A Modular Spatial FPGA Cell Placement Engine

Thesis by

Henry Barnor

In Partial Fulfillment of the Requirements
for the Degree of
Bachelor of Science

California Institute of Technology
Pasadena, California

2005
(Submitted June 3, 2005)

© 2005
Henry Barnor
All Rights Reserved

Acknowledgements

I would like to thank my advisor, Professor Dehon first and foremost for his support and guidance, secondly for taking a chance on me and welcoming me to his lab even though i lacked the necessary background. I would also like to thank Nachiket Kapre for explaining to me innumerable times how to think about VHDL code. I am also grateful to Micheal Wrighton for helping me understand his thesis work[1]. Thank you to everyone at the IC lab.

Abstract

The availability and increasing power of Field Programmable Gate Arrays (FPGAs) is causing a shift towards implementation of algorithms in spatially programmable hardware. This seems to hint that in the future basic algorithms will be implemented in programmable hardware to achieve higher performance than possible with software running on sequential processors. Before this can happen a number of key drawbacks need to be overcome. One such drawback is the time required to map the program logic to physical programmable resources anytime the machine reconfigures itself for a task.

This thesis describes the progress made on the design and implementation of a $200 Mhz$ spatial placement engine based work done in [1].

Contents

Acknowledgements iii

Abstract iv

1 Introduction 1

1.1 Introduction . . . 1

1.2 Background . . . 1

2 Problem Statement and Solution Sketch 3

2.1 Problem Statement . . . 3

2.2 Simulated Annealing Overview . . . 3

2.3 Hardware Simulated Annealing . . . 4

3 Solution Implementation 6

3.1 The Processing Element . . . 6

3.1.1 Entropy Assembly . . . 6

3.1.2 Accumulator Assembly . . . 7

3.1.2.1 CurrentCost Accumulator . . . 7

3.1.2.2 HypoCost Accumultor . . . 8

3.1.2.3 Diff Accumulator . . . 8

3.1.3 Swap Assembly . . . 8

3.1.4 Memory Assembly . . . 8

3.1.5 SwapMemory Assembly . . . 8

3.1.6 PositionUpdate Assembly . . . 8

3.1.7 Control Assembly . . . 9

3.1.7.1 Pseudo-Code for State Machine . . . 9

3.2 VHDL/Hardware Implementation . . . 9

3.2.1 Approach . . . 9

3.2.2 Analysis of Hardware Implementation 11

4 H-Tree **13**

4.1 Motivation for H-Tree 13

4.2 H-Tree Characterization 14

4.3 Simulation Results 16

4.4 Metric Analysis 16

5 Summary and Future Work **24**

5.1 Design Speed 24

5.2 H-Tree Simulation 24

Bibliography **25**

List of Figures

2.1 Systolic Placer: Array of Processing Elements . . . 5

3.1 Systolic Placer: High level block diagram of PE . . . 7

3.2 Status Block diagram for PE . . . 10

4.1 Array of PE's with Position Chain . . . 14

4.2 Array of PE's with H-Tree . . . 15

4.3 Plot of Final Metric vrs Updates per Step . . . 17

4.4 Plot of $\frac{SystolicChannelWidth}{VPRChannelWidth}$ vrs Updates per Step . . . 18

4.5 Plot of Final Metric vrs $\frac{UpdatesPerStep}{N_{benchmark}}$ . . . 19

4.6 Plot of $\frac{SystolicChannelWidth}{VPRChannelWidth}$ vrs $\frac{UpdatesPerStep}{N_{benchmark}}$ . . . 20

4.7 Plot of $\frac{SystolicChannelWidth}{VPRChannelWidth}$ vrs Runs . . . 21

4.8 Plot of Final Metric vrs Runs . . . 22

4.9 Plot of Final Metric vrs $\frac{parameter}{N_{benchmark}}$ . . . 23

List of Tables

3.1 Statistics of Implementation 11
3.2 Summary: Analysis of Hardware Implementation 12

Chapter 1

Introduction

1.1 Introduction

Reconfigurable computing is a hot topic in academia and research. A large and growing community of researchers have used field programmable gate arrays (FPGAs) to accelerate computing applications and have achieved performance gains of one or two orders of magnitude as opposed to general use processors[2]. The IEEE in April of 2005 organized its *13th* Symposium on Field Programmable Custom computing machines, yet reconfigurable computing has not made it onto the consumer market. Before this can happen a number of key drawbacks need to be overcome. One such drawback is the mapping of program logic to physical programmable resources anytime the machine reconfigures itself for a task. This thesis proposes to build on one possible solution to this drawback in the hopes of bringing consumer-reconfigurable computing machines one step closer to reality.

1.2 Background

A Field Programmable Gate Array (FPGA), is formally defined as an array of uncommitted logic blocks and interconnect resources[3]. This means that given an FPGA with enough logic blocks, one can program it to emulate any kind of logic circuit. For this reason, FPGAs are also used extensively in simulating application specific integrated circuits(ASICs) before they go into production. An example is XILINX's implementation of an internet protocol router that handles multiple gigabit ethernet networks running either IPv4 or IPv6[4].

The use of any reconfigurable machine involves implementing programming logic with gates and the subsequent mapping of the gates to the resources available on the board. Mapping the gates to the physical board is an NP-Hard placement problem. This means that the problem is not optimally solvable in polynomial time. The typical FPGA boards consist of 30,000 LUTS[1], with 100,000 LUT models available. Current

[1] Look-Up Tables - the basic unit of an FPGA

advances in nano-scale solid state devices, promises FPGAs with a million or more LUTS. This implies that, not only do we need optimal placement algorithms but optimal placement algorithms that will scale with the size of the input. We know that optimal placement is not possible so we will have to settle for really good placements that scale with the size of the input.

Placement algorithms constitute a whole field of computer science, with its own classification system. One of the most developed algorithms in use today is that of simulated annealing. Simulated annealing is an iterative improvement algorithm that tries to avoid getting stuck in local mimima by occasionally accepting moves that result in a cost increase[5].

In his master's thesis, Caltech graduate student, Michael Wrighton, using simulated annealing as a basis, presented a spatial approach to the placement problem for FPGA's that promises to scale with the growth of the number of gates that can be built on an FPGA[6].

This thesis takes Wrighton's work a step further by attempting to provide a modular hardware implementation that is capable of running at a frequency of $200Mhz$. In addition, methods of reducing stale data in the algorithm are explored.

Chapter 2

Problem Statement and Solution Sketch

2.1 Problem Statement

The placement problem can be stated as: "Given an arbitrary graph of logical nodes, construct a two-dimensional physical mapping that is ideal in some way"[6]. For an FPGA, this can be reworded to read: Given an arbitrary logic design and a set of programmable LUT resources, map the design to the two-dimensional array of resources such that some criterion is satisfied. In most cases the criterion will be area or speed. Wrighton showed in [1] that it is possible to use an FPGA to solve its own placement problem using simulated annealing as a basis. We present a modular implementation of his design in hardware.

2.2 Simulated Annealing Overview

Simulated annealing is a technique for solving optimization problems. It is based on the manner in which crystals form from liquids/gases[7]. At high temperatures the molecules of liquids/gases are in a high energy state and thus are very mobile. As the temperature decreases, the molecules move into lower energy states and become less mobile. This is the annealing process and it ends when the molecules attain their lowest energy state(ground state). Local ground states are avoided by lowering the temperature slowly and spending a long time in temperatures around the freezing point[5].

Generally, in solving the placement problem via simulated annealing, we simulate on a sequential processor the annealing process. Each element to be placed is modeled as a molecule in it's non-optimal energy state. The element tries to find its ground state(i.e position of least contribution to global cost) by swapping positions with neighbouring elements. This process often converges easily to a local extrema. To counter this, a simulated system adds some stochastic behavior by randomly causing elements to make moves that do

not necessarily optimize the criterion. As the system temperature drops such random movements decrease until elements settle into their final and possibly optimal positions.

2.3 Hardware Simulated Annealing

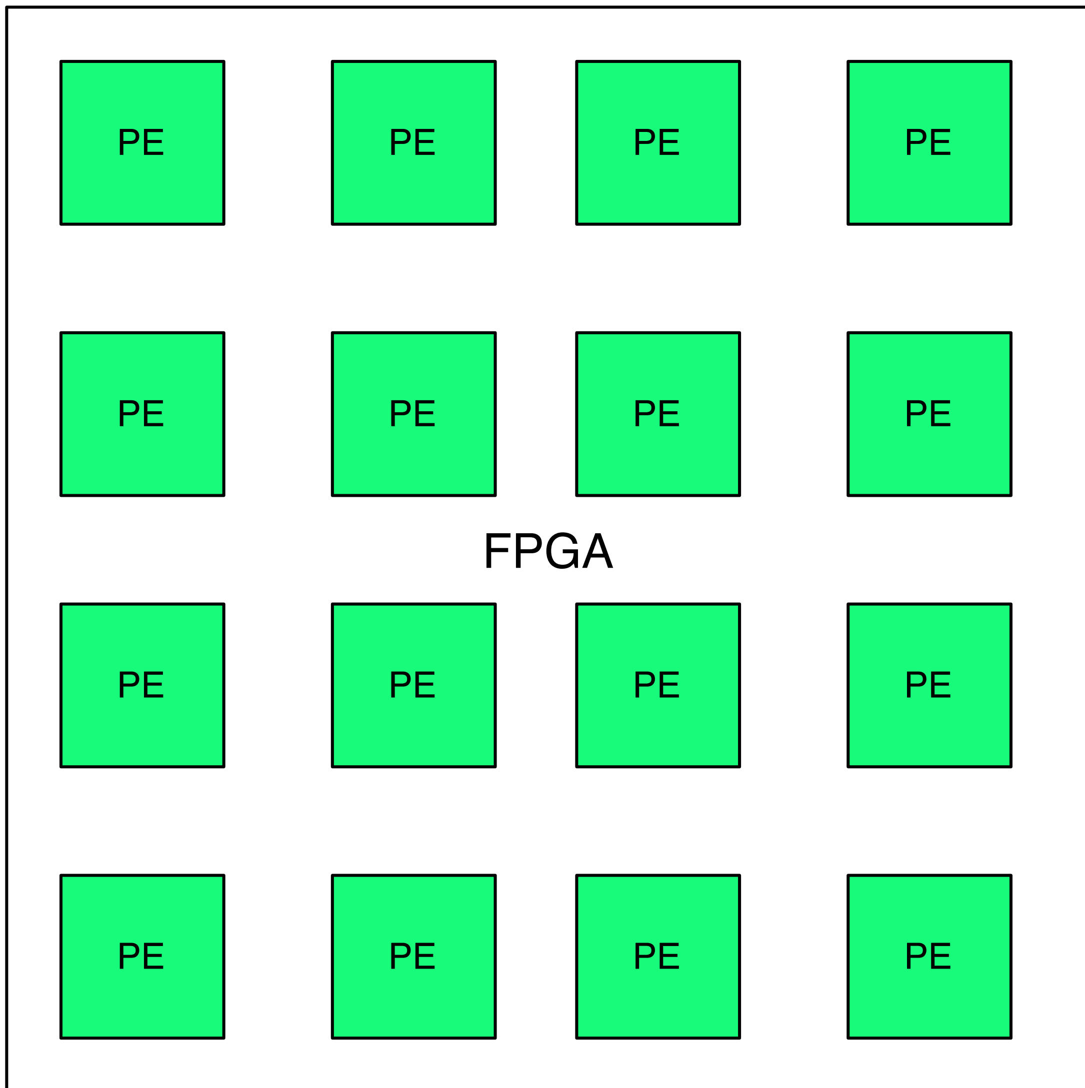
In hardware simulated annealing, the idea is to instantiate an array of processing elements(simulated annealing molecules) on an FPGA (Figure 2.1). The PEs work as follows: given a circuit for placement on an FPGA, assign a unique id to each logic block. Randomly assign each logic block in the circuit to a processing element. For each processing element consider swapping positions with each neighbour in turn. Swap regardless of change in optimization criterion if system temperature is higher than a randomly generated number; otherwise, swap based on optimization criterion[1].

In our implementation, the optimization criterion is the total manhattan distance between connected elements. Thus, we only swap if the new positions of the elements will result in a smaller total manhattan distance.

Swapping is achieved by swapping the unique id between the two blocks. In addition, the position data for connected-LUTs is also swapped.

Temperature is usually controlled by a cooling function. In our case we use a linear decreasing cooling function. The placement is complete when the temperature attains its minimum value.

Figure 2.1: Systolic Placer: Array of Processing Elements



Chapter 3

Solution Implementation

3.1 The Processing Element

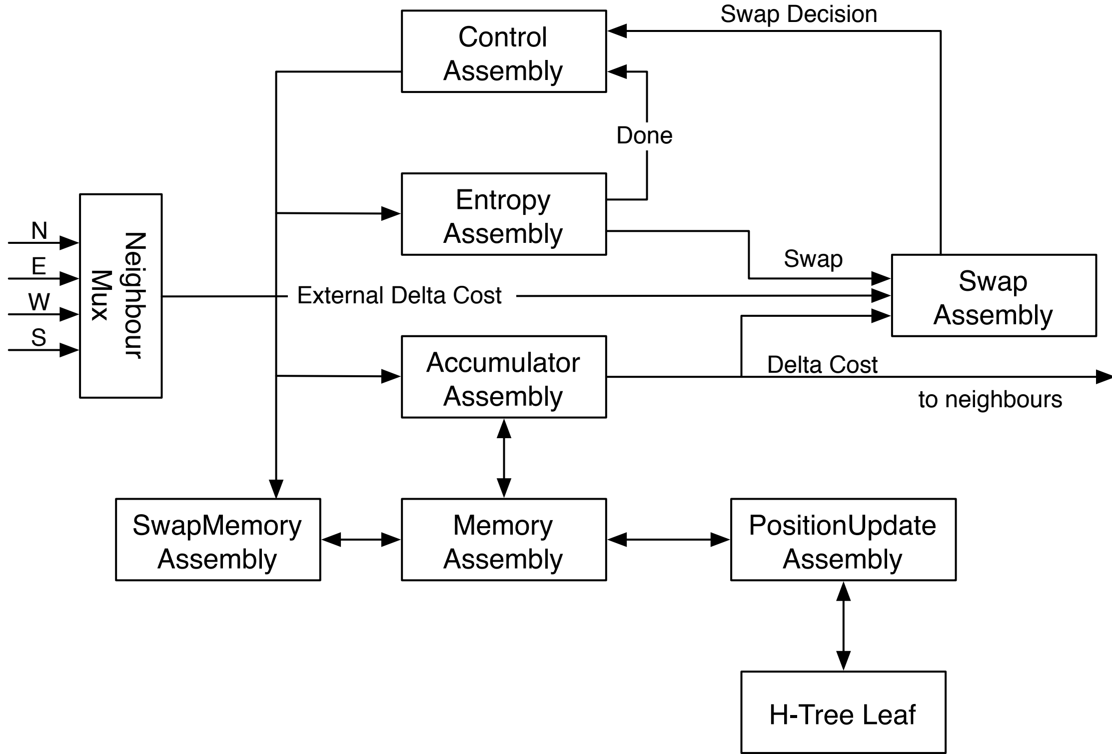
In order to achieve the goals of a modular design that is able to run at 200Mhz, the processing element is broken into 7 major blocks(Figure 3.1): the Entropy assembly, the Accumulator assembly, the Memory assembly, the PositionUpdate assembly, Swap assembly, SwapMemory assembly and finally the Control assembly. Each block is implemented with the 200Mhz goal in consideration. This enables us to determine bottle-necks in the design and thus direct more effort towards getting individual blocks to attain the overall goal. In addition, the modular design allows the algorithm implemented in each design to be swapped thus enabling experiments. We describe in more detail each sub-block in the subsections below.

3.1.1 Entropy Assembly

The entropy assembly is responsible for the randomness and cooling schedule of the simulated annealing process. It consists of two main elements. A cooling schedule and an LFSR[1] used as a random number generator. Its outputs are the done signal and the swap signal. The swap signal is asserted whenever the randomly generated number is less than the current temperature. The done signal is asserted when the temperature attains its minimum value.

[1] Linear-Feedback-Shift-Register

Figure 3.1: Systolic Placer: High level block diagram of PE



3.1.2 Accumulator Assembly

This block calculates the delta cost for the PE. Equation 3.4 defines delta cost.

$$L(E_i) = \text{current location of } E_i \tag{3.1}$$

$$C(E_i) = \text{position array of connected-LUTs of } E_i \tag{3.2}$$

$$M(P_i, P_j[]) = \text{Manhattan distance between position } P_i \text{ and positions } P_j[] \tag{3.3}$$

$$\text{DeltaCost} = M(L(E_2), C(E_1)) - M(L(E_1), C(E_1)) \tag{3.4}$$

The accumulator assembly breaks down further into the following blocks.

3.1.2.1 CurrentCost Accumulator

The CurrentCost Accumulator calculates the PE's contribution to the global cost/optimization criterion using information it has about the positions of it's connected-LUTs.

3.1.2.2 HypoCost Accumultor

This is a counterpart to the CurrentCost Accumulator. It calculates the cost assuming a swap is made with the current neighbour under consideration.

3.1.2.3 Diff Accumulator

The Diff Accumulator calculates the delta cost for the PE.

3.1.3 Swap Assembly

The swap assembly is responsible for making swap decisions. A swap is made if the entropy swap signal is high irrespective of delta costs. If the entropy swap signal is low then a swap is made if and only if the total delta cost increases the optimization criterion.

3.1.4 Memory Assembly

The memory assembly provides position data of connected-LUTs to the accumulator. The position data is used in calculating a PE's cost contribution to the optimization metric.

The data is kept in a RAM[2]. In addition, a content addressable memory(CAM) is used to quickly and efficiently determine if a PE is connected when a position update is received by the PE. A condensed copy of the CAM is kept in another RAM block to allow for easy swapping of the CAM data. This additional RAM is usually referred to as the shadow RAM. A more detailed description of the CAM and its design can be found in [1].

3.1.5 SwapMemory Assembly

When a swap is to be made, this block handles the swapping of the memory content. This is done by iterating through the addresses of the shadow RAM and the position RAM. The RAMs are dual-ported so that the algorithm can read out stored values and write incoming values at the same time.

3.1.6 PositionUpdate Assembly

The PositionUpdate routes position updates to the position RAM and also sends out position updates of its PE. In our implementation it also acts as a leaf on an H-Tree structure that connects all PEs. The H-tree is used to broadcast position updates to all PE's. The H-Tree is one of the key differences between this work and [1].

[2] Random Access Memory

3.1.7 Control Assembly

The control block is a finite state machine (fsm) that controls the actions of the PE except for the position update. Pseudo-code is given below to help give an overview of the scope of the control assembly. One run of the fsm equates to one cycle of a swap phase.

3.1.7.1 Pseudo-Code for State Machine

```
S0:
clock entropy
initiate AccumulatorAssembly
enable next neighbour in mux
GOTO WAIT

WAIT FOR SWAPDECISION DATA PRESENCE:
IF swapdecision=1
 initiate SwapMemoryAssembly
 GOTO SWAP
ELSE
 GOTO S0
ENDIF

SWAP:
IF swap=done
 GOTO S0
ENDIF
```

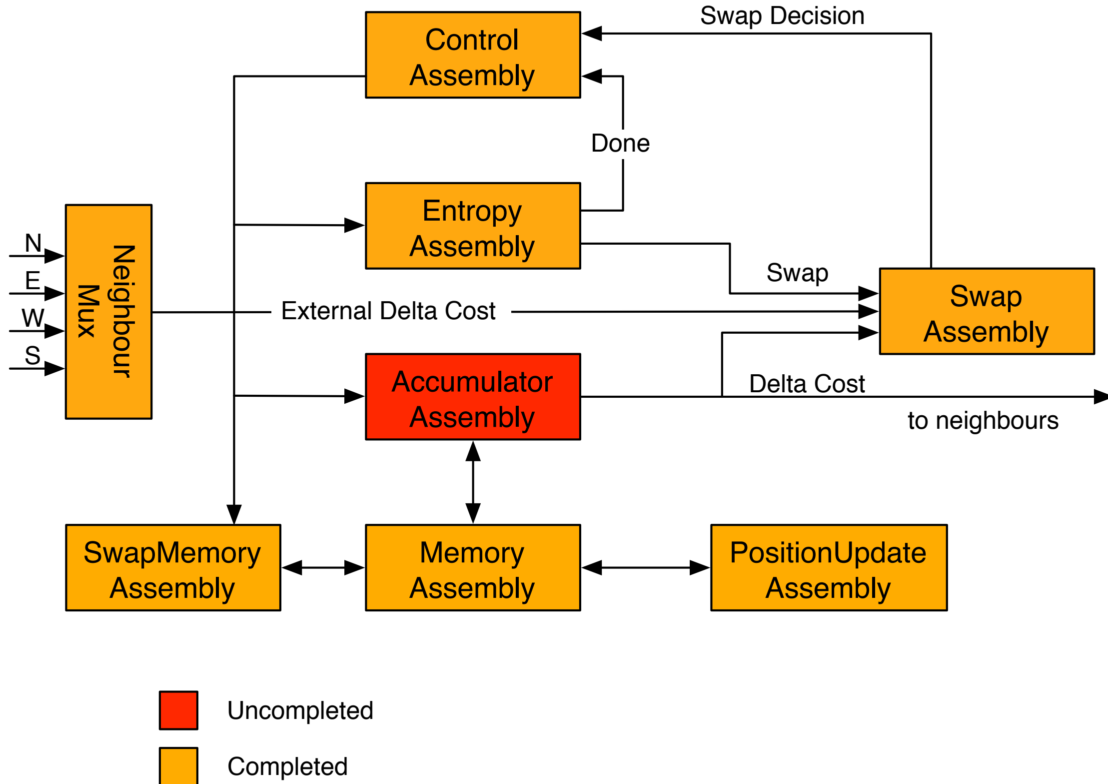
3.2 VHDL/Hardware Implementation

3.2.1 Approach

In order to achieve the goal of a 200Mhz placement engine, the plan was to implement the RTL level of the algorithm in VHDL and verify the correctness by simulation in ModelSim. The verified design could then be synthesized using Synplify Pro. This would give us an edif[3] file that could be sent through the Xilinx ISE software to map to actual FPGA resources. From this process we would obtain timing and area usage data. In order to communicate designs to be placed with the placement engine, software would be written

[3]Edif is a file format and it stands Electronic Design Interchange Format

Figure 3.2: Status Block diagram for PE



in an appropriate programming language to interface with the hardware. The software would be responsible for loading the netlist to be placed and also reading out the final placed netlist.

The RTL level implementation of the design was mostly successful. Figure 3.2 shows the current status of the blocks. The blocks were implemented in VHDL, synthesized for a Xilinx Virtex2(XC2V6000). Synthesis was done using Synplify Pro 7.7.1. Place and route was done using Xilinx ISE 6.3i. Table 3.1[4] shows the resource usage for the completed blocks.

We note that for the most part we meet our timing requirements. In the cases where we do not meet this requirement, we note that it might be possible to achieve the target with some human intervention. The only constraint used during the place and route stage was a 200Mhz speed requirement. Thus if we do floor-planning, implement better pipelining as well as enabling retiming during synthesis, it should be possible to attain a 200Mhz frequency for each of the blocks. This will subsequently improve the frequency

[4]Discrepancy between total and full PE synthesis can be attributed to optimizations by the synthesis tool.

Block	Synthesis(Mhz)	PPAR(Mhz)	Resource Usage(LUTS)
Control	445	513	10
Entropy	224	214	15
Swap	193	201	19
Memory	182	161	199
SwapMemory	483	586	3
PositionUpdate	497	256	14
MUX	-	-	24
Total	-	-	**284**
SystolicCell	**184**	**163**	**273**

Table 3.1: Statistics of Implementation

of the overall system.

3.2.2 Analysis of Hardware Implementation

A single cycle of computation in the placer cell begins with the ControlAssembly sending a signal to the accumulator to begin calculating delta costs. A signal is also sent to the EntropyAssembly in parallel to calculate the next random value.

The accumulator iterates through the list of connected-LUTs to calculate the delta cost. Number of iterations is equivalent to the total number of connected-LUTs. Our design has 12 connected-LUTs per PE. We expect the accumulator to take 4 cycles per calculation. This breaks down as follows

1. Register input
2. Calculate Manhattan wire length
3. Add to running total
4. Register the running total

The HypoCost Accumulator needs an extra cycle to calculate its Hypothetical position. The Diff Accumulator will take 2 clock cycles to calculate the delta. Thus the number of clock cycles needed to calculate the local delta is

$$12 + 4 + 2 + 1 = 19 \text{ cycles}$$

The local delta is sent to the SwapAssembly as well as the swapping neighbour. It is expected that it will take 1 cycles for the externalDelta to arrive at the SwapAssembly. Worst case time for SwapAssembly to send a swapDecision is 4 cycles. It takes the ControlAssembly a cycle to send a swap signal or restart the whole process. Thus a single cycle of processing without swaps will take

$$19 + 1 + 4 + 1 = 25 \text{ cycles}$$

Swaping involves iterating through the position RAM. There is a fixed latency of 4 cycles between the swap signal and the first swap value being available on the output. In addition, it takes another 18 cycles to program the cam for lookup. Thus a full swap takes

$$12 + 4 + 18 = 34 \text{ cycles}$$

.

Thus a full cycle for the design involves $34 + 25 = 59$ cycles. Table 3.2 shows a summary of the cycle count.

We also note that there is a minimum of 25 position updates per processing cycle. This number can be increased however by introducing stalls/delays into the process. We explore varying the number of updates via simulation in the next chapter.

Process	Num. of Cycles
Delta Cost	19
Ext. Delta Latency	1
Swap Decision	4
Swap Latency	1
Swap	34
Total	**59**

Table 3.2: Summary: Analysis of Hardware Implementation

Chapter 4

H-Tree

4.1 Motivation for H-Tree

In [1], the algorithm is presented as follows:

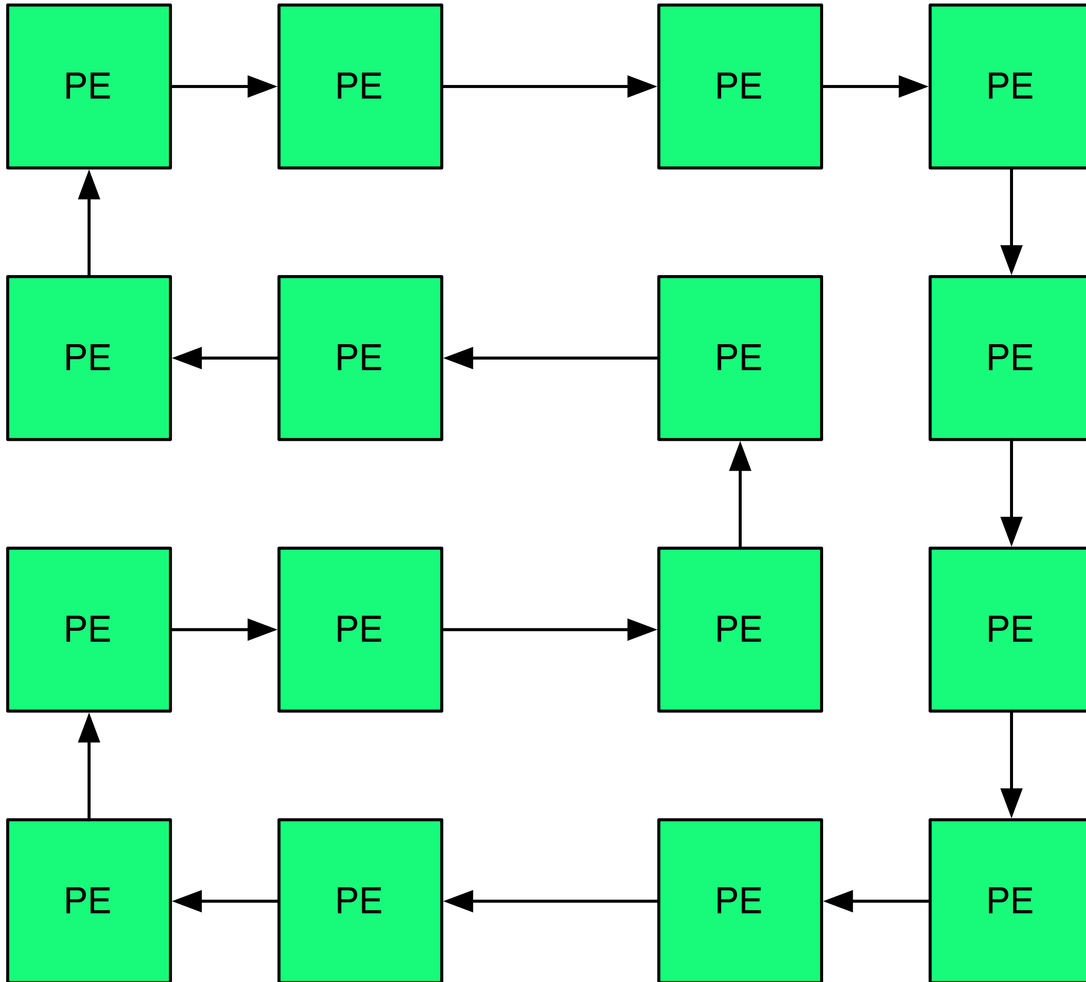
```
randomly place the design
WHILE NOT done
  FOR each node PE DO in parallel
    PE.shiftoutcurrentposition()
    LOOP numberofcells times DO
      PE.shiftpositionchain()
      PE.updateConnectedCells()
    LOOP swapsPerInterval times DO
      FOR four phases DO
        PE.swapIfAppropriate()
return the placement stored in the array
```

From the above, we make the observation that position data is updated every swapsPerInterval and that it takes $N \times ClocksPerUpdate$ to update. This update is achieved by cycling the position chain shown in Figure 4.1. Each cell puts its position data on the chain and its cycled N-times so that each cell sees every other cells position. This method has a couple of problems.

- $N \times ClocksPerUpdate$ is spent just updating position data
- $\frac{SwapsPerInterval}{N}$ of data is stale before an update. $\frac{iteration}{SwapsPerInterval}$ of swap decisions are made using stale data. Staleness becomes significant as $iteration$ approaches $SwapsPerInterval$.

We attempt to avoid these problems by using an H-Tree instead of the position chain method. The H-Tree allows us to broadcast an update to the whole array and thus every element receives the same position

Figure 4.1: Array of PE's with Position Chain



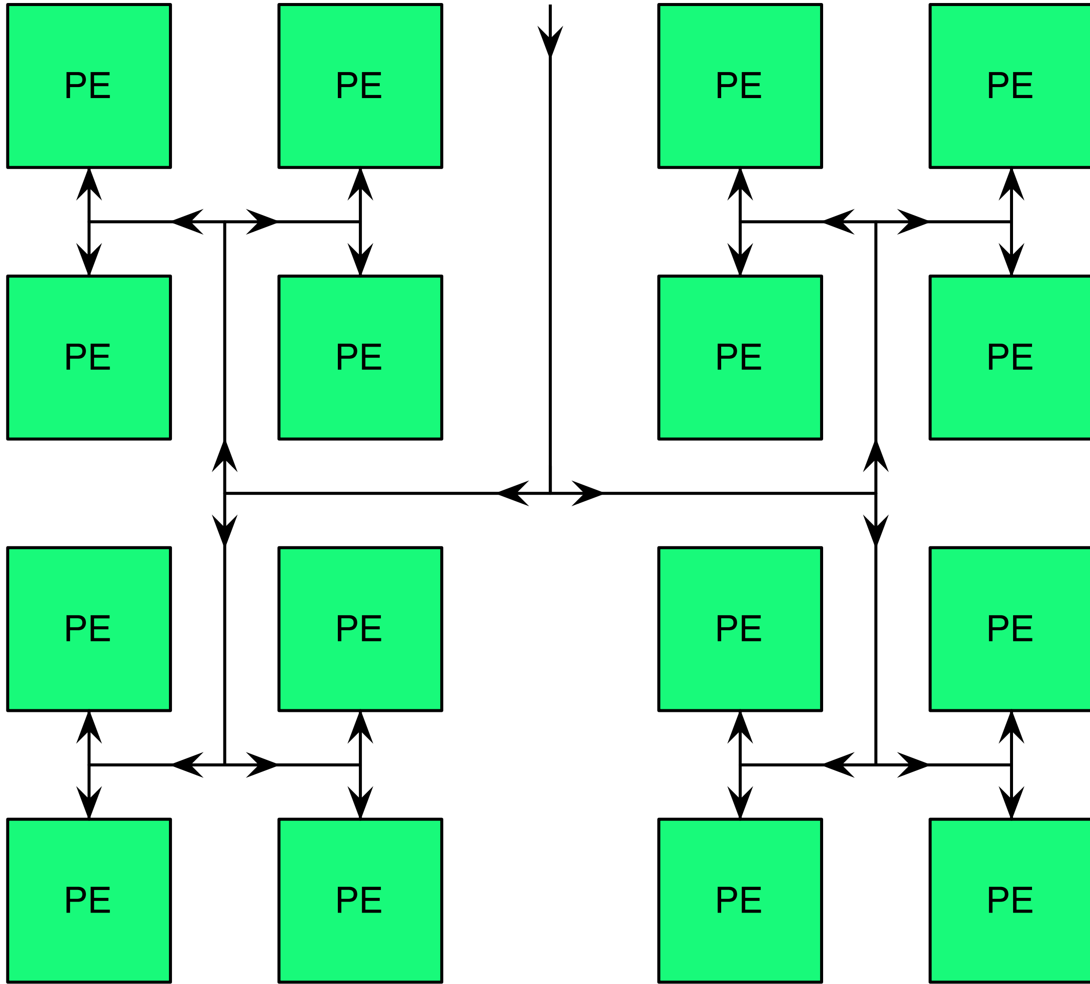
update $\lceil Log_2(N) \rceil$ cycles after the broadcast. This is easily pipelined to provide updates every cycle. In comparison to the Position Chain method, the H-Tree has lower latency in terms of time to propagate to all nodes. In addition, we reduce the staleness of the connected-LUTs position data.

4.2 H-Tree Characterization

The H-Tree structure is shown in Figure 4.2. This structure can be used in a number of different ways.

- The simple update - in a round-robin, each PE outputs its position on the tree.
- The sorted update - the PE who's position has changed the most gets priority on the tree

Figure 4.2: Array of PE's with H-Tree



- The random update - at each crosspoint of the h-tree, randomly pick one input to send up the tree.

Simple Update The simple update is parameterized in terms of the number of updates per step. The minimum value of this parameter is set by the number of clock cycles between swaps. This we determined to be 25 in the previous chapter. We explore the possibility of making this number larger using Manhattan distance as a metric. Approximately, $\frac{UpdatesPerStep}{N}$ of data is not stale when we make a swap decision. Thus increasing $UpdatesPerStep$ reduces staleness in the system. Intuitively, we expect that reducing staleness will produce relatively better placements.

Sorted Update The sorted update is parameterized by the number of sorted inputs that propagate down the tree. We note that allowing the whole set of sorted input to propagate will result in updating nodes with stale data. So after n sorted inputs are sent down the tree, we reset the tree and start filling the tree with new data.

Random update The random update is parameterized by the number of randomly selected inputs that propagate down the tree. We observe that allowing the whole set of randomly selected inputs to propagate will result in updating nodes with stale data. So after n randomly selected inputs are sent down the tree, we reset the tree and start filling the tree with new randomly selected data.

4.3 Simulation Results

We constructed a Java simulation of the Simple Update method and run it for varying $UpdatesPerStep$ using a subset of the toronto20 benchmark. Figure 4.3 and 4.4 show the results obtained in terms of the total Manhattan distance metric and channel width of the post-placed and routed design.

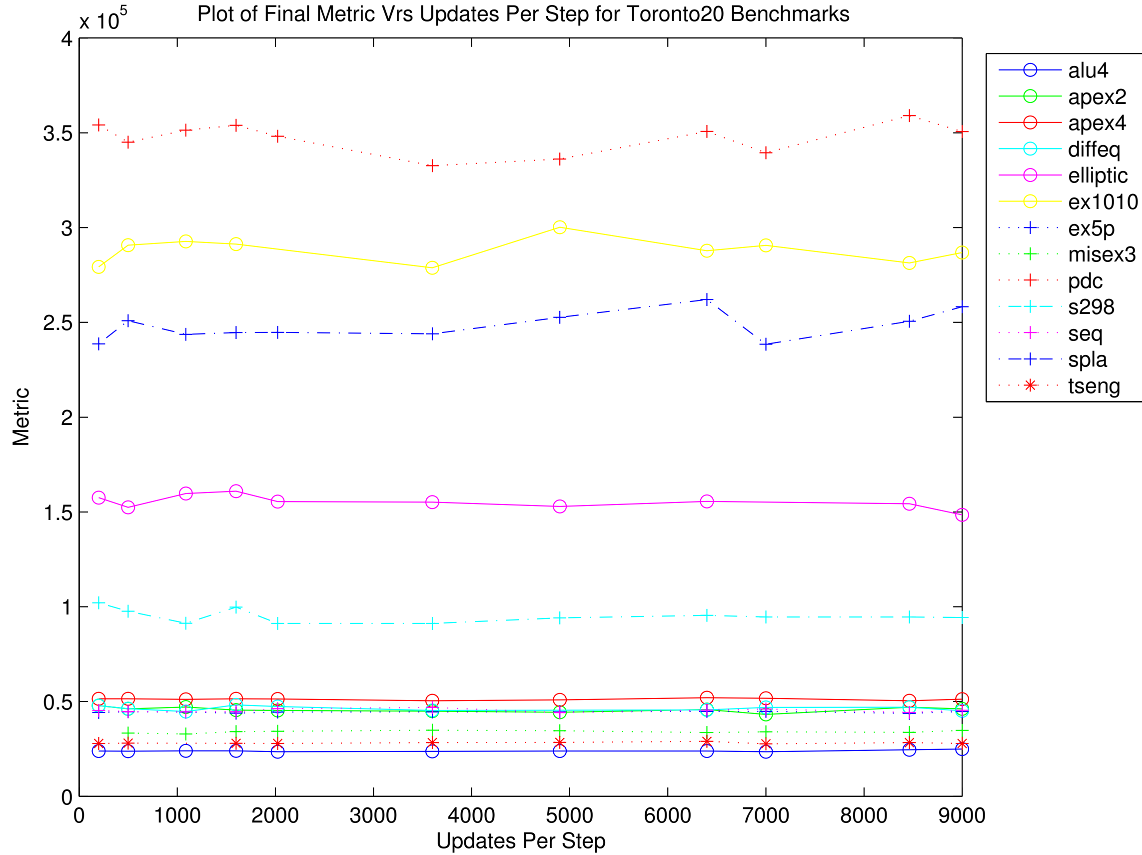
We note that by minimizing the Manhattan distance, we are localizing the route between connected logic elements. This corresponds directly to minimizing the channel width. The total area used by a design $= Area_{LogicElement} + Area_{interconnect}$. The area of the Logic Element(LUT) is constant and what actually varies is the number of wires needed to route between LUTs. Thus by minimizing the channel width, we minimize the number of wires needed to route and thus directly minimize the area of the design.

4.4 Metric Analysis

Figure 4.5 shows the metric plot for a representative subset of of the benchmark. The figures show that there is no clear improvement in performance as we increase the updates per step. In all cases our best performance does not correspond to the greatest number of updates per step. Similar statements can be made about Figure 4.6 where we look at relative channel width to judge performance. It is tempting to make the conclusion that stale data does not have a direct effect on our performance relative to VPR. This conclusion is however counter-intuitive so we explore the possibility that we are seeing run-to-run variations in our data or our simulation model is faulty.

Figure 4.7 and 4.8 show the data for 10 identical runs of our representative data set. We observe that these plots are very similar to the plots obtained by varying the updates. We discuss the implications of this data and the further work in the summary.

Figure 4.3: Plot of Final Metric vrs Updates per Step



One of the ways to verify the H-Tree model is to verify the inverse relationship between the $UpdatesPerStep$ and $SwapsPerInterval$. We did this by running experiments with identical values for the two parameters. Figure 4.9 shows the Final metric data for these runs. The data shows significant variation for low values of the parameters but not for high values. This suggests strongly that one of the models being used is not correct and needs to be identified, corrected and verified.

Figure 4.4: Plot of $\frac{SystolicChannelWidth}{VPRChannelWidth}$ vrs Updates per Step

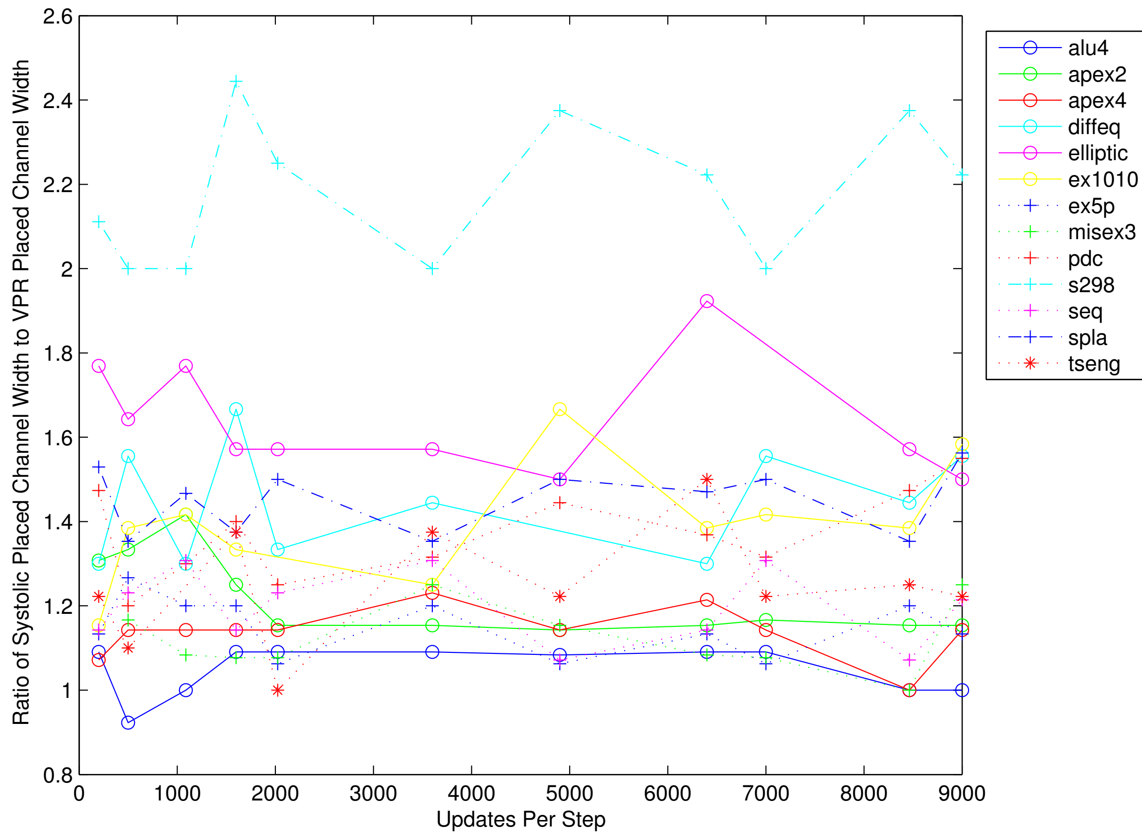


Figure 4.5: Plot of Final Metric vrs $\frac{UpdatesPerStep}{N_{benchmark}}$

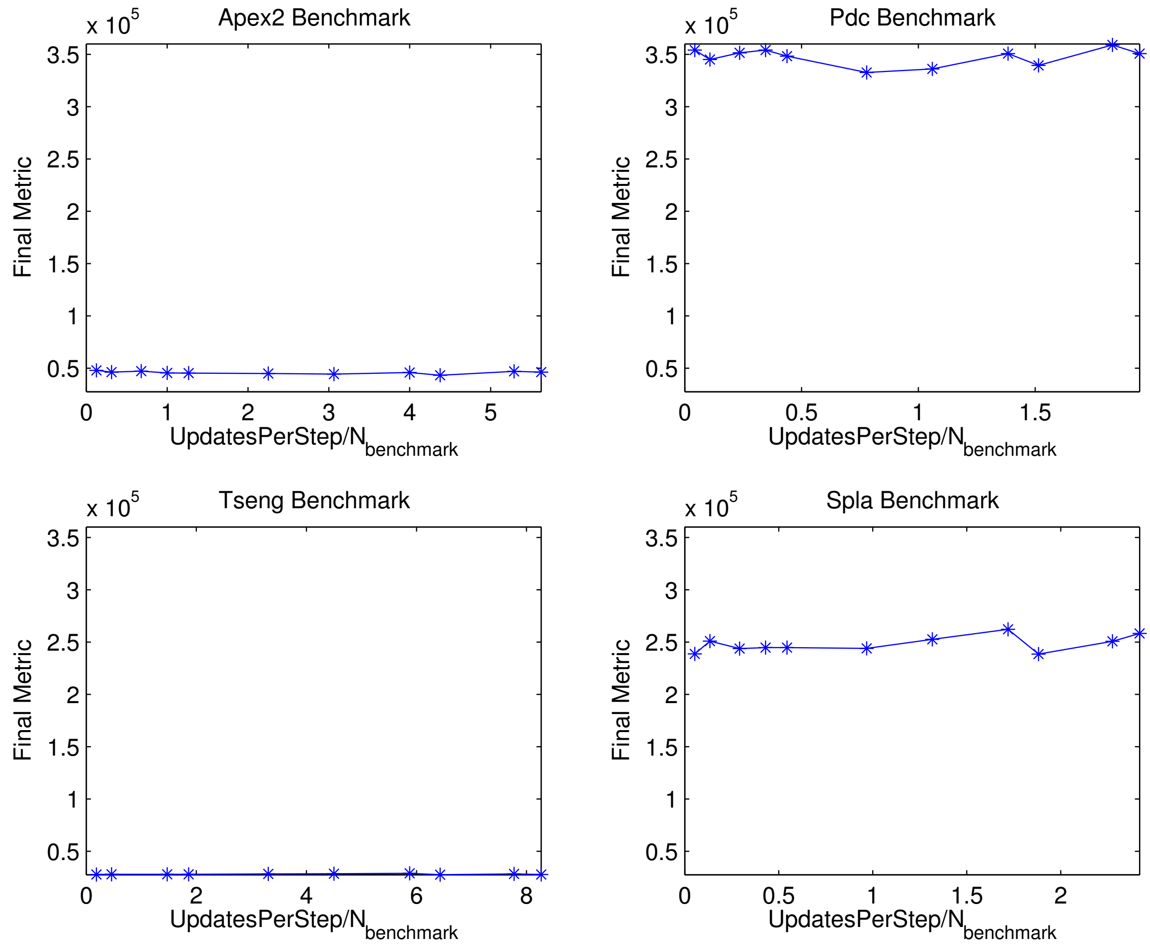


Figure 4.6: Plot of $\frac{SystolicChannelWidth}{VPRChannelWidth}$ vrs $\frac{UpdatesPerStep}{N_{benchmark}}$

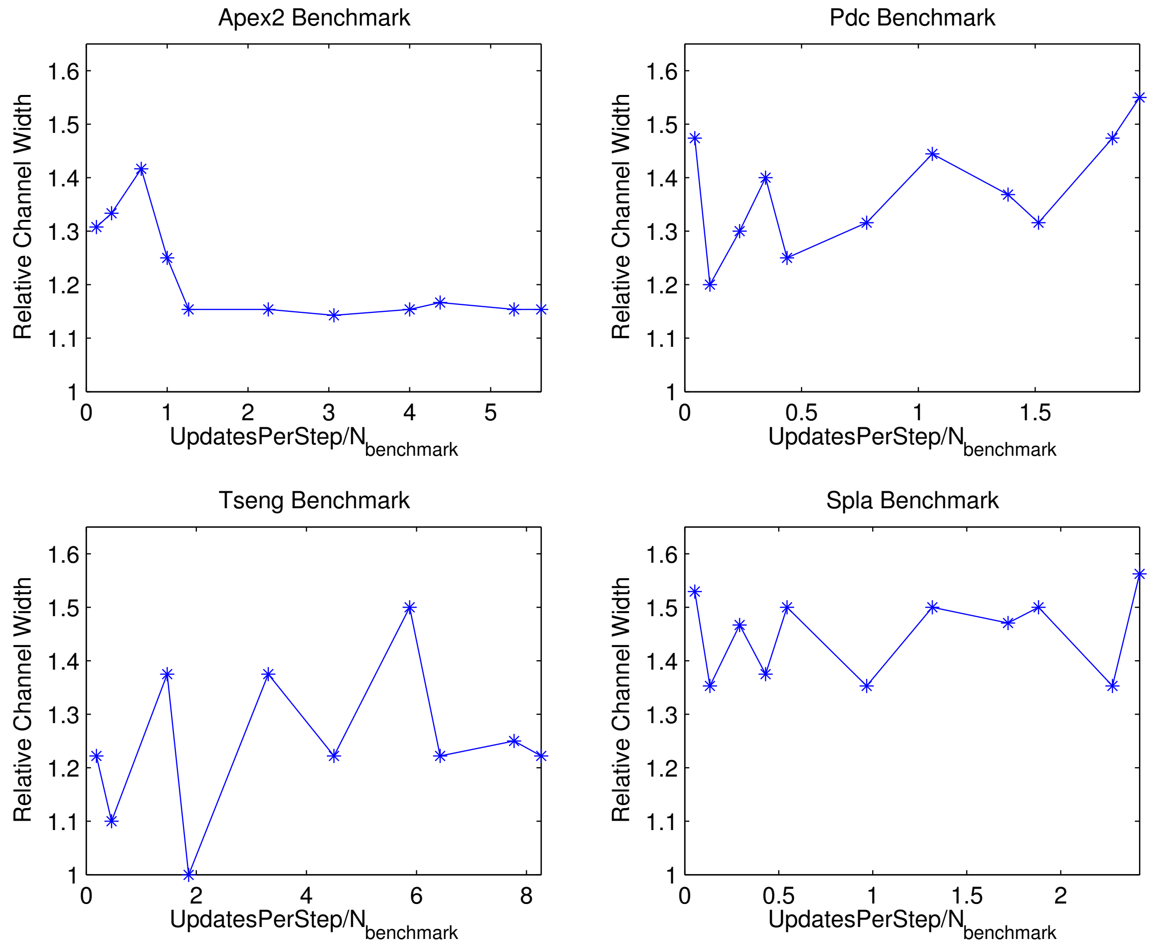


Figure 4.7: Plot of $\frac{SystolicChannelWidth}{VPRChannelWidth}$ vrs Runs

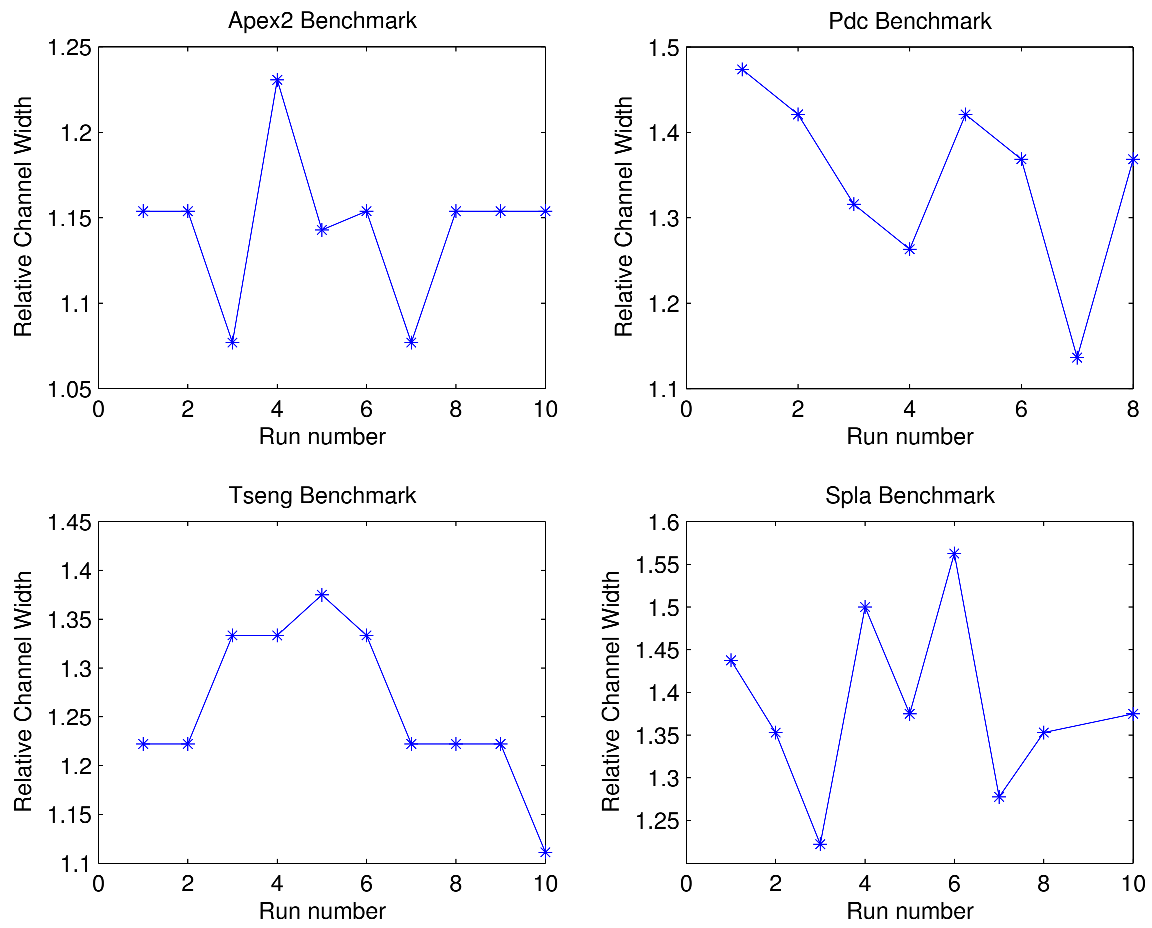


Figure 4.8: Plot of Final Metric vrs Runs

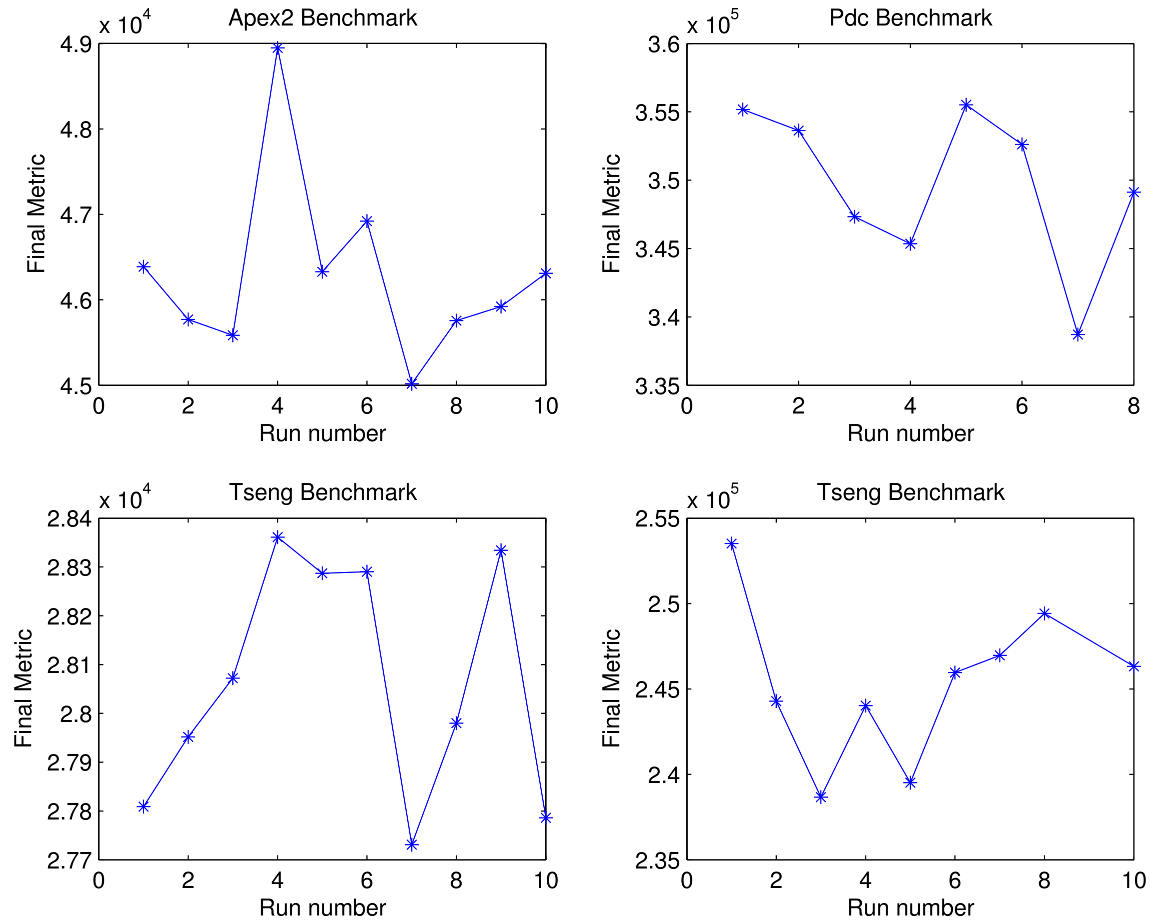
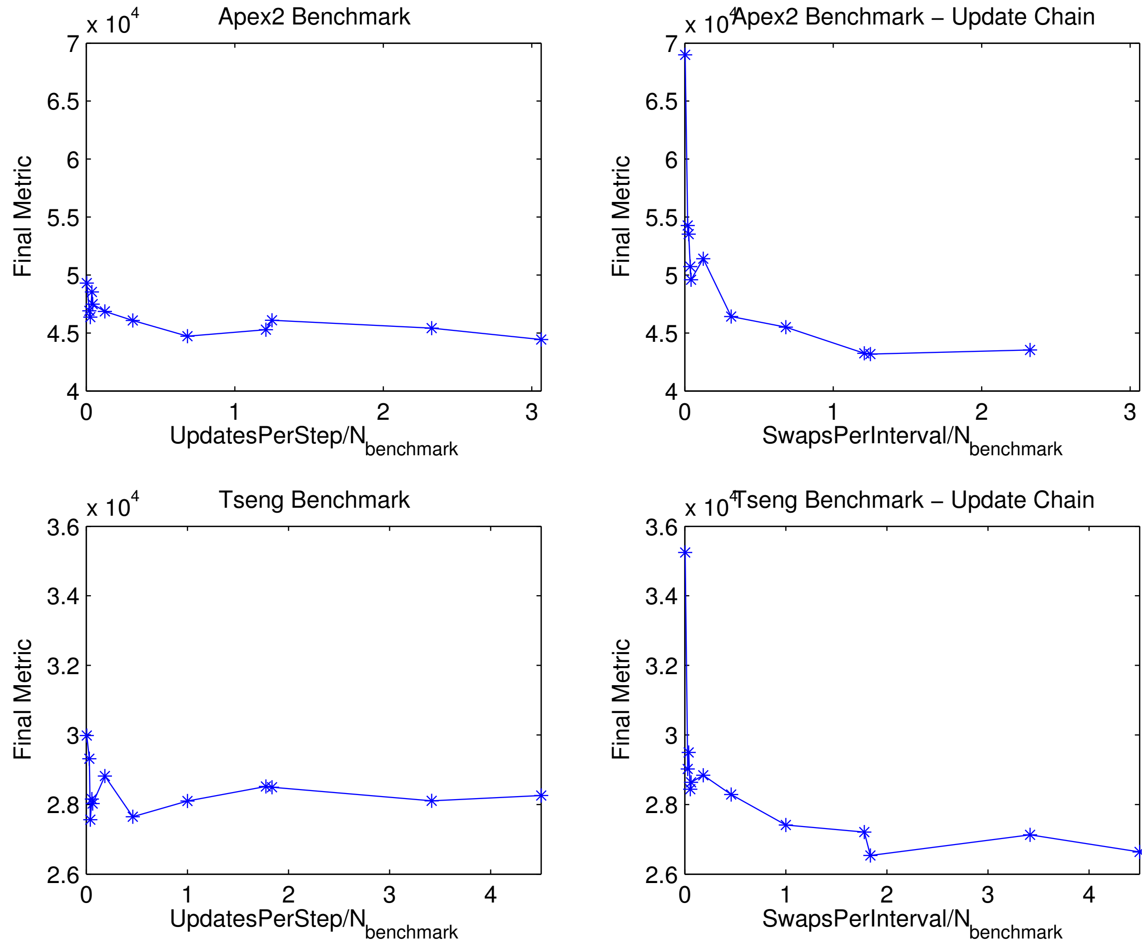


Figure 4.9: Plot of Final Metric vrs $\frac{parameter}{N_{benchmark}}$



Chapter 5

Summary and Future Work

5.1 Design Speed

Table 3.1 shows that most of our sub-blocks meet the design speed. Our bottleneck is in the Memory and more specifically the CAM element. The CAM is well pipelined and thus any gains to be made will have to be in floor-planing the placement of the MemoryAssembly.

We also note that we loose some performance in connecting the sub-blocks together. This drop in performance can be reduced by pipelining the inter-module connections.

5.2 H-Tree Simulation

The data from simulation suggests that the simulation model produces results that can best be described as noisy. There is a need to determine the source of the noise in the model and attempt to improve it. Our current guess is that we are cooling too fast. From our earlier explanation of the annealing process, we observe that at any given temperature, it takes time for the elements to find their lowest energy state for that temperature. In addition, the temperature corresponds to a range of movement that allows the element to attain this minimum energy state. Moving to a lower temperature limits the range of movement and thus puts the lowest energy state out of range for the rest of the process. Thus moving to a lower temperature too early in the process causes elements to end up frozen in a non-optimal positions and at best will end up in a local minima.

With this hypothesis in mind, the next step would be to either use a cooling function with a lower cooling rate and/or a cooling function that better models the way annealing is done. In addition, there is a need to verify the models being used in simulation.

Bibliography

[1] M. Wrighton and A. DeHon, "Hardware-Assisted Simulated Annealing with Application for Fast FPGA Placement," in *FPGA*, February 2003, pp. 33–42.

[2] A. DeHon, "The Density Advantage of Configurable Computing," *IEEE Computer*, vol. 33, no. 4, pp. 41–49, April 2000.

[3] S. Brown and J. Rose, "FPGA and CPLD architectures: a tutorial," *Design and Test of Computers, IEEE*, vol. 13, no. 2, pp. 42–57, 1996.

[4] *Mixed-Version IP Router (MIR)*, Xilinx, Inc, 2100 Logic Drive, San Jose, CA 95124, October 2003, xAPP 655 <http://www.xilinx.com/bvdocs/appnotes/xapp655.pdf>.

[5] S. Kirkpatrik, C. D. Gellatt, Jr., and M. P. Vecchi, "Optimization by Simulated Annealing," *Science*, vol. 220, no. 4598, pp. 671–680, May 1983.

[6] M. Wrighton, "A Spatial Approach to FPGA Cell Placement by Simulated Annealing," Master's thesis, California Institute of Technology, June 2003.

[7] M. W. Majid Sarrafzadeh and X. Yang, *Modern Placement Techniques.* Norwell, USA: kluwer Academic Publishers, 2003.