High Level Synthesis

Scott Walter

April 29, 2014

Contents

1 Executive Summary **2**

2 Defining the Hardware **3**
2.1 Rules 3
2.2 The Process 3

3 Defining the AI **4**
3.1 AI terms 4
3.2 Assumptions 5
3.3 Comparing Algorithms 6
3.3.1 A* 6
3.3.2 Depth First Search 7
3.3.3 Hill Climbing with Restart 8

4 Conclusion **8**

1 Executive Summary

When you examine the inside components of a computer you will always find a motherboard. On that motherboard you will see all sorts of intergrated circuits, wires, and other components. One might ask: "who placed all of these components on this board and who planned it out?" The placing of wire and component is done by machines, but the planning is carried out in a process called synthesis. Essentially, to create and design hardware one must first program a design in a hardware description language (HDL). Two of the most common types of HDLs include Verilog and VHDL. Once the programming portion is complete, i.e. you have written code that realizes the type of design you want to implement, the synthesis comes next. In this process a synthesizer will check the code's syntax and then begin organizing components. The way in which it does this is by following a set of user defined rules, often called constraints, and implementing a search algorithm. For example, say you want to create a one bit binary adder. The inputs to this system are the two bits you're adding, and a carry in. The outputs to this system are the sum of the two bits, and a carry out. What a search algorith will do in this problem is ingest your code which may look something like this:

```
output s, cout;
input a,b,cin;

assign s = a ^ b;
assign cout = a & b;
```

and produce some kind of real tangible hardware based on your code. Note, the purpose of the code is not to understand it, but just merely to see what the synthesizer is handed to pick logic. Now, what the synthesizer must do is select from a library of logical components, i.e. AND, OR, NOR, etc. and place them in a way on a logic board that realizes what the programmer wanted to do (in this case, implement a one bit binary adder), and additionally follow any rules that the programmer laid out which will be explained later. The purpose of this paper is to examine which particular algorithm is best at traversing the library of components available to our synthesizer to find the best design for our program.

2 Defining the Hardware

As aformentioned, one of synthesis' jobs is to literally place the components found in a design in a specific layout. It is the job of a sorting algorithm to take a component and place it somewhere in the schematic so it follows some user constraints, and produces the same logical output that is expected, and ensures the system is fast enough. This next section will better define what some of these rules are.

2.1 Rules

1. User constaints are typically found in a file that specifies what inputs and outputs map to the actual device you are programming. For instance, say we are programming our Verilog code to a device such as an field programmable gate array (FPGA). Going back to our single bit binary adder, we have three inputs and two outputs. Those three inputs and two ouputs need to be mapped to some physical spot on the FPGA such as a pin, or a push button. Additionally, the outputs also need to be defined in the same way. This is important for us because it will determine where certain wires and components must go.

2. When you synthesize something in verilog code you specificy a minimum timing allowed. This refers to the phrase: "My design meets timing." This relates to the critical path in the logic that is formed. Essentially, you specify that the longest logical path found in your schematic is only allowed to take a certain amount of time to resolve. This type of requirement might cause the algorithm to shift wires around skipping certain components, or use different components that have a faster speed but may take up more space as a possible trade off.

3. There is a space requirement as well. A programmer can specify how large a design may be. As aforementioned, shrinking the size of your schematic can have other effects on the circuit such as increasing the critical path's timing.

4. Additionally, a synthesizer will take your code and create the logical equivalent and then optimize it. It does this by implementing large Karnaugh maps to realize your equation, and then it optimizes it using a Quine–McCluskey methodology. For the purpose of this paper, it is not important that you understand these concepts.

2.2 The Process

Considering all of these rules there is a very specific process that a synthesizer may follow. In this paper I will be discussing the process outlined in Xilinx's Integrated Software Environment which is a very powerful integrated development environment, but the only portion we are concerned with is the synthesizer.

1. The HDL Parsing phase is first. In this phase, the synthesis tool checks your code for any syntactical errors

2. Next is the HDL Synthesis phase. In this phase we do multiple things. First we parse the code for any kind of inferred logic such as multiplexors, RAM, adders, subtractors, etc. This are items we already have predefined layouts for and just quickly implement them without having to sort through all of our logic. We then parse the code for any inferred finite state machines and create them in a style in accordance with a user defined setting. For instance, in Xilinx's ISE there is a compile time variable you can set that specifies how finite state machines are realized. This variable is "fsm-encoding" and an example of a possible option is "one-hot" which refers to the one hot coding scheme where each and every state is given a flip flop and only one flip flop is active at a time. Finally, in this phase user constraints are considered. These constraints effect where certain wires are laid and through which logic they propogate.

3. Finally, optimization occurs. This step is where the synthesizer takes the proposed schematic made in the last step and optimizes its' logic so that it not only meets timing and size requirements, but also uses optimized logic. Note, it is possible to have specifications that are impossible to meet. When this is done, an error is returned during synthesis. Also note, this particular phase is not important to the search algorithm we decide to use, but is include for competion's sake.

3 Defining the AI

In this section we will discuss which algorithms are best to use for the problem at hand. The problem being: "How do I best place hardware in a schematic so that it is optimized, follows the given rules, realizes what I had programmed, and produces the schematic in a timely matter." This problem is very similar to the einstein's puzzle in that you have several rules that you need to follow and no clear solution in the beginning. We will discuss the validity of three algorithms for this job. We will look at A*, Depth First Search, and finally Hill Climbing with Restart. Before discussing the merits of search algorithms, we have to define the problem in AI terms so that we can easily compare each search algorithm.

3.1 AI terms

1. Goal Evaluation: we will know we have reached our goal when we've placed all the necessary hardware to realize the logic we put forth in our code, and that hardware passes all user, timing, and size constraints, or returns an error stating that it is not possible.

2. Cost: Cost is a variable amount. As more items are placed, complexity increases, and difficulty of rewiring when components are moved increases. Therefore, as more items are placed, cost increases. Additionally, placing components with more pins (higher complexity) is more costly. This will be defined more clearly later on.

3. Determinism: The next state of the schematic is completely dependant on the current state and the change this state makes to it.

4. Dynamic: The environment is dynamic because it is constantly placing hardware in a schematic and removing it based on initial designs and optimization aftewards.

5. Observation: This problem is fully obeservable, i.e. you are able to see the complete state after each and every action made.

6. Time: This problem is discrete because there is only a finite amount of hardware configurations that can be made.

7. State Space Representation: The state space will be represented with a graph.

8. Dead Ends: Dead Ends do exist in this problem. For example, imagine we've placed 10/12 components and the last one we placed breaks a certain constraint and we had no other options with our last placed component. We will need to backtrack and find a different configuration that allows that component to be placed in a different way so that it does not break any constraints. This backtracking eliminates the possibility for a dead end and a search algorithm stalling. For search algorithms such as hill climbing algorithms, we will utilize "random restart" to avoid dead ends.

9. Solution Depth: The solution depth is the same as the maximum depth. Once we have placed all the components necesary to realiz the logic we have programmed, and it follows the rules, we are done. However, every single component needs to be placed for this to occur.

10. Relative Error Cost: Not all errors have the same cost in this problem. For instance, imagine an intergrated circuit with 32 pins on it and then an 8 pin device. If you incorrectly place the 32 pin chip, you will need to rewire all of the pins used by that chip once it has been reposititioned. In this case, it would take longer to reposition the 32 pin chip as opposed to the 8 pin chip. Thus, devices with a higher pin count are more costly.

11. Run Time Performance: The time it takes to come up with a solution is important. A user does not want to be waiting multiple days to produce a schematic for a design. The algorithm should be able to come up with a solution in a few minutes/hours depending on design size.

3.2 Assumptions

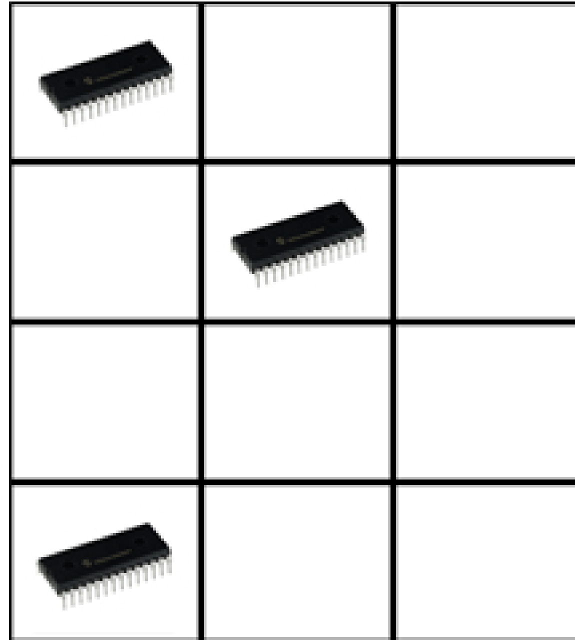
1. Component Amount: We will be placing 12 components on a schematic.

2. Schematic Layout: The schematic layout will be a 3x4 grid where each square is a slot that a single component can occupy.

3. Unique Slots: Each and every piece of hardware can be installed into every single slot, provided it follows the given constraints. There are no duplicate components in a design.

4. There are 12 components to realize your schematic. Each component is unique and can only be used once. This means that the depth of the tree is 12 and the branching factor starts out at 12 and decreases by one each time we go one level deeper in the tree.

5. To represent the increasing complexity of the hardware, as we place components and iterate through our graph at each depth, the amount of time it takes to place a component is increased by 10 ns. So, initally, the first component requires 10 ns, then the second requires 20 ns and so forth.
6. Here is an example of this grid:



3.3 Comparing Algorithms

In this sub section we will compare the three given algorithms for the problem at hand.

3.3.1 A*

A* is a heuristic assisted search algorithm. One of the main necessities of a heuristic assisted search algorithm is to know where the solution is in a given state space. The state space in this case is a graph. Unfortunately for A*, the goal node(s) is not known. This, however, does not completely count A* out of the race for planning out the schematic during synthesis. A* has two components that make up how it makes decisions, i.e. which node to expand next. It bases its' decisions off of a heurtistic cost, and an actual cost. In our case, the actual cost for each node increases as we get deeper in the tree, and there is no heuristic cost known because the goal node is not known.
Essentially, A* starts at a given state (no hardware placed) and has to do two things. It has to pick which component to place and where to place it. This is done simply by constructing a list of components, our aforementioned library

of logic, and ordering it by complexity. The synthesis tool wants to place the most complex item last. You might refer to this implementation as LCV or least constraining value. This is a constraint satisfaction feature, and is relevant to this search algorithm. The reason for ordering it this way was mentioned in section 2.2 under the "Relative Error Cost" definition. Simply put, you want to place the most complex items last because if your algorithm ever needs to back track before getting to the end (where your most complex items are) you will need to uproot less complex circuit components which will cause less "collateral damage" if there are lower pin count items as opposed to higher pin count items. After that list is sorted, you pick the first item (least complex), and place it in any slot and then you goal and rule evaluate after every component is placed. If a rule is broken, you revert the previously made change and expand the next node in your open list.
If there is no heuristic cost to measure, A* operates just like Uniform Cost Search. This search algorithm organizes which node to expand in its' open list based purely on actual cost.
This solution is complete and it has a worst and best case performance of $O((b-n)^{C/e})$[1] nodes searched where n refers to whatever level of the tree we are in, C refers to the optimal solution, and e refers to the cost of the first device placed, in our case, the lowest costing node. In this scenario, C* will equate to 780ns. This is the value of the time it takes to find the solution with no back tracking. The math to support this is in the DFS analysis. "e" then is equal to 10ns, meaning that every subsequent node placement costs less than 10 ns. With a variable branching factor, it is difficult to realize how much time we will spend at a best case and worst cast scenario searching with this particular algorithm. So, instead of usign the complexity as b-n, we will just simply use the average value that that equates to. So, if b-n can be equal to 12-1, 12-2, 12-3, etc., its' average value is 5.5. Additionally, it is difficult to account for the variable complexity of placing components, so we will just use the average value of 60 ns. So, the average compelixty, subsequently the best and worst case compelixty for A* equates to:

$(5.5^{7.8}) * 60 = 35.73s$

3.3.2 Depth First Search

Depth first search expands every node it visits and selects the first child node of that parent node and expands it continuing on until it hits a dead end. It then iterates back to the previous parent node it last visited that had another option and traverses down that branch in the same way. This algorithm will search all of nodes in the tree as a worst case scenario. It has a best case scenario of d. This is the case where DFS finds the solution on the first branch that it iterates through to the bottom. The best case scenario is equated by:

$10 + 20 + 30 + 40 + 50 + 60 + 70 + 80 + 90 + 100 + 110 + 120$ So, it has a best case search time of $780ns$, and a worst case search time of:

$\sum_{i=1}^{d}(12 * i) = (12 * 1)(10) + (12 * 2)(20) + (12 * 3)(30) + ...$

which equates to$43.68ms$. This solution is also complete. It will never return failure unless the user specified settings are impossible to meet.

3.3.3 Hill Climbing with Restart

This search algorithm is a form of hill climbing. In this problem, moving up the hill is placing a component that follows the rules. The "restart" metric refers to plateaus within the graph. When the algorithm has struck a local maxima, i.e. no move it makes moves it uphill and it has not yet reached the goal (all 12 components placed while following the rules) it will restart all over again. Essentially, the purpose of the algorithm is to store the "score" it got on each successive attempt and every time it scores "higher" than the last iteration, the new value replaces the high score. These intermediate values are of no value to us because we have either found our solution or have not at all in this application. We can modify this algorithm to never store intermediate values and stop only when all 12 components have been placed and follow the rules. To compare this to the other algorithms some additional assumption is required:

Sucess Rate	Average Steps to Win	Average Steps to Fail
20%	5	5

On every single trial there is a 20% success rate. The amount of average iterations it takes to fail and succeed is the same, 5. This means that this algorithm will need to on average run 5 times in order to guarantee 100% success. The worst case scenario is that it fails on the first four trials and succeeds on the fifth. The best case scenario is that is suceeds on the first trial. The complexity of hill climbing is $O(d * (b - n))$ [2] where d is the depth, b is the branching factor, and n is which level in the tree we are. In order to equate the best and worst case performance we need to understand how much time a failure takes and how much time a success takes. A failure, on average, requires 5 steps. This means that a failed trial equates to:

$10(12 * (12 - 1)) + 20(12 * (12 - 2)) + 30(12 * (12 - 3)) = 6.69ms$

A successful trial is simply this equation expanded out to the 12th value of n. If the worst case trial on average has 4 failures and then a success, then the time it requires to find a solution in the worst case trial is equal to:

$6.69 * 4 + 34.32 = 61.08ms$

The best case is simply:

$34.32ms$

4 Conclusion

When choosing the best algorithm for this particular job one must consider the average amount of time spent searching, and the completeness of the algorithm. In all three searches, our algorithms are complete given the assumptions are true. This means, that the only metric we care about is time spent searching on average. To review, based on assumption, this value takes into consideration the amount of nodes searched, and which nodes those were in the tree (later in the tree meaning higher complexity) and showcases the best case and worst case scenario of time spent searching these nodes.

Algorithm	Best Case Search Time	Worst Case Search Time
A*	35.73 s	35.73 s
DFS	780 ns	43.68 ms
Hill Climbing	34.32 ms	61.08 ms

Based on the the chart, A* is completely eliminated from being chosen as a possibility. However, the other two remaining algorithms are still competing. The next important feature to analyze would be how often DFS and Hill Climbing operates at its' best or worst case scenario, or in other words, its' average case search time. Since the Hill Climbing values already consider how frequently successes and failures occur, all we need to do is average our two values to find the average time spent searching which is 47.7 ms. We will assume the same for DFS. The reason we do this is because DFS can find a solution in the first branch it iterates through and the last, and every single branch in between. Averaging the two values at 50% accounts for all of the positions where DFS might find a solution. This average case value is 22.23 ms which for our algorithms, is the best search time meaning that DFS should be chosen for the task of organizing hardware during synthesis.

References

[1] "Uniform-cost Search." *Wikipedia.* Wikimedia Foundation, 04 Feb. 2014. Web. 28 Apr. 2014.

[2] "Algorithms & Complexity" *Nicolas Stroppa* Dublin City University. 2006-2007, 21 Nov. 2006. Web. 28 Apr. 2014.