SA move and evaluation path is duplicated using a central control unit to coordinate operations.

5.4.2 Hardware Assisted SA

The notion of using an FPGA's characteristics [16] to accelerate the SA algorithm for placement is not new, though work in [41] focused on using a systolic structure for FPGA and not cell placement as in this work. This structure assigns a separate SA structure for each look-up table (LUT) to be placed in the FPGA (an FPGA LUT is analogous to a design component). Three points are made justifying an FPGA's application to SA placement; the first being the reduction of communication cost by having processing entities near each other. The non-synchronizing parallel moves implementation as implemented in this work does not have to account for this concept as each parallel path only communicates with a central controller, though this communication is minimized due to existing on the same die. The second point is that in a systolic processing structure there is no cache thrashing as seen on SA implemented in general purpose processors by the fact any data that will be used in manipulations will exist in the structure at runtime while traditional SA randomly accesses the entire solution space stochastically. This is a problem in the design implemented in this work though it is mitigated by keeping the entire solution memory near to the FPGA and not allowing the central controller to become fully utilized fetching memory when other functional operations can be performed. Furthermore, it can be enforced that each parallel processing entity independently requests memory access through staggered scheduling. The third point simply states that the calculations and decisions associated with SA can be
optimized inside of the FPGA thus reducing the overhead typically required in the general case. Figure 5.4 illustrates the above three points.

Figure 5.4: Characteristics Justifying FPGA Application

5.4.3 Greedy Mixed Perturbations

Cell placement though usually modeled and implemented in the Simulated Annealing heuristic has similarities to other common algorithms. One such algorithm is the N-body problem or springs model where the forces between all related components are summed for each component and their positions are changed based on its individual zero-force location. This can be envisioned as either a group of bodies which experience gravitational attractions to each other or a group of bodies interconnected by springs, in either case in general there will exist a force vector each body experiences causing it to move.

For each net in a design, the member components will be "drawn" towards each other in trying to reduce the wire length required to interconnect the net. Whereas SA produces move attempts that will be rejected thus increasing the number of wasted processing cycles, an N-body implementation will utilize every calculation. This increased utilization comes at the cost of non-scalability in general as the N-body problem is inherently $O(n^2)$ [36].
With respect to placement, modeling wire-length reduction is a simple matter of calculating an attractive force between all members of a net and for each component the sum of all forces of each net in which it shares membership. Modeling repulsive forces such as overlap and wire congestion is not as simple; calculating the optimal force to represent cost reduction is not obvious. For these reasons this method is not as efficient as others if used exclusively.

[18] looks to randomly intermix traditional move attempts and greedy moves based on the sum of force calculations. When a greedy move is performed, a sum-force vector is calculated for a component which is then moved accordingly. An amount of success is seen with this research and shows promise as at any point during an SA-based placement run these greedy moves are presented allowing the design to converge more quickly. This concept is not explored in this work as the focus here is to analyze the traditional SA algorithm and accelerate it via a hardware implementation though the concept discussed in this section lends itself quite easily to this hardware acceleration method.

### 5.4.4 Multiple Heuristic Combination

A very interesting area of research in placement algorithms is the combination of different methods utilizing the strengths of each to lessen the impact of the others' weaknesses. The work performed in [20] focuses on augmentation of a Genetic Algorithm with SA and Tabu Search stating that results are better than each algorithm alone. [29] proposes to begin SA-based placement after operating on the high energy solution with a min-cut based algorithm. Results show convergence on the same cost value as traditional SA while decreasing run time through parallelizing of both types of placement.

Though only traditional SA is examined, implemented and modeled in this work, it was independently discovered in the software implementation that, if parallelized, it would be logical to assign one processing path to separate partitions of the design during very low energy placement. This is different from [29] in that parallelism moves components outside of the processing entity’s partition and communicates this to the receiving entity.
It will be proposed in this work that two rounds of placement will occur. During the first round parallel moves placement will extract most of the cost decreasing moves on a die wide scope. The second round will implement the placement procedure on an individual partition as if it were starting from the beginning, i.e., an accelerated temperature schedule will be reset and placement will occur within individually bound partitions.
Chapter 6

Software Implementation

6.1 Software Revisions and Lessons Learned

In order to study and characterize the Simulated Annealing algorithm when applied to placement optimization, a C-language program was implemented. The program uses the LEF/DEF file formats as its input and outputs, manipulating actual designs in the same manner, though not exactly, as commercial PDA tools. The software implementation of the SA placement algorithm went through several phases as lessons were learned and applied to following revisions.

6.1.1 Experimental Code

This work began with the definition of the data structures to be used and their interactions. As a design consists of nets and their member components, structures representing these elements are defined and populated with data required to manage the placement algorithm. Figure 6.1 provides a graphical overview of this initial organization. From this overview one can see that a central list of components is maintained by the design from which random components are chosen during the move phase. The design knows of the target die’s height and width, the aforementioned component list and the netlist. The netlist knows of only the number of and each individual net, each net in turn knows its constituent member
components. A component maintains its own information regarding size and current position; the relationship between components in the software and in the actual design is one to one meaning that only one data structure may represent a component. This holds true for nets, the netlist and design structures, no duplicate data is created and the minimum number of data structures is maintained with pointers being employed to reference a single structure from multiple locations. In this manner the memory footprint of the program is minimized and data coherence is maintained. Additionally, the placement algorithm allows components to be placed at any x-y coordinate whereas most placement tools define rows in which components will reside due to the construction of the technology library components. This was deliberately chosen to allow research to apply more generally to the SA algorithm though giving that comparison to timing related results of most others using the same benchmark circuits would be rendered irrelevant.

The algorithm used to implement SA during this stage of experimentation followed the flowchart in Figure 5.2 save for one difference, instead of only calculating the change in
cost after a move, the entire design's cost was calculated. A very small test circuit consisting of three components was used to validate functionality of the software and scalability issues were not considered at this point. After correct behavior with respect to wire-length estimation via the HPWL method was observed it was immediately noticed that the naive solution was generated by placing all components on top of each other; obviously an overlap penalty would be required. Having defined a cost function consisting of wire-length and overlap estimations the algorithm then behaved correctly and generated acceptable placement solutions.

6.1.2 Initial LEF/DEF Placements

The LEF/DEF file reader was created to facilitate reading designs from these formats in order to begin characterizing the SA algorithm with larger designs. It was quickly noticed that the program took an inordinate amount of time to place a moderately-sized design, this was quickly attributed to recalculating the cost of the entire design where only recalculation of nets which contained a modified components was required. This brought a change in that every component would then register the nets of which it was member such that when it was moved it could notify its member nets that their wire-lengths should be recalculated. This modifies Figure 6.1 in that each component would then contain a list of pointers to each net it is a member of, each net would cache its cost such that instead of recalculating, it would reference the unmodified value, if able.

This provided good performance though still noticeably not scalable, it became obvious that all decoupling from a design's size would be necessary. At this point, during a cost update each net was polled for its status with respect to requiring wire-length recalculation; this act was simply very inefficient. Additionally, the overlap calculation became the limiting factor after making the other functions more efficient. Solving this involved conditionally checking overlap only for components that moved or were known to be previously overlapping another component.
6.1.3 Benchmarks

At this point the ICCAD04 benchmark suite \([4]\) in LEF/DEF format was used to test the software implementation with mixed results. It was apparent that due to the size of the benchmark circuits and the order of the problem, any tie to design size would result in poor scalability. The entire codebase which began as the experimental implementation, mentioned above, was torn down and re-written to leverage all knowledge gained thus far. This implementation represents a fair model of a placement tool using SA as its core algorithm with the exception of not implementing row-based placement, discussed above.

To decouple size of the design from the algorithm implementation, two more lists were created, one to hold pointers to the nets requiring an update after a move and one to hold pointers the components which currently have some amount of overlap. Additionally, the running cost of the solution was decoupled from moves and only the change of the cost is calculated; the overall cost is then only incrementally updated after the move decision is made. Thusly, only data directly related to a move is ever accessed and manipulated during that cycle with the exception of the overlap calculation. It was recognized that calculating the overlap involved checking a component’s position against every other component in the design, this was resolved by partitioning the design to limit the number of components which are required to be compared. A much further analysis of the mature implementation follows in the following section.

6.2 Logical Modules

In order to analyze the SA algorithm to characterize its behavior it is necessary to logically modularize the implemented software. Below is a graphical representation of the various modules in the software placement implementation. Each module is represented by number of function calls in the program, all modules except the LEF/DEF reader and writer exist in the main function of the placement implementation.
6.2.1 LEF/DEF Reader & Writer

Though not examined for performance analysis, modularization of the LEF/DEF readers and DEF writer is only logical. As other design and library definition formats exist, it is only a matter of exercise to produce another module capable of translating from that format to the data structures used in the placement program.

In order to produce the populated data structures on which the placement program operates, an LEF file is required from which component definitions can first be read. The LEF reader builds a list of all components that may be used in the design whose instantiations will be read from the DEF file. As the DEF file is read, each instantiation will look up that particular component’s information from the list built by the LEF reader. All component instantiations are read and a master component list is created for the design. As the netlist is read each member component in each net is found in the master component list and referenced, each component then knows the nets of which it is a member. After all nets have been read, the DEF reader then passes the populated data structure back to the main
function, which is then ready to perform placement.

6.2.2 Design Perturbation

The perturbation function is the combination of two different perturbation methods, specifically, random positioning of a randomly chosen single component and the interchange of the positions of two randomly chosen components. The selection of either of these methods is randomly biased, the method which is currently performing better will be chosen more often than the other, but not to exclusion, thus giving the poorer performing method a chance to increase its success.

Components are chosen by first randomly selecting a net, then a member component of this net. First choosing a net forces the algorithm to equally consider all nets for perturbation as opposed to choosing a component directly which may unfairly bias some nets over others due to the design’s connectivity. Having chosen the appropriate number of components, one for random positioning, two for interchange, the algorithm then appropriately modifies the position of the component(s). If single-component positioning is chosen, the x-y coordinates of the component are assigned by randomly choosing a point which lies inside of a limiting radius which is calculated as a function of the current system temperature. If interchange is chosen as the method of perturbation the two components’ coordinates are simply interchanged. Each method’s success counter is incremented if a cost reducing move is found in that attempt and reduced if the move is rejected, no change is made for accepting a cost increasing move.

It is possible that if two components are chosen for interchange their sizes will not be identical. In this case, one component will likely not experience overlap (depending if the other component originally experienced overlap) while the other will while moving into its new position. This stresses the importance of allowing overlap to exist in intermediate solutions. Allowing this will give that wirelength reducing moves which generate overlap (likely to be eliminated with future moves) are accepted. The overlap penalty factor should then be set by the amount of overlap is acceptable per unit of wire-length reduction.
As part of the perturbation process the nets of the component(s) which have been moved are flagged to indicate that their recorded cost is now out of date and must be recalculated. This is simply performed by iterating through the list of member nets contained in each component’s data structure and adding each net to the cost update list. Additionally, each component which has been moved must be checked for overlap, a running list of overlapping components is kept and components involved in the current move are added. After this has been completed the design is then ready to be evaluated for the change in cost generated by the perturbation.

6.2.3 Wirelength Estimator

The main component of the cost evaluation function is the wirelength estimator which calculates the minimum bounding box that encloses all members of a net and takes half the perimeter as the estimate of the wire-length which would be required to interconnect the net, the HPWL estimate (a small example is given by Figure 5.3). For two and three component nets, while knowing port locations, this is an exact calculation [36] while for higher order nets the approximation is acceptable for the amount of calculation required – at this point the exact routing is not known. This algorithm does bias components connected with buses or more generally does not consider the degree of interconnectivity of each component. A net’s wirelength estimation is simply added to the total without regard to if it is a member of a bus. This gives that the higher the bit-width of the bus, the more closely its member components will likely be placed; a cost reducing move will be considered much more desirable due to reducing the length of all the bus wires.

For each net in the update list this approximation is performed and the previous cost is subtracted from the current approximation; the previous cost is recorded in case the move is rejected. The total change for all nets is accumulated and recorded as the change in the wire length approximation.
6.2.4 Overlap Calculation

Overlap is an important cost factor as it prohibits components from finding lower cost positions by moving over top of others. If not prohibited from doing this, the lowest cost solution will be found by placing all components on top of each other, clearly not an acceptable solution. As stated above, a running list of components which are experiencing overlap is kept such that they can be examined after each move to see if any components have moved away from them. One does not know if a component which has been moved to another position is overlapping other components so it is added to the list during the move. This is required by the nature of the problem, components may have different sizes so interchanging two components’ positions may produce overlap and, obviously, randomly moving a component may cause overlap.

For each component in the overlap list, overlap is examined against any other component (or portion of a component) which resides in the same partition. This method attempts to limit the coupling between the amount of calculation required to detect overlaps and the size of the design. If components are found in the same partition, the boundaries of each component are calculated and compared to those of the component in the overlap list. If overlap is detected the offending area is calculated and added to an overlap accumulator. Using this method it is possible to count overlap double if both overlapping components’ overlap are calculated considering the other. To prevent this, components are assigned a rank value simply by their order as read in by the DEF reader, overlap between two components is only calculated by the lower ranking component. If no overlap is detected for a component, it is removed from the overlap list as it has been verified not to overlap any other component.

Having calculated the wire-length and the overlap values, the cost of the move is then given by

\[
C_{\text{move}} = C_{\text{wirelength}} + F_{\text{overlap}} \times C_{\text{overlap}}
\]  

(6.1)

where \( F_{\text{overlap}} \) is an overlap weighting factor which determines how severely it will be penalized as accepting some interim overlap may be beneficial. \( C_{\text{move}} \) is the value passed
to the move decision module and represents the change in cost of the design due to the current perturbation.

6.2.5 Move Acceptance

This module is simple in concept but represents the essence of SA, the unconditional acceptance of cost decreasing moves and the possibility of accepting cost increasing moves based on the current system temperature. The change of the design's cost is generated by the combination of the previous two modules and represents the factor by which the move(s) will be analyzed for acceptance.

Any negative change in cost (i.e., a “good” move) is unconditionally accepted while a positive change in cost is not unconditionally rejected. A factor given by the positive change in cost and the current system temperature is calculated by Equation 6.2 (reiterated from Equation 5.1, this factor is then a value between 0 and 1 while a random value in this same range is produced by a call to the random number function \texttt{rand()} for comparison.

\[
e^{-(\frac{\Delta C}{T})}
\]  

(6.2)

If the random value is less than the calculated factor, the move is accepted regardless of its cost. It is easily seen that during the high temperature phase of placement, moves that increase the design’s cost are readily accepted. As the temperature is reduced, the size of the cost increase that will have a certain chance of being accepted is also reduced. This gives that large increases will be less likely to be accepted beginning at higher temperatures while small increases will experience this at lower temperatures. If the move is not accepted under the above two circumstances it is rejected and any changes that have been made are undone.

6.2.6 Design Update

After the decision has been made to either reject or accept the move, the design must be then updated accordingly as some change to the design has been made during the move
attempt. If it is decided that the move is to be accepted, all that is required to be performed is to check the overlap list for and remove components which no longer exhibit overlap. If the move is to be rejected, the design needs to be reset as if the move was never attempted. This involves not only moving the component(s) back to their original position(s), but also iterating through the list of modified nets and resetting their cost, iterating through the list of overlapping components and resetting any changes made to their overlap status and finally resetting the design's overlap cost which is recorded through iterations. In both cases the list of nets to be updated is deleted as this point represents the end of the move attempt.

6.2.7 Temperature Update

This module is insignificant in terms of processing time however in the interest of implementing alternative temperature schedule types and the fact that the temperature schedule greatly influences the behavior and quality of the placement, it deserves a place distinct from the rest of the algorithm. Inside of this work a simple temperature schedule with three regions is implemented, the temperature is degraded by two factors determined by the user of the program. One factor is used at the outer regions of the schedule while the other is used in the middle region. Typically, the temperature degradation factors and the borders which determine their use are set such that a high temperature zone is quickly transversed to a point where moves begin to reduce the overall cost of the design. From this point the temperature is slowly degraded until a point where moves are mostly rejected, where again, the temperature is quickly degraded as in the high temperature region. Figure 6.3 gives a unit-less sample temperature schedule as described above. A more in depth analysis is given the following sections.

6.3 Temperature Schedule

One of the most important lessons learned during experimentation is the proper definition of the cooling schedule in order to maximize the cost reduction for each temperature step.
It was observed that if too much time is spent in upper temperatures, placement attempts are wasted due to an inordinate amount of cost increasing moves being accepted. At this point in the temperature schedule the design can be thought of as liquid and placement is simply randomizing the initial solution. If the design is cooled too quickly, the algorithm tends to get trapped in local minima that would otherwise be avoided if the design was allowed to cool slowly. This allows cost increasing moves to be accepted, moving the design to new points potentially allowing previously unavailable cost reducing moves. If the design is cooled too slowly, each temperature step will reach a point where no further cost reductions are seen; the algorithm converges on a cost which is then maintained by the combination of acceptance of cost increasing moves and discovering cost reducing moves.

In order to correctly model the system as a physical material it is required to limit a component’s movement as a function of temperature. When the system temperature is high, a component is free to explore most of the die area, as the temperature is reduced, so is the radius of a potential move. If logically analyzed, it is clear that while placing on a die-wide scope components will be grouped for lower cost but not placed efficiently with respect to other components in the same group, performing coarse placement. By reducing the scope of a potential movement, the chances that a component will be placed to reduce
cost within its net(s) is increased, performing placement on a finer scale.

By and large, changing the cooling schedule has the most effect on the efficiency of the algorithm, implementation and tuning of good schedule was not a trivial effort. Building some intelligence into the temperature schedule and allowing it to modify itself using knowledge from the placement run (cost change slope over the life of the placement, total cost improvement, etc.) could prove to yield some enhancements in the efficiency of the algorithm. This modification is otherwise known as adaptive SA and is outside the focus of this thesis.

6.4 Benchmark Circuits

In order to provide an algorithm characterization test set the IBM-MSwPins benchmarks from the ICCAD04 mixed-size placement benchmark suite [4] are employed. These benchmark circuits originated in 1998 as the ISPD98 circuit benchmark suite [6], produced by IBM after MCNC, a non-profit research organization who produces benchmark circuits for use by the Design Automation community, did not follow industry trends with respect to overall design size and component mixture. The ISPD98 benchmarks were further refined in 2002 under the work performed in [3] though defining actual cell dimensions having non-trivial aspect ratios where only cell area was previously defined. The ICCAD04 [2] benchmarks further refined the ISPD98 benchmarks by defining pin locations for each component in the designs. In the previous two benchmark versions the pin locations for each component were placed at the center of the component, forcing an approximation wire-length calculations.

The benchmark sizes range from 13k to 210k components across 18 circuits, approximating the range of sizes with respect to actual designs seen in industry. The base benchmark set has been cited in many other works and represents a standard assessment of the quality of a PDA placement algorithm/tool. In this work, only the first four circuits were used in order to reduce processing time; specific component and net count along with the
average number of components per net is given by Table 6.1.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Components</th>
<th>Nets</th>
<th>Average Components per Net</th>
</tr>
</thead>
<tbody>
<tr>
<td>ibm01</td>
<td>12752</td>
<td>14111</td>
<td>3.58</td>
</tr>
<tr>
<td>ibm02</td>
<td>19601</td>
<td>19584</td>
<td>4.15</td>
</tr>
<tr>
<td>ibm03</td>
<td>23136</td>
<td>27401</td>
<td>3.41</td>
</tr>
<tr>
<td>ibm04</td>
<td>27507</td>
<td>31970</td>
<td>3.31</td>
</tr>
</tbody>
</table>

Table 6.1: Benchmark ibm01 – ibm04 Information

### 6.5 Characterization & Optimization

From Figure 5.2 that it is clearly seen that at the core of the algorithm is perturbing the design, calculating a move’s cost and making a decision based on that cost; a large amount of program execution time will be spent inside of these steps. Characterization was performed using the first four *IBM-MSwPins* benchmark circuits and the program execution profiling tool, gprof. Table 6.2 gives the percentage of the total time of program execution for the most significant modules. Obviously, calculation of the change in cost for a move represents a large majority of time spent in order to place a design with the smallest value being 93% of all execution time for benchmark ibm04 (HPWL + Overlap Detection). It should be noted that time not central to the execution of the algorithm is accounted for in the total time, this includes LEF/DEF input and output as well a visualization in FIG format.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>HPWL</th>
<th>Overlap Detection</th>
<th>Perturbation</th>
<th>Decision</th>
<th>Others</th>
</tr>
</thead>
<tbody>
<tr>
<td>ibm01</td>
<td>75.9</td>
<td>21.6</td>
<td>0.2</td>
<td>0.2</td>
<td>2.3</td>
</tr>
<tr>
<td>ibm02</td>
<td>73.4</td>
<td>21.0</td>
<td>1.0</td>
<td>0.6</td>
<td>4.0</td>
</tr>
<tr>
<td>ibm03</td>
<td>79.7</td>
<td>15.2</td>
<td>1.1</td>
<td>0.7</td>
<td>2.4</td>
</tr>
<tr>
<td>ibm04</td>
<td>83.1</td>
<td>10.1</td>
<td>1.6</td>
<td>0.9</td>
<td>4.3</td>
</tr>
</tbody>
</table>

Table 6.2: Benchmark ibm01 – ibm04 Time Profiles (% total)

Figure 6.4 gives the number of seconds spent in each module of the algorithm during program execution on a logarithmic scale for each benchmark. Again, it is clearly shown that the time required to calculate a move’s cost change accounts for nearly all of the core
Figure 6.4: Software SA Placement Time Profile

It can be seen that the general trend for the behavior of the algorithm remains constant as benchmarks increase in size. Obviously and logically, any speedup enhancements in hardware would be best targeted at the cost evaluation and move decision modules of this algorithm.

Table 6.3 gives the average processing time required to calculate one cost delta on a 2.2 GHz AMD Athlon64 FX-51 general purpose processor (GPP) based workstation running Gentoo Linux 2005.0 with one gigabyte of memory; the executable was compiled with GCC 3.3.5. Each benchmark's placement execution calls the cost delta function the same number of times as an identical temperature schedule was used for all placements; in general this would not be the case as the characteristics of each benchmark would determine the temperature schedule. Table 6.4 gives the composite requirements of each benchmark.
Table 6.3: Cost Delta Function Time Analysis

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Function Calls</th>
<th>Clock Time (s)</th>
<th>Average Time per call (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ibm01</td>
<td>2723000</td>
<td>11274.74</td>
<td>0.0041</td>
</tr>
<tr>
<td>ibm02</td>
<td>2723000</td>
<td>7998.13</td>
<td>0.0029</td>
</tr>
<tr>
<td>ibm03</td>
<td>2723000</td>
<td>10144.42</td>
<td>0.0037</td>
</tr>
<tr>
<td>ibm04</td>
<td>2723000</td>
<td>10523.83</td>
<td>0.0039</td>
</tr>
</tbody>
</table>

in terms of components and nets required to be processed via HPWL and overlap detections per average cost delta call. Benchmark ibm03 shows an inordinate amount of overlap detection calls per cost delta calculation as compared to the other benchmarks. It is surmised that the composition of the particular design dictates this particular behavior. This can arise from the existence of different-sized components that when moved may overlap many other components and vice-versa, or alternatively the overall component area to die area ratio may be higher than the other benchmarks giving that more intermediate overlap will be seen during placement. Whatever the cause of this behavior it serves to show that placement behavior from one circuit to another cannot be easily predicted and placement behavior should not be over-generalized.

Below, in Figure 6.5 a graph of the benchmark’s cost during the placement process and

Table 6.4: Average Cost Delta Function Composite Analysis

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Nets Approximated</th>
<th>Bounding Boxes per call</th>
<th>Overlapping Components per call</th>
<th>Overlaps Checked per Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>ibm01</td>
<td>8.13</td>
<td>29.10</td>
<td>8.21</td>
<td>11.90</td>
</tr>
<tr>
<td>ibm02</td>
<td>8.03</td>
<td>33.33</td>
<td>9.04</td>
<td>7.00</td>
</tr>
<tr>
<td>ibm03</td>
<td>10.50</td>
<td>35.80</td>
<td>9.55</td>
<td>119.80</td>
</tr>
<tr>
<td>ibm04</td>
<td>16.01</td>
<td>53.01</td>
<td>9.02</td>
<td>6.61</td>
</tr>
</tbody>
</table>

shows how the design’s cost decreases exponentially as the algorithm operates. Very good gains are seen early on after the algorithm has reached a breakpoint temperature (where the design moves from a “liquid” to a “solidifying” state) but the cost reduction for each temperature step is decreased with the cost tangentially approaching a convergence value.
Taking the above observations into account it is proposed that a hardware architecture implementing multiple simultaneous movements may improve the algorithm's efficiency in a range from die-wide to net-limited movements. Evaluating a number of movements simultaneously with a scope of the whole design would find those resulting in the quickest reduction in cost. After the system has reached a point at which die-wide movements are no longer efficient to find, the simultaneous movement architecture can be used to focus on smaller independent sections of the system in order to find movements which will result in further cost reductions. At this point the design is partitioned such that each element of the simultaneous movement architecture may operate exclusively its own die section as if the algorithm were starting from scratch on a smaller scale. The simplest method is to subdivide the design and assign each partition to a single processing element. This architecture would ideally find the movements formerly obscured by searching the entire scope of the design to produce a faster convergence – to reduce the exponential quality of the cost vs. iteration chart in Figure 6.5 making it more linear thus requiring less program execution time.
6.5.1 Benchmark Visualizations

Below, Figure 6.6 gives visualizations of the benchmark ibm05 before (top) and after (bottom) placement; the dark areas represent components. The top visualization gives the benchmark after it has been initially placed by assigning random positions to non-fixed components. This was simply performed outside of the placement algorithm in order to give a repeatable starting point for the software placement algorithm. Normally, this initial placement would be produced with related components near each other in order to “jump start” placement with knowledge from previous EDA steps.

The bottom visualization gives the placement of the components after the software program has been executed. Clearly shown is the centered pattern generated by the free-placement, i.e., not row-based placement. The density of the components tapers from the center of the die because of fixed components around the periphery of the die implementing pad functionality. Components which are members of these nets will find lower cost positions between the fixed pad components and the other members of its net. This will in turn move other components out toward the periphery of the die resulting in the density gradient seen in the final placement.

The final placement gives the impression that the die could be smaller while allowing space for all components. It should be noted that the final placement given here is not an acceptable placement as per the design’s row-based technology. If placed in rows, the visualization would seem less dense (requiring more die area) while maintaining a comparable HPWL value.
Figure 6.6: Initial and Final Placements of Benchmark ibm05
Chapter 7

Hardware Model

In taking all of the above research results into account, it is clear that in order to accelerate the SA algorithm focus should be placed on the cost calculation and move decision modules. The following hardware model is proposed in Figure 7.1; a custom calculation path for both the wire-length estimation and overlap detection. To support this data path and to allow parallelization, a move decision module is implemented such that a move can be evaluated, giving a simple binary answer. This hardware is envisioned to be driven by a controller that will perform the selection of the type of move and component(s) and receive the decision to possibly update the design.

From Table 7.1 one can see the wire-length and overlap calculation inputs, for each move in order to evaluate the change in cost component data will be fed serially to the datapath by the external control unit. Table 7.2 gives the outputs of the evaluation, simply two one-bit signals, a ready indicator and a binary decision signal, accept or reject. An early negative overlap signal is included to limit the amount of time required to return a zero overlap result. Also included in the output group are two values, the moved component’s new overlap value and the net’s new wirelength value.

7.1 Wire Length Estimator

The main focus of the SA placement algorithm, wire-length minimization, is implemented in a structure which uses the HPWL approximation, discussed above. To accomplish this,
the structure given in Figure 7.2 receives component data serially after being converted by logic higher in the hierarchy, namely, the coordinate values of the component’s bounds. The module then compares the received values to the currently held values defining the bounding box; if any of the received values create a larger box they are latched and used in the calculation. After comparing all components in a net, the estimator will then have the smallest bounding box containing all components in that net. This value is accumulated and the latches holding the bounding box values are reset enabling the next net’s components to be compared. Each net’s original HPWL is subtracted from the overall accumulated value giving that the final result after all nets have been evaluated is the move’s change in wire length.
<table>
<thead>
<tr>
<th>Port</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OFACTOR(15:0)</td>
<td>Overlap cost multiplication factor</td>
</tr>
<tr>
<td>O1SIZEX(31:0)</td>
<td>Overlap detection, component 1 width</td>
</tr>
<tr>
<td>O1SIZEY(31:0)</td>
<td>Overlap detection, component 1 height</td>
</tr>
<tr>
<td>O1SWX(31:0)</td>
<td>Overlap detection, component 1 SW corner X coordinate</td>
</tr>
<tr>
<td>O1SWY(31:0)</td>
<td>Overlap detection, component 1 SW corner Y coordinate</td>
</tr>
<tr>
<td>O2SIZEX(31:0)</td>
<td>Overlap detection, component 2 width</td>
</tr>
<tr>
<td>O2SIZEY(31:0)</td>
<td>Overlap detection, component 2 height</td>
</tr>
<tr>
<td>O2SWX(31:0)</td>
<td>Overlap detection, component 2 SW corner X coordinate</td>
</tr>
<tr>
<td>O2SWY(31:0)</td>
<td>Overlap detection, component 2 SW corner Y coordinate</td>
</tr>
<tr>
<td>PREV_NETC(31:0)</td>
<td>Previous cost of the net</td>
</tr>
<tr>
<td>PREV_OC(31:0)</td>
<td>Previous cost of overlap</td>
</tr>
<tr>
<td>WSIZEEX(31:0)</td>
<td>Wirelength estimation, component width</td>
</tr>
<tr>
<td>WSIZEY(31:0)</td>
<td>Wirelength estimation, component height</td>
</tr>
<tr>
<td>WSWX(31:0)</td>
<td>Wirelength estimation, component SW corner X coordinate</td>
</tr>
<tr>
<td>WSWY(31:0)</td>
<td>Wirelength estimation, component SW corner Y coordinate</td>
</tr>
<tr>
<td>CLK</td>
<td>Datapath clock</td>
</tr>
<tr>
<td>OCLK</td>
<td>Overlap detection clock</td>
</tr>
<tr>
<td>ONEW_COMP</td>
<td>Overlap detection new comparison signal</td>
</tr>
<tr>
<td>RESET.L</td>
<td>Datapath reset</td>
</tr>
<tr>
<td>WCLK</td>
<td>Wirelength estimation clock</td>
</tr>
<tr>
<td>WNEW_COMP</td>
<td>Wirelength new component signal</td>
</tr>
<tr>
<td>WNEW_NET</td>
<td>Wirelength new net signal</td>
</tr>
<tr>
<td>SEED(4:0)</td>
<td>Pseudo-random number generator seed value</td>
</tr>
<tr>
<td>TEMP(15:0)</td>
<td>Current system temperature</td>
</tr>
<tr>
<td>LCLK</td>
<td>Acceptance logic clock</td>
</tr>
<tr>
<td>LOAD</td>
<td>Pseudo-random number generator seed value load signal</td>
</tr>
</tbody>
</table>

Table 7.1: Top Level Hardware Input Interface

7.2 Overlap Detector

In order to prevent or limit accumulation of disruptive overlap this module evaluates the overlap generated by a move. An alternative method of overlap detection is performed in order to account for the amount of memory manipulation required in the software implementation. Each component will maintain its own overlap status and design wide overlap detection will not be performed. Since the overlap calculations can be performed in parallel alongside the wire-length estimations, this will not result in additional processing.
<table>
<thead>
<tr>
<th>Port</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCEPT_H</td>
<td>Move acceptance signal</td>
</tr>
<tr>
<td>READY</td>
<td>Move acceptance decision ready</td>
</tr>
<tr>
<td>OC_NEW(31:0)</td>
<td>New overlap cost</td>
</tr>
<tr>
<td>OGONOGO</td>
<td>Early negative overlap indication</td>
</tr>
<tr>
<td>NETC_NEW(31:0)</td>
<td>New net cost</td>
</tr>
</tbody>
</table>

Table 7.2: Top Level Hardware Output Interface

![Diagram of wirelength estimator RTL schematic](image)

Figure 7.2: Wirelength Estimator RTL Schematic

time. Any component which has been moved will be compared against other components in the partition gaining the moved component's membership. In order to detect overlap, the bounds of the two components to be compared are checked in parallel along with calculating the amount of overlap. It is possible that there will be no overlap between the components, in this case extra cycles to compute the overlap value are avoided by the overlap detection comparators; if no overlap is detected, the next component is immediately checked.
7.3 Move Acceptance Logic

The move acceptance logic module implements the software’s decision procedure in a straightforward pipelined method. Equation 5.1 is manipulated to a point where a division between the change of cost and the current system temperature is not required, this manipulation results in the following

\[ \Delta C < -T \times \ln(\text{rand}(0, 1)) \]  

which then only requires producing a random value to look up a value of natural log (ln), rather than calculating it. This look-up value is multiplied by the system temperature to give the value which the change in cost is compared against in order to make a move acceptance decision.
To maintain a low maximum clock cycle period, this calculation is pipelined and parallelized. The first stage of the pipeline both checks if the change in cost is negative and performs the multiplication between the lookup value of \( ln \) and the current system temperature. If the cost change is negative, a move acceptance signal is returned and no further processing occurs. If the cost change is positive the next stage of the pipeline is utilized wherein the multiplied factor is shifted to account for the fixed point precision of the lookup value of \( ln \). The third pipeline stage, if used, then performs the comparison in Equation 7.1 and returns an appropriate signal accepting or rejecting the current move.

![Figure 7.4: Move Acceptance Logic RTL Schematic](image)

### 7.4 Interfaces

This model only represents the datapath of the modules requiring the most calculation time in the serial, software algorithm; there is no control logic, memory interface or overall program implementation. Abstractions of these are discussed in order to address issues that
may be present in an integrated system that would degrade performance as analytically described here. The overall hardware model including the interfaces to be described is given in Figure 7.5. It is not the focus of this work to thoroughly define a complete implementation of the hardware accelerated placement algorithm and some descriptions of these interfaces are intentionally vague.

![Overall System Model](image)

**Figure 7.5: Overall System Model**

### 7.4.1 Datapath Control

Most noticeably, this datapath contains no driving logic, a module would be required to manipulate the inputs and observe output signals. This control would then implement the functionality of the design specified here, as this would not be a very complex task, a finite state machine (FSM) would be used. This module would be in charge of marshaling data
to and from the datapath but not necessarily in charge of managing it, this responsibility falls to the central controller. As it is an action simply parallelized, generation of the perturbation type and random values in order to choose components is handled by this control. These values are generated when needed and passed to the central controller which then fulfills the request for all component data associated with the particular move.

To hide the latency of bringing component data from central memory to the datapath, the controller will maintain two sets of move data, one active and one queued. As the active set it being operated on (cost delta calculation and acceptance decision) the queued data will be filled into the datapath's local memory. This will eliminate the time between when a decision is made with respect to the current move and the next move data is ready. Having the next move's data immediately available will increase overall utilization of the datapath by reducing idle time.

### 7.4.2 Component Supply

As it is desirable to keep the data of the design as close to the processing path as possible and designs may require several megabytes to reside in memory, the data in general will not be able to be retained on the FPGA. This will require a GPP in order to effectively and efficiently perform the required actions to properly move component data from memory to datapaths and to update the design after processing. Current FPGAs support the integration of so-called soft processors (or IP cores) which implement microprocessor functionality and furthermore, some FPGA families include true microprocessor cores such as the PowerPC. The use of a GPP is ideal in order to satisfy all other processing requirements besides those accomplished in the parallel datapath; Xilinx's Virtex-II Pro is chosen as the target in this work as it incorporates the PowerPC core as well as traditional FPGA resources.

As each datapath will require its own access to memory, having all paths ready to receive data at the same time would not be desirable. This would result in a low amount of utilization while each datapath awaits the data on which it is required to operate. It is
proposed to implement a mechanism to intelligently schedule data movements from design memory to local controller memory, staggering access will result in only one datapath waiting for memory at one time. The central controller knows what each datapath is operating on and when to expect it will need another input of component information in order to keep it fully utilized. Knowing this, it is simple to refill the queued data memory of the datapath which is going to require it first. In this way, no artificial data starvation is created, this will only occur if one is unlucky enough to encounter a condition where calculations are complete before data is available. Of course, as one increases the datapath parallelism the memory utilization will increase and these situations will become more common. There will likely be a point in which additional parallelism will not contribute to a larger speedup. After the GPP and memory are fully utilized by a number of datapaths, further increases will only decrease the utilization of each individual datapath.

7.4.3 Design Update & Algorithm Management

As stated above, all other processing required by the SA algorithm is performed by the central processor, this includes updating the system temperature and component information which is produced by the parallel datapaths. More importantly the central processor would also manage the interface between the host system, reading and writing to a system bus which communicates the design data. It is envisioned that the host system software will read the design input and configuration files, setting initial parameters for the placement run. This includes pseudo-random number generator seeds such that subsequent placement runs can be repeatable if the design is changed slightly by the designer.

The GPP core will run at a faster clock rate than the core datapath given that it will be able to accomplish processing required to support placement. The number of parallel datapaths would be limited by the amount at which the GPP becomes the new performance bottleneck.
Chapter 8

Method of Investigation

One of the goals of this work is to show that the core of the Simulated Annealing algorithm and in turn VLSI placement can be accelerated using parallel and advanced implementations. Since a complete hardware based system is not offered as part of this work, the following method is used to analyze experimental investigation, given in Figure 8.1.

Since the LEF/DEF based ICCAD04 placement benchmarks [2] are used in this thesis the test method does not utilize the design entry or synthesis steps directly. This was performed by those generating the benchmarks, though if it were desired to use one’s own designs these steps would be required. Having the LEF/DEF files describing the designs to be placed, the performance of this work’s software placement tool is compared against that of a common tool. Capo [2] is chosen as the tool for comparison as its performance was shown to be superior to other tools when operating on the ICCAD04 benchmark suite. [2] gives the execution time and HPWL metrics for Capo’s placement while these values are extracted from executions of this work’s experimental placement tool for comparison. Analysis of the hardware assisted SA placement tool’s is performed using the information generated from the Xilinx IDE (maximum clock frequency and clock cycles required to perform calculations) and the software placement tool’s execution. An analytical speedup is calculated from both the hardware and software implementations’ characteristics. The speedup and placement quality (HPWL) results are then compared to Capo’s performance.

Since the software implementation in this work does not implement row-based place-
Figure 8.1: Test Method

ment it is impossible to compare its results to any other’s research results with direct correlation. This is not detrimental to validating the results of this work as if it can be shown that the algorithm was enhanced through the proposed methods, then implementing another form of placement is simply a matter of changing the perturbation and possibly other functions to produce desired behavior and further enhance performance, respectively.
Chapter 9

Results

This chapter is organized to first present the experimental software placement tool’s performance and compare it to another tool presented in [2]. This provides correlation to the quality of the placement solutions but not so much to the execution time required which can be accounted for in the fact that the experimental software tool implemented in this work does not perform row-based placement but free-area placement which greatly increases the problem’s search space. The next section presents the datapath’s clock cycle requirements in terms of move evaluation calculations, maximum clock cycle and resource utilization as determined by the FPGA synthesizer. Finally, using the characteristics from the software implementation’s placements of the benchmark circuits, cycle requirements for the FPGA are extracted and the execution time per move cost evaluation is compared.

9.1 Software Implementation

In comparing the performance between the software placement tool implementation performed in this work and Capo, implemented in [2], one has to consider the placement methods implemented. Specifically, Capo implements row-based placement as defined by the benchmark’s technology whereas this work’s placement tool implements free-area placement in which components are allowed to be placed anywhere on the die. The search space disparity between the two methods is directly responsible for the difference between execution times of the two tools. However, the final HPWL total given by the two tools shows
that given time to explore the search space, this work's implementation is able to produce similar results. Benchmark ibm04 was not able to be placed as well as Capo's solution as the search space becomes larger than the given temperature schedule can search as effectively as the other benchmarks. The selection of an appropriate temperature schedule for the larger benchmarks would follow that starting temperature and degradation boundaries are increased. Increasing the temperature ranges at which the algorithm operates for larger designs allows the cost increasing acceptance inputs $T$ and $\Delta C$ to remain correlated. Additionally, either the degradation factors should be increased nearer to unity or the number of perturbations per temperature step should be increased to force more component perturbations. As multiple placement attempt are likely to occur, the temperature schedule can be modified after each attempt thus increasing its efficiency.

Table 9.2 clearly displays the factor of the difference in number of the points available for components to be placed inside of each benchmark. Not surprisingly, the factor between the two approaches is the same due to the same technology being used which determines the standard cell height. Though the difference factor is the same through the benchmarks, the sheer number of grid points and components which may be applied to the grid increases, greatly enlarging the search spaces (given by Equation 4.3) thereby requiring greater effort to determine an acceptable solution. Additionally, free placement has the

<table>
<thead>
<tr>
<th>Benchmark Circuit</th>
<th>Capo v9.0 -feedback placer</th>
<th>Software Placer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>HPWL (e6)</td>
<td>Time (min)</td>
</tr>
<tr>
<td>ibm01</td>
<td>2.67</td>
<td>3</td>
</tr>
<tr>
<td>ibm02</td>
<td>5.54</td>
<td>5</td>
</tr>
<tr>
<td>ibm03</td>
<td>8.67</td>
<td>6</td>
</tr>
<tr>
<td>ibm04</td>
<td>9.79</td>
<td>7</td>
</tr>
</tbody>
</table>

Table 9.1: Time Comparison to a Mature Placement Tool
overhead of checking overlap in two-dimensions (discussed above) while row-based placement must only perform this check along the row in which a component is moving. The relative doubling of ibm01’s time difference factor could be due to the higher number of overlapping components per cost delta evaluation as compared to ibm02.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Row-Based Placement</th>
<th>Free-Area Placement</th>
<th>Area Difference Factor</th>
<th>Time Difference Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>ibm01</td>
<td>838155</td>
<td>5595570</td>
<td>6.68</td>
<td>48.7</td>
</tr>
<tr>
<td>ibm02</td>
<td>1651034</td>
<td>11009108</td>
<td>6.68</td>
<td>28.2</td>
</tr>
<tr>
<td>ibm03</td>
<td>1914230</td>
<td>12773460</td>
<td>6.68</td>
<td>29.7</td>
</tr>
<tr>
<td>ibm04</td>
<td>1810475</td>
<td>12072150</td>
<td>6.67</td>
<td>26.9</td>
</tr>
</tbody>
</table>

Table 9.2: Die Grid Points per Benchmark

9.2 Hardware Model

The synthesis of the above proposed datapath in Xilinx’s FPGA integrated development environment yields a maximum clock rate of just over 200 MHz and a device utilization of around 5% for the XC2VP20 Virtex-II Pro FPGA which contains more than twenty thousand logic cells and two PowerPC cores. The cycle requirements extraction for calculations performed on the FPGA is given in Table 9.3. Using the breakdowns of the benchmarks in terms of operations required for an average cost delta calculation (Table 6.4) and the time required for these average calculations on a 2.2 GHz AMD Athlon64 FX-51, the speedup of the FPGA implementation can be calculated.

9.3 Speedup Justification

Table 9.4 clearly shows that the FPGA datapath yields several magnitudes of speedup over the GPP implementation. These values are calculated by taking the cycles required by the
Table 9.3: FPGA Clock Cycle Requirement per Calculation

<table>
<thead>
<tr>
<th>Action</th>
<th>FPGA Cycles Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calculate Net Bounding Box</td>
<td>2 cycles per component + 4 cycles</td>
</tr>
<tr>
<td>Detect Zero Overlap</td>
<td>2 cycles</td>
</tr>
<tr>
<td>Calculate Non-Zero Overlap</td>
<td>4 cycles</td>
</tr>
<tr>
<td>Accumulate Overlap</td>
<td>2 cycles per component comparison</td>
</tr>
<tr>
<td>Accept Move (Cost Delta =&lt; 0)</td>
<td>2 cycles</td>
</tr>
<tr>
<td>Accept or Reject Move (Cost Delta &gt; 0)</td>
<td>4 cycles</td>
</tr>
</tbody>
</table>

average cost delta calculation for each benchmark and finding the overall FPGA clock cycle count which is required to fully compute the acceptance decision. It should be noted that the overall clock cycle count is defined by the cost estimation module which requires the most cycles as the wire-length estimator and the overlap detector operate in parallel, the acceptance logic is a constant requirement. All of the above values are calculated in worst case, i.e., all overlap checks are taken as non-zero overlaps and all acceptance decisions are taken as non-zero costs, this provides the most conservative speedup value. These calculations cannot, however, account for memory access requirements and assume all data is present at the datapath at the time it is required. It is most likely that these speedups will not be seen in a complete system.

For the worst case benchmark, ibm03, which requires over one-hundred overlap comparisons per cost delta calculation, the speedup given is over one-thousand. This speedup depends on many factors the most important which is having data present at the local control unit when the datapath is available to operate on it. If this can be achieved for multiple
datapaths, the speedups seen would then increase as a function of this parallelization. Although, as stated above, these large speedups are not likely to be seen in a fully operational system and in order to prevent under-utilization of the datapaths, the clock speed will have to be reduced as compared to the GPP’s. The PowerPC cores inside of the XC2VP20 are able to operate at 300 MHz, reducing the datapath’s operating speed to a tenth of this value provides speedups of over one-hundred in the worst observed case.
Chapter 10

Conclusions

This work began by implementing and characterizing the Simulated Annealing algorithm with respect to component placement and optimization inside of a software program. This step allowed insight to be gained into the intricacies of the SA algorithm giving the ability to conceive a novel approach to modeling a hardware accelerated system. Having characterized and understood the serial SA algorithm, a parallelized architecture was designed allowing the SA core to be replicated, removing data dependencies thus enabling a general number of parallel datapaths. The hardware design was completed and characterized statically in order to determine clock cycle requirements used to generate timing calculations; benchmark circuits were used to provide data for an average execution of the datapath. Timings for each benchmark between the software implementation and the hardware model were compared giving the speedup that could be expected under ideal situations.

10.1 Discussion

A speedup is shown in this work for a single hardware datapath which is then proposed to be duplicated in parallel inside of an FPGA. The speedup given by this work which only focuses on the datapath of the hardware accelerated placement system clearly moves any performance restriction to another point in the overall system. Most likely this will occur at the memory management point which is now required to move all data involved in a cost calculation from design memory to the datapath controller’s local memory in the time that
it can operate on a set of data in order to be fully utilized. This problem is compounded by the fact that the hardware datapath proposed in this work is to be multiply instantiated in an FPGA, each requiring similar memory access. Clearly there will exist some amount of data starvation during placement, this will be exacerbated by the width of the parallelization inside of the FPGA; the memory access will be the limiting factor with respect to a speedup, the fastest at which data can be offered will give the maximum speed at which it can be manipulated. It is likely that the datapath will be operated at a speed much slower than the maximum found here in order to limit data starvation reducing overall speedup.

The benchmark placements used to characterize the Simulated Annealing algorithm have a large amount of disparity with respect to the amount of time required to produce similar solutions in mature tools which has been shown to be caused by the placement types implemented. The speedup results are relevant to any tool implementing Simulated Annealing as its core algorithm, the execution times given and analysis performed is on the core of the algorithm which is similar across implementations.

Overall, this work is successful in showing that the Simulated Annealing algorithm can be accelerated in an FPGA and that a hardware accelerated system may be feasible. Further research must be performed in order to accurately predict the speedup seen in this type of implementation due to memory limitations, but with the work performed here this is now warranted.

10.2 Future Work

In order to fully implement a hardware accelerated placement tool, further work must be performed to first study the characteristics of such a system, then with positive results, designed. Suggested in this section are next steps to be taken in order to bring a fully implemented system to reality.
10.2.1 Software Implementation

As this work did not implement row-based placement in order to study the Simulated Annealing algorithm more generally, execution time suffered. Other placement tools were shown to have more than forty times the efficiency than in this work. In order to produce a base to which to compare a hardware accelerated implementation it is suggested that the software algorithm implemented here is modified to place components according to the row-based grid system thus reducing the search space of the design and removing the need to perform overlap checks in two-dimensions. Having done this and performed subsequent benchmark placements it is likely that comparable execution times will be seen.

10.2.2 Control Unit

As only the processing entity of the core of the hardware acceleration path was modeled, the control structure which provides the functional behavior is required to be designed and modeled to ensure correct operation. This includes modeling the local memory which will contain data related to the active and queued moves being evaluated. After having this portion of the design, the model can be operated and characterized to gain insight to memory access requirements. This will allow the central controller to be designed accordingly in order to provide the highest memory throughput, thus resulting in the highest overall speedup possible.

10.2.3 Data Management

The data management unit will undoubtedly be implemented in the GPP core of an accommodating FPGA, at this time no other alternative is seen beyond using RAM located on the peripheral board hosting the FPGA. Having a GPP on chip will allow the central controller to operate independently of the datapath(s) thus allowing it to solely focus on administering operation of the placement. This would include moving data in and out of the memory containing the design ideally allowing the datapath(s) to remain fully utilized.
Recent research into processor-in-memories (PIM) [43] provides an interesting solution to alleviate the foreseen memory bottleneck. Otherwise known as smart memory, this architecture embeds processing logic into the memory itself such that some operations may be offloaded. This includes, but is not limited to, address manipulations in memory allowing the GPP to be utilized for other operations. It may perhaps be possible to give the memory a reference to a component selected for perturbation and have it return all data required to complete the cost delta calculation. The GPP would then be free to simply manage the data movement, keeping both the datapaths and memory fully utilized.

10.3 In Closing

Presented in this work is the first step to creating a hardware accelerated placement implementation. The speedup seen in the datapath which provides core functionality for the Simulated Annealing algorithm provides the foundation for future work. As design time becomes more important to any group creating technology instantiated in integrated circuits, so does the execution time of supporting tools. Perhaps with further research it can be shown that a hardware accelerated placement solution can be superior to traditional software implementations.
Bibliography


