Hephaestus: Solving the Heterogeneous, Highly Constrained Analog Placement Problem

Andrew Price

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering

University of Washington

2012

Committee:

Scott Hauck

Joseph P. Skudlarek

Ken Eguro

Program Authorized to Offer Degree:

Department of Electrical Engineering

Contents

Acknowledgements:

Scott Hauck—Thesis and Academic Adviser

Scott has been incredibly supportive throughout my time at the University of Washington. He has helped with several difficult decisions and has always been available for advice or friendly banter. In the classroom, it is obvious Scott has an enthusiasm for CAD, FPGAs, and algorithms. His interest in the subject shows and I can without a doubt say it has been a driving factor in my interest in these studies. Thank you for everything you've done and thank you for putting up with me over the past couple years.

Joseph P. Skudlarek—Industry Liaison to Cypress Semiconductor

Beyond simply being my go-between with Cypress, Joseph has proved time again to be both insightful and understanding, having helped guide the direction of this research. It has been a joy to work with Joseph both on this and other ventures. The more I've worked with Joseph, the more I've come to appreciate his attention to details and belief in clarity, simplicity, and elegance.

Ken Eguro—Affiliate Faculty and Member of Thesis Committee

Ken has provided insightful feedback at several points over the course of this research. In particular he was kind enough to discuss the implications and motivation behind the approach after hearing a talk on this presented at the University of Washington's Cascadia conference. Thank you for the discussions we had and thank you for being a part of this.

Rex and Kathy Price and Family and Friends

I, of course, want to thank my family and friends: my parents, especially, who have supported and encouraged me to always push myself and grow. If it weren't for their support I can honestly say I don't know where I would be today.

Angela

And I'd like to thank and acknowledge my very significant other who has always been supportive, understanding, and caring. Thank you for always being there to pick me up when I can't quite get there on my own.

Introduction to the Problem:

This research was undertaken in conjunction with Cypress Semiconductor. Cypress has a novel Programmable System on Chip (PSoC) family of devices which are composed of a digital subsystem, an analog subsystem, and a microprocessor. The integration of digital and analog domains make the devices highly flexible, capable of filling many different needs. The analog subsystem consists of programmable and fixed function analog components and a routing interconnect.

As will be further discussed, the architecture of the PSoC devices makes traditional estimates of routability non-ideal. Therefore, this research focuses on improving the placement of analog functional units by developing alternative methods of estimating the routability of a placement. This report documents the approaches that were considered, the pitfalls associated with some of those approaches, and presents an alternative metric with implementation details and results from running a set of designs through the new algorithm.

This report details several approaches that were taken to formulate a better metric to gauge routability of highly constrained heterogeneous architectures. Those methods that were not pursued are discussed, and the reasons for abandoning them are detailed. Finally, the proposed solution is presented, the implementation details are disclosed, and results are shared from several test cases placed using the new algorithm.

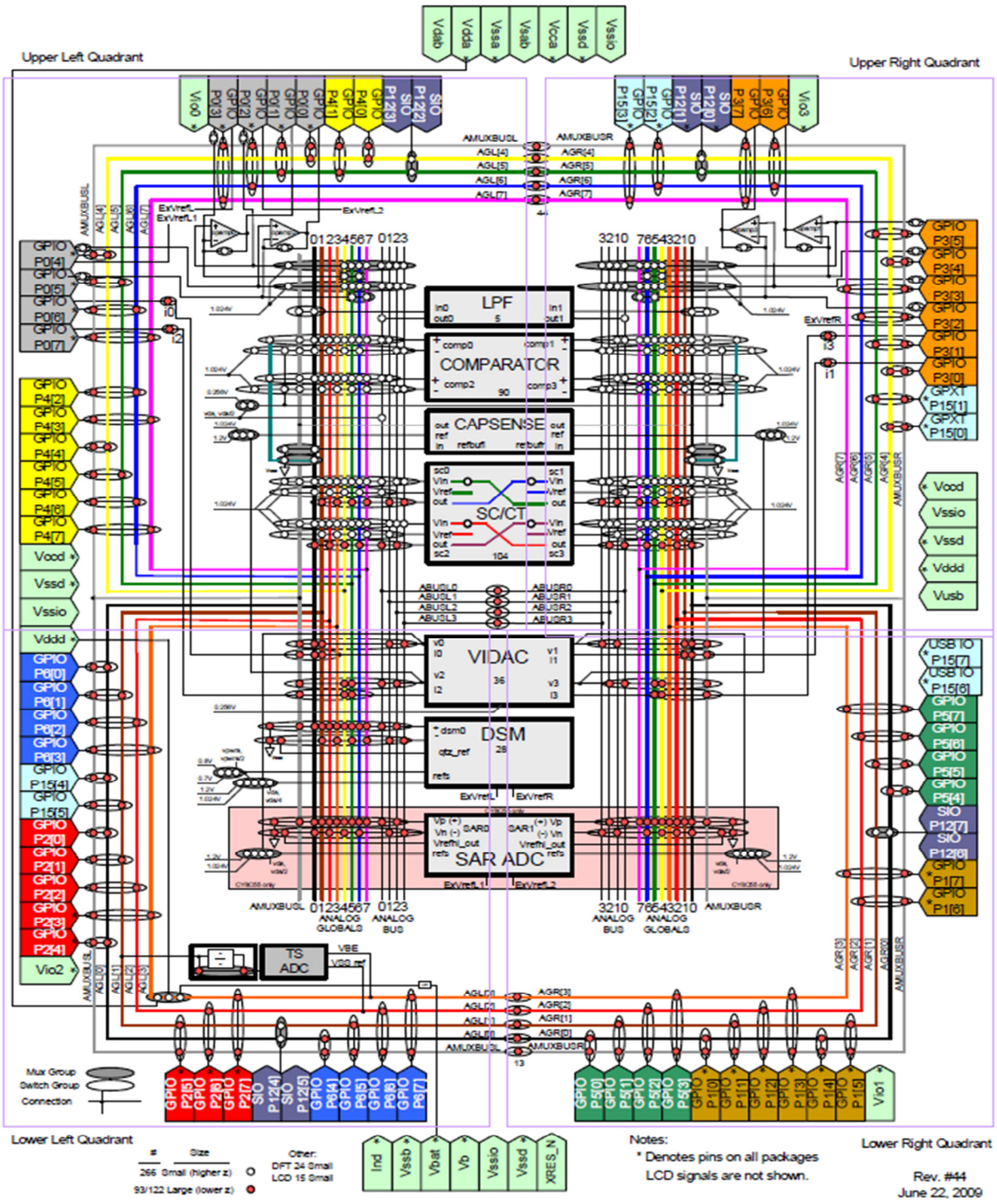


Figure 1. The analog routing graph of PSoC3/5 (1)

Architecture Glossary:

Through the course of the research project, several architectural elements and mechanisms have been given names to help communicate nuanced details.

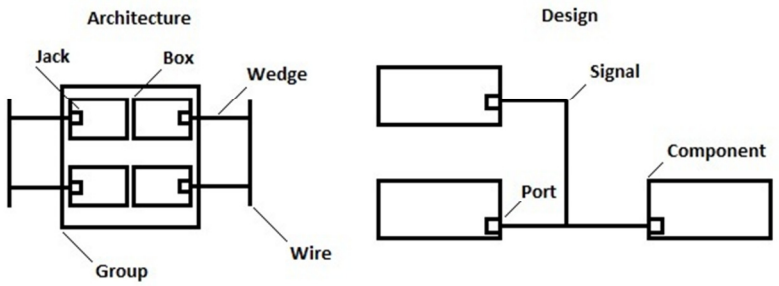


Figure 2. Constituent parts of an architecture and design.

- Box: Physical functional unit (available in silicon)
- Component: Logical functional unit
- Wire: Routing resource connected to other wires by switches
- Jack: Interface between a Box and a wire
- Wej (Pronounced Wedge, Acronym for '**W**ire **e**xtending from **Jack**')
- Neighbor: Any element adjacent to another in the architecture hypergraph
- Foreign (as in 'foreign signals'): For any signal, any other signal
- Ownership: Signals own wires. A wire may either be unowned or owned by one and only one signal.

The Wej element requires additional explanation. A Wej is a special-case wire. Each Jack of each Box has one and only one wire adjacent to it: the Wej. Later during route discovery, the Wejs, not the Jacks, are the elements the tool routes between. If a Box is occupied by a component (i.e. the placer has placed a component into that Box) then the Wejs of each Jack of that Box must be owned appropriately. If a Box is unoccupied, the Wejs of each Jack of that Box are free to be owned by any arbitrary signal for the purposes of track jumping (i.e. connecting two separate routing resources by a third common resource).

Architecture Details:

The PsoC 5 gives user access to the following functional units: 1 high-resolution delta sigma modulator (DSM), 4 8-bit digital to analog converters with voltage or current output (VIDAC), 4 comparators, 4 configurable switched capacitor/continuous time blocks (SCCT), 4 operational amplifiers (ABUF), 2 successive approximation analog-to-digital converters (SAR-ADC), 4 pairs of serial input/outputs (SIO) (while there are 8 available, they move together as pairs), and 60 general purpose input/outputs (GPIO). Two CapSense buffers (CSABUF) are also available but follow special placement rules. Unlike FPGAs which generally consist of generic programmable logic blocks exhibiting similar electrical and behavioral characteristics, these components differ in functionality and number of Jacks.

The platform also supports bundling routing resources to form muxes. If multiple components in a design require the Delta Sigma Modulator (only one exists) the inputs to the DSM can become a shared common resource with separate resources connecting components to the shared input through switches. For the purposes of this research, muxes were fused into static nets to eliminate the complexity of mux synthesis.

The architecture is divided into two connected halves, and each half is further halved, resulting in quarters. A total of 16 global wires, 8 local wires, and 2 global buses are the primary routing resources available (See Figure 1). The global wires, local wires, and global buses are divided evenly between halves. Those on the left half can connect to those on the right through a switch. Global wires are further divided equally between the two quarters. Global wires connect to half of all component Wejs and a quarter of all GPIO Wejs (respective of which half and quarter the global wire lays). Local wires only connect to some subset of all component's Wejs on their respective half. The global bus touches every Wej on its respective half. Keep in mind, because the global bus is available at each Wej on its respective side, it moreso than any other wire is attractive for its ability to get anywhere easily.

As shown in Figure 3, connections available to a Box differ based upon location. Figure 3 depicts the differences in routing resources available to the Jacks of the SCCT group (other groups exhibit similar characteristics). All locations are not equal with regard to available connections:

1. The Vin Jack of all SCCT Boxes on the left-half is adjacent to all left-half global wires. Similarly, the Vin Jack of all SCCT Boxes on the right-half is adjacent to all right-half global wires.
2. The Vref Jack of SCCT Boxes SC0 and SC1 are adjacent to the even enumerated wires on their respective sides (SC0 on the left-half and SC1 on the right-half). Conversely, the Vref Jack of the remaining Boxes SC2 and SC3 access the odd enumerated global wires.
3. The Vin Jack of boxes SC0 and SC1 connect to local wires #0, #2, and #3 whereas the Vin Jack of SC2 and SC3 connect to local wires #1, #2, and #3.
4. The Vref Jack of all Boxes touch one local wire (either local wire #1 or #2 depending upon location)
5. If the Vref Jack of a Box touches the even enumerated wires, the Vout Jack of that same Box touches the odd enumerated wires.

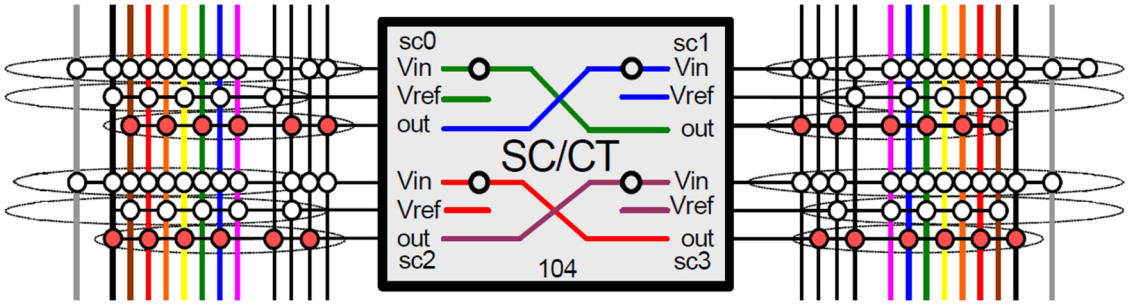


Figure 3. Close-up view of the SCCT group.

Several special wires, available within the architecture, provide direct and dedicated connections between specific Boxes. Several such examples include:

1. Current output (Iout) Jack of VIDACs to GPIO through a single switch.
2. Voltage output (Vout) Jack of op-amps directly to GPIO (See Fig. 4).
3. Feedback of voltage output (Vout) to negative terminal (V-) of op-amps through a single switch (See Fig. 4).
4. Negative terminal (V-) of op-amps to GPIO through a single switch (See Fig. 4).
5. Voltage output (Vout) Jack of VIDACs to voltage reference terminal of CSABUFs/ADCs through a switch.
6. Direct external voltage reference Jacks on DSMs/SARs to GPIOs.

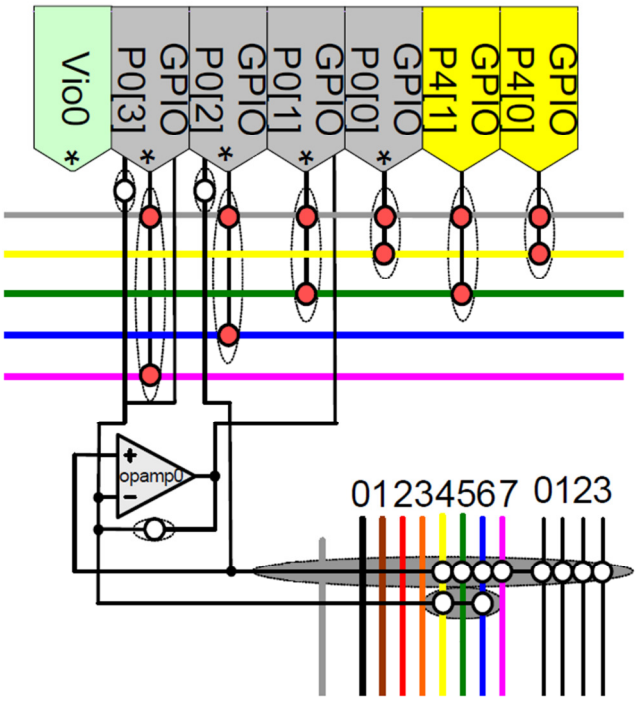


Figure 4. Close view of one op-amp. Note the dedicated connections between the op-amp's input Jacks and the GPIOs (Jack V+ to P0[2] and Jack V- to P0[3] both through a switch). Also of importance is the direct connection between the op-amp's Vout and GPIO P0[1] and the feedback switch between its V- and Vout Jacks.

Motivation:

The intricacies of the analog sub-domain of PSoC merit a closer look into alternative methods of performing placement. Traditional methods such as the Manhattan distance (semi-perimeter) routability estimate used by the de-facto FPGA place and route tool Versatile Place and Route (VPR) (2) do not correlate well to heterogeneous and constrained architectures. Semi-perimeter metrics make several fundamental assumptions about the architecture:

1. The routing interconnect of the architecture is uniform: Manhattan distance is commonly used as a metric for gauging the goodness of a placement for traditional island style FPGAs. These FPGAs exhibit uniform interconnects such that all potential placements for a logic block have access to equivalent routing resources. Within the analog sub-domain of the PSoC every potential placement is not equal. Earlier, while discussing the specifics of the architecture, it was noted that the routing resources available to Jacks between Boxes with similar functionality vary.

2. The placement positions are uniformly distributed across the architecture: In island-style FPGAs, logic blocks are distributed uniformly across the device. Moving a block from one location to another directly affects the semi-perimeter. The components of the PSoC architecture are geographically confined. Moving one component from a location to another does very little in terms of changing the semi-perimeter. Because changing component location does not dramatically affect this metric, relying on component location as an estimate of proximity and subsequently routability is impractical.

3. Closer together implies better chance of routability: the base assumption of close components being easier to route between is not necessarily true given this architecture. In fact, a densely packed quarter of the chip might very well pose more problems then had the components been spread evenly between all quarters. The limited number of routing resources per half and quarter implies that dense areas might not be capable of supporting a large number of signals.

While integrated place and route tools such as Independence (3) provide data-driven architecture adaptive solutions to the placement problem, the tool is overkill for the magnitude of our problem (i.e. the architecture should be place and routable with a simpler tool). This research introduces a new placement tool based on these ideas.

Alternative Methods:

Complete Enumeration:

The relatively small size of the architecture originally made complete enumeration of the problem space a seemingly reasonable approach to solving the placement problem. In this formulation both components and signals become placeable objects. All permutations of placement are found, and after checking the routing, invalid solutions are thrown out.

The densest design utilizing all components of the device would require the following number of permutations be examined:

Component	Count	Permutations	Combined
Operational Amplifier (ABUF)	4	4! = 24	24
Comparators	4	4! = 24	576
Switched Capacitor (SCCT)	4	4! = 24	13,824
Digital-to-Analog (VIDAC)	4	4! = 24	331,776
Successive Approximation ADCs (SAR-ADC)	2	2! = 2	663,552
CapSense Controllers (CSABUF)	2	2! = 2	1,327,104
Low Pass Filters (LPF)	2	2! = 2	2,654,208
Serial I/O Pairs (SIO)	4	4! = 24	63,700,992

Table 1. Total permutations per component group and the cumulative number of global permutations for all component groups

While, by today's standards, this many configurations is relatively small compared to complete enumeration of larger architectures, this number only takes into account component permutations. The possible permutations of the 60 GPIOs alone increase the previous count by a factor of 60! = 8.32E81. The argument could be made that very rarely

would all GPIOs be utilized by a design. One test case used while testing Hephaestus includes 25 GPIOs. Under the complete enumeration formulation described here, those 25 GPIOs increase the count of component permutations by a factor:

$$\frac{60!}{(60-25)!} \cong 8.05e41 \quad \textbf{(1)}$$

A second argument could posit that GPIO locations should be fixed by the application developer to prevent this added complexity. However, all possible permutations of signal assignments to routing resources cause an untenable mess. Rather than compute a naïve estimate of the possible permutations, a couple of simplifying assumptions are made for the sake of exposition:

1. Any signal can own the global bus (left/right). However, signals owning the global bus cannot own any global wires on the same half.
2. A signal can own one and only one global wire within a quadrant.
3. Any signal (regardless of global wire/bus ownership) may own one and only one local wire per half.

These assumptions are overly restrictive as they prevent any track jumping at the Wejs of GPIOs (i.e. under the assumptions, a signal can only own the global bus or one global wire per quadrant so it is impossible jump between two owned resources). This yields a lower estimate of the total possible permutations as this method of track jumping does occur in practice. The assumptions still allow track jumping at the Wejs of components so long as the jumping occurs either between two global wires in different quadrants or between a global wire and a local wire. Let us also assume that the GPIOs have been fixed in place and are not elements to be moved during placement. Ignoring GPIOs there are a total of 22 placeable components with a combined 48 Wejs between them. Thus, there are approximately 2.2 Wejs per component. Thus, for each component in the design, there are approximately 2.2 fewer Wejs to use for track jumping. These rules yield the following equations:

Global Bus	$X = N^2$	
Global Wire	$Y = \left(\frac{(N-1)!}{(N-5)!}\right)^4$	
All possible signal assignments to global wires in all quadrants. Four wires per quadrant and four quadrants in total. One signal is omitted as it owns the global bus.		
Local Wire	$Z = \left(\frac{N!}{(N-4)!}\right)^2$	
All possible signal assignments to local wires overall. Four wires per half.		
Track Jumping	$W = N^{48-C*2.2}$	
All possible signal assignments to the estimated wires available for track jumping as a function of the number of non-GPIO components used in the design.		
	$Permutations = X * Y * Z * W$ $N =$ number of unique signals $C =$ number of non-GPIO components	(2)

Using the formulation in Equation 2, to find the added complexity of completely enumerating signal to routing resource assignments for a moderately difficult design with nine non-GPIO components and eight unique signals yields:

$$X * Y * Z * W = 2.64e45 \quad (3)$$

If we remove track jumping from the assumptions, that greatly reduces the complexity but not enough to warrant further investigation:

$$X * Y * Z = 8.99e19 \quad (4)$$

For even a modest design, computing all permutations of signal to wire assignments explodes. For this reason complete enumeration was abandoned in favor other methods.

Boolean Satisfiability:

Rutenbar et al. discusses a formulation to find valid placements using Boolean satisfiability (4). In their formulation, possible placement locations for components are encoded into Boolean literals and clauses. Clauses represent constraints such as component to routing resource connectivity and limiting signals to only one resource within a channel. This method discovers clauses that must inherently be true or false and prunes the search space of those solutions that cannot be routed accordingly.

The authors make a point to note that creating Boolean functions to represent all detailed routes in not practical. They made a compromise by globally routing nets and creating small bounded regions over which to use Boolean satisfiability to find a detailed route. The architecture the authors used was highly homogeneous. In fact, each region could be described by three parameters: W the channel width within the region, F_c the count of tracks adjacent to each terminal, and F_s the number of neighbors per side to which a track is adjacent within a switch-block. First, the PSoC architecture has no equivalent architectural element to a switch-block. Second, there is no structurally direct equivalent to a channel. One could force the analogy by claiming that all global wires within a quadrant or half comprise a channel but that leaves local wires and the global buses unaccounted. And finally, the number of connections between Wejs and adjacent routing resources varies between and within groups of similar components. Even stretching analogies, none of W, F_c, or F_s can aptly describe the architecture.

VPR:

The standard implementation of VPR uses the Manhattan distance estimate of routability (2). As discussed earlier in the motivation section, semi-perimeter metrics do not accurately gauge the routability of a placement on this particular architecture. While VPR allows its users to supply custom cost metrics to target alternative architectures, it does not allow custom move sets.

The cost function used by VPR is:

$$Cost = \sum_{n=1}^{N_{nets}} q(n)\left[\frac{bb_x(n)}{C_{av,x}(n)} + \frac{bb_y(n)}{C_{av,y}(n)}\right] \quad \textbf{(5)}$$

The default cost is a function of the number of vertical and horizontal channels covered by the bounding Box bb of the net, an adjustment parameter $q(n)$ which varies depending upon the number of terminals on the net, and a measure of the average channel capacity C_{av} within the area covered by the bounding Box.

Independence and PathFinder:

The placement and routing algorithm Independence makes essentially no assumptions about the architecture other than the goodness of routes are able to be graded and that routability is a good estimate of the goodness of a placement. As it was designed to be adaptive, both during placement and routing, it is capable of finding placements for unique heterogeneous architectures. Independence uses simulated annealing with a routing algorithm tightly integrated into the annealer's inner loop to grade placements. The PathFinder router (5), is congestion aware and maintains state throughout the routing operation to track the overuse of each routing resource. It is publically available knowledge that Cypress Semiconductor currently uses Independence and PathFinder to perform place and route for some PSoC architectures. While Cypress' current solution does work, the need to perform full trial routes after every move is heavy handed. Being able to use an architecture adaptive algorithm without such strenuous routing would help further increase performance.

Equation 6 describes the cost of using a routing resource n during the current routing iteration under PathFinder. Under the formulation, b_n covers the base cost of using the resource, h_n introduces penalties if, historically, the resource has been congested during previous iterations, and p_n is a function of the current occupancy of n during the current iteration.

$$c_n = (b_n + h_n) * p_n \quad \textbf{(6)}$$

The cost function used by Independence is:

$$\Delta C = \frac{\Delta WireCost}{prevWireCost} + \lambda * \frac{\Delta CongestionCost}{CongestionNorm} \quad \textbf{(7)}$$

The cost function is dependent upon both the total count of routing resources used and the congestion of those routing resources. While grading a net, the route-graph is traversed and both wire use and congestion are recorded. The congestion cost increases as the occupancy exceeds the capacity of the routing resources. In Equation 7, the λ parameter weights the congestion cost and is generally tuned to be inversely proportional to the density of the architecture's interconnect. Those architectures with limited interconnectivity, such as the PSoC devices, would incur steeper penalties from overusing congested resources rather than using additional resources.

$$WireCost = \sum_{i=1}^{N} NumRoutingResources_i \quad \textbf{(8)}$$

$$CongestionCost = \sum_{i=1}^{R} \max(Occupancy_i - Capacity_i, 0) \quad \textbf{(9)}$$

Before annealing begins, all components are placed randomly and all nets are routed using the PathFinder routing algorithm. After every component swap made by the annealer, all nets of the affected logic blocks are ripped up and rerouted. It is possible, after many

moves that, because only a subset of all nets is rerouted, the reported congestion does not accurately reflect the true congestion of each resource. Therefore, at the end of a temperature iteration, the congestion history is adjusted according to the current congestion and the history in preparation for routing during the next iteration and all nets are rerouted. Keep in mind that PathFinder was originally intended to be run after placement had already finished; however, since placement and routing are integrated in Independence, congestion changes after each annealing operation. In light of this, Independence includes a decay factor while computing the congestion history to reduce the affect of congestion in areas that might no longer have high net density.

The adaptive nature of the Independence algorithm is appealing because of its ability to overcome non-uniformity and heterogeneity. Hephaestus borrows heavily from Independence by integrating a lightweight routing algorithm in the annealer's inner loop. The new tool diverges from Independence by not using full-fledged trial routes or considering the congestion.

Hephaestus:

Even with a thorough understanding of the PSoC architecture, architecture specific metrics are difficult to formulate due to the heterogeneity of both components and boxes, and the constrained nature of available resource. Independence became a large contributor to the inspiration for this research. Its ability to overcome architecture idiosyncrasies in an adaptive way was attractive and Hephaestus, the tool developed as part of this research, borrows heavily from some of its strengths. Hephaestus takes a graph of the architecture and performs simulated annealing to place a design. As we will see, Hephaestus also delegates the responsibility of routing to the annealer. Periodic quick connectivity checks are performed to determine when the design has been successfully placed. The move set chosen for the annealer is non-specific to the architecture and is capable of handling the architecture’s complexities.

Annealing:

The current implementation of Hephaestus uses simulated annealing to find placements, and lightweight routing to guide and grade the current placement. Hephaestus takes both a hyper-graph representing the architecture and a user mapped design. The design informs the annealer of the component to signal connectivity and is used by the annealer to color ownership of routing resources and Boxes accordingly.

The internal representation used by the annealer converts the original hyper-graph describing the architecture. After converting the hypergraph, routing resources (hyper-wires) become nodes and switches become edges connecting nodes. A 1:2 relationship now exists between edges and nodes--one edge is adjacent to at most two nodes. This conforms to the standard PathFinder routing graph formulation.

$$MovePerIteration = 10 * (N_{blocks})^{1.33} \quad \textbf{(10)}$$

$P =$ Previous Grade, $C =$ Current Grade, $T =$ Temperature

$$Goodness = -\exp\left(\frac{P - C}{T}\right) \quad \textbf{(11)}$$

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

Table 2. Temperature schedule as outlined in (2). The same schedule is used by Hephaestus to keep the acceptance rate near 44% throughout the annealing process

Many of the parameters of the annealer are borrowed directly from VPR. The cooling schedule, termination condition, and moves per temperature iteration are those used by VPR. The number of moves per temperature iteration is optionally tunable at runtime for adjusting the quality of the placement. Experimentation with Hephaestus has shown that the move count can drastically be decreased and still yield routable placements while

dramatically reducing the algorithm's runtime. The temperature schedule dynamically changes as a function of the fraction of accepted moves during the last temperature iteration. The variable α determines what fraction of temperature should be kept during the next iteration. Thus, the temperature drops rapidly while the acceptance rate of all (good, bad, and neutral) moves is high.

The annealer step fulfills two purposes: it makes perturbations to the location of components, and it also has the responsibility of assigning signal ownership to wires. In more direct terms, the annealer is in charge of both placing and routing the design. To empower the annealer to perform routing, the annealer's move set also includes an operation to extend signal ownership from a wire to an adjacent wire. The traditional component swap move is also present. Rather than simply assigning signal ownership at random, by relying on the extend signal move, the current implementation grows from pre-existing clusters of similarly owned wires.

Every wire in the device either has one owner or is unowned. A wire can only be owned by one signal. If, during annealing, a signal asserts that it now owns a wire currently belonging to a different signal, ownership changes.

While Hephaestus is implemented from scratch, its basic structure and flow is superficially similar to other annealers. Hephaestus makes a single move, re-grades all nets that were perturbed by that move, and then probabilistically accepts or rejects the moves depending upon the current temperature.

```
Hephaestus (Netlist,G(V,E)) {

      // All components are assigned random Boxes before annealing begins
      createRandomPlacement(Netlist,G(V,E));

      // Each net in the netlist is graded after random placement of all
      // components. Initial grade serves as a baseline to gauge improvement
      foreach (Net in Netlist)
            Net.grade = grade(Net);
            Grade += Net.grade;

      while (terminatingCondition() == false) {

      // Check whether the requisite number of moves have been made
      // this iteration. If not, perform one move
            while (innerLoopExitCondition() == false) {

      // Make a single move and collect the nets that have been perturbed
      // because of the move. For each of these, regrade the net and
      // accumulate the difference in grades
                  Perturbed = makeMove(G(V,E));
                  foreach (Net in Perturbed) {
                        Net.delta = grade(Net) – Net.grade;
                        Delta += Net.delta;
                  }

      // Choose whether to accept the move or not. All zero cost moves
      // and moves which improve the overall grade are outright accepted.
      // Detrimental moves are probabilistically accepted as a function of
      // the current temperature.

      // If the move is accepted, increment the count of accepted moves
      // and adjust the overall and per-net grades accordingly
                  if (accept(Delta,T) == true) {
                        Accepted = recordAccept();
                        Grade += Delta;
                        foreach (Net in Perturbed)
                              Net.grade += Net.delta;
                  }

      // If the move is rejected, undo it
                  else undoMove(G(V,E));
            }
```

```
        // After the iteration finishes, adjust temperature proportional to the
        // count of accepted moves
              T = updateTemperature(Accepted);
        }
}
```

Figure 5. Pseudo-code outlining the general flow of the Hephaestus annealer. The cost function for grading the nets is selectable. One option is to grade nets by counting the number of disconnected subsets. Another is to create a minimum spanning tree between the net's terminals over the ownership graph.

Grading:

To grade a placement, a minimum spanning tree (MST) on the ownership graph is constructed for each signal. For each signal A, wires owned by A cost 1 while wires not owned by A (either owned by another signal or unowned) cost 10. The MST connects together all of the Jacks associated with that signal. We formulate the problem by creating a graph of the Wejs (which become nodes) to be connected with a single edge separating each. The weight of each edge is found by performing Dijkstra's shortest path algorithm between each pair of nodes over the ownership graph. Once the edges have been weighted, Prim's algorithm is used to construct the minimum spanning tree. The grade for a given signal is the sum of the weights of the edges in the minimum spanning tree.

The formulation allows for unowned nodes to be included in the path between two Jacks. This allows signals to prefer shorter paths over long paths. Long circuitous paths consisting of a signal monopolizing many routing resources might successfully connect two Jacks; however, those routing resources might better be used by other signals. By preferring shorter paths, the likelihood of consuming resources to the exclusion of other signals is reduced. A fine balance must be struck between weighting the cost of owned and unowned nodes as the unowned penalty should not be such that it precludes preferring a shorter but unowned path over one that is long and circuitous. A small cost is attributed to including owned nodes to help track wire length.

By crafting a metric dependent both upon the length of paths and the inclusion of unowned nodes, the metric performs well at being both predictive and descriptive. By minimizing the cost, both the number of nodes owned by a signal and the routability of the signal is optimized. As the signal approaches routability, the occurrence of unowned penalties decreases. As the signal reduces the number of owned nodes, the owned penalties likewise decreases.

Optimizations:

An incremental approach to grading has been implemented to reduce the runtime of the algorithm. Whenever the annealer makes a move, only the disturbed signals (those that have been extended, those that have been extended upon, or those that emit from the Jacks of moved components) have their grade recalculated. This significantly reduces the runtime duration compared to recomputing the grades of every signal after every move.

Hephaestus Annealer Specification:

This section describes in detail the implementation specifics of the Hephaestus placement tool. It includes the preferred cost metric and moves, as well as other alternatives that were considered, implemented, and tested.

Locking Elements:

The tool permits the locking of components to Boxes and locking ownership of wires.

1. Several GPIO Boxes of the PSoC device are reserved for special functions and cannot be considered as locations during the annealing process. Locking those Boxes at the beginning of annealing denies any GPIO from occupying the restricted locations.
2. Wejs of occupied Boxes cannot have their ownership overwritten. As Wejs are the sinks during routing, the appropriate signal must always own them unless the Wej's Box is unoccupied. When the annealer places a component into a Box, signals take ownership of the Box's Wejs. These Wejs are then locked to prevent overwriting. When the Box is vacated, the Wejs are unlocked to allow other signals to take ownership.

Initial Placement of Components:

Regardless of choice of move set or cost metric, the initial component to Box assignments are chosen randomly when annealing begins. All non-Wej wires are left unowned at the start of annealing. For all move sets, the probably of selecting any of the moves is weighted equally. Within the tool, there is the ability to weight some more heavily than others but this tuning parameter was not explored.

Move Set #1 (MS1):

This move set attempts to utilize only small non-coordinated moves. No series of moves are performed in succession. None of these moves use adjacency information to direct the search. This set allows components to move between valid Boxes within the group, and signals to take ownership of wires adjacent to owned wires. This set minimizes the complexity of the available moves to ensure the entire problem space is explored.

Swap Component:

1. Choose a random component from the design.
2. From the component, find which Box it currently resides within.
3. Randomly find a second Box within the same group, regardless of occupation.
4. Store component A from the first Box.
5. Store component B from the second Box.
6. Vacate both Boxes, unlocking all Wejs.
7. Occupy the second Box with component A.
8. Occupy the first Box with component B.
9. For both components, their signals take their Wejs and lock all such Wejs.

Extend Signal:

1. Choose a random signal from the design.
2. Choose a random wire which is owned by that signal.
3. Find an unlocked neighbor to the wire which is either empty or owned by a foreign signal.
4. Assign ownership of neighbor to the signal from the first step.

Move Set #2 (MS2):

The second move set deviates from the first by including added coordination and series moves. The goal is to direct exploration of the problem space towards choosing moves that will lead to successfully routing signals. The important change to this move set is the look one neighbor away after performing a component swap. This look ahead helps direct the annealing process, making easy decisions early in order to get to the more difficult decisions sooner.

Swap Component

1. Perform MS1 Swap Component
2. Afterwards, for each Wej of all occupied Boxes involved in the move, examine all neighbors of that Wej.
    a. If a neighbor is already owned, do nothing.
    b. Otherwise, if an unowned neighbor exists, own that neighbor, picking randomly if there are more than one.
    c. Otherwise, pick a random neighbor and own it.

Extend Signal:

1. Choose a random signal from the design
2. Choose a random wire that is owned by that signal
3. Find an unlocked neighbor to the wire which is either empty or owned by a foreign signal.
4. Pick empty neighbors in preference to neighbors owned by a different signal.
5. Assign ownership of neighbor to the signal from the first step.

Neither move set is particularly complex, but it should be noted that they both do include a small amount of directedness. Rather than assigning ownership of wires to signals at random, only neighbors of owned wires may be targeted by the signal.

Cost Metric #1 Connectedness of Components (CM1):

The first cost metric focuses on the number of disconnected Wejs. Using this metric, adjacent wires, including at least one Wej, are clustered according to ownership. If, after clustering, two or more disconnected clusters with the same ownership exist then a route has not yet been discovered between the Wejs. Under this metric, the grade for the signal is equal to the number of disconnected clusters of that signal.

1. For a given signal, populate a list of Wejs that that signal must span.
2. The initial grade equals the number of nodes from Step #1.
3. For each node in the list:
    a. Remove that node from the list
    b. Recursively visit the neighbor nodes owned by the signal.
    c. If the neighbor is another node from list in Step #1, remove it from the list, decrement the grade, and continue visiting neighbors.
    d. Continue from b. until no further nodes with appropriate ownership are available and then repeat from #3.
4. The score for that signal is the final resulting grade (ranging from 1 to the number of Wejs for that signal).

For a sparse design, the number of zero cost generating moves produced by this cost metric makes up the majority of all moves, particularly for routing moves. The inflated number of accepted moves (because a zero cost move is automatically accepted by the annealer) quickly drove down the temperature due to VPR's adaptive cooling schedule, causing this cost metric to rapidly exhibit greedy behavior. One quick remedy to ameliorate this was to only count the number of accepted and rejected non-zero cost moves when adjusting the temperature schedule. The range of grades produced by this cost metric was also problematic as the very narrow set of possible grades made it difficult to determine the goodness of a placement; many different placement configurations can result in the same grade.

Cost Metric #2 Minimum Spanning Trees (CM2):

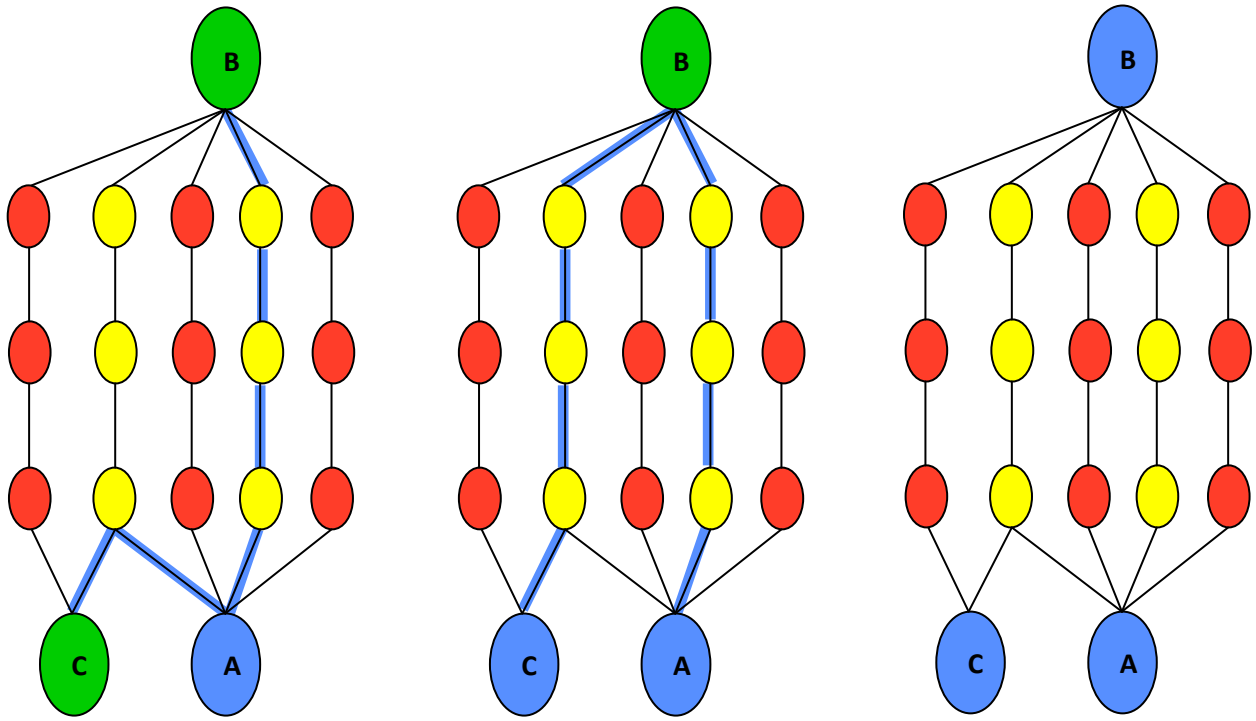
The range of grades produced from the first cost metric, and the large number of zero-cost generating moves, spurred the search for a more detailed cost metric. This metric was designed to promote reducing the number of disconnected components and to reward moves that extend signals towards their targets.

After each move that disturbs a signal (extending signals, having a pre-existing signal overwritten, moving components which in turn affects the signals at Wejs), all disturbed signals have their minimum spanning trees recomputed. The grade of the placement is the sum of the costs of the minimum spanning trees. The algorithm for finding the minimum spanning tree to connect all ports on the signal follows:

1. For a given signal, find all of the Wejs to which that signal must connect and put them into the unvisited list.
2. Set a cost for every wire node in the architecture according to the following rules:
    a. If the node belongs to the signal under consideration, the node is marked with cost = 1.
    b. If the node does not belong to the signal, the node is marked with cost = 10.
3. Remove one Wej from the 'unvisited' list and add it to a 'visited' list
4. Until the list of unvisited Wejs is empty (essentially Prim's algorithm)
    a. Maintain a queue that stores the current nodes to be visited (traversal of neighbors) and prioritizes those nodes according to the cost to reach it from the originating Wejs.
    b. Clear the priority queue and add all of the visited Wejs to the queue with cost zero.
    c. Do until an unvisited Wej is reached: Pick a node from the queue with the least cost to reach and mark it so it is not revisited, then for each neighbor:
        i. Add the current cost-to-reach of the node to the ownership cost of the neighbor and store that value as the cost-to-reach of the neighbor in the priority queue.

d. Move the found Wej from the unvisited list to the visited list, add the cost-to-reach to the cumulative cost.

After all Wejs have been visited, the grade of the signal under inspection is set to the cumulative cost. The grade of the placement is the sum of the grades of the signals.



Cost from A => B : 5
Cost from A => C : 3
Add C to MST,
Grade = 0 + 3

Cost from A => B : 5
Cost from C => B : 5
Add B to MST,
Cost = 3 + 5

Figure 6. Example of using Dijkstra's shortest path algorithm to formulate the minimum spanning tree problem. Only the lettered nodes are part of the minimum spanning tree. A single path (the shortest path, **overlaid**, atop the ownership graph) connects each node. At each iteration, we find the nearest node not in the tree and add it to the tree. The cost to get to that node is summed into the grade. Because we only calculate the shortest paths between nodes in the tree and those not in the tree, we might double count some of the path in the final grade. Double counting in this context is okay, we simply want the sum of the lengths of the shortest paths. (Colors: Blue = node in the MST, Green = nodes to include in the MST, Red = unowned intermediary nodes, Yellow = owned intermediary nodes)

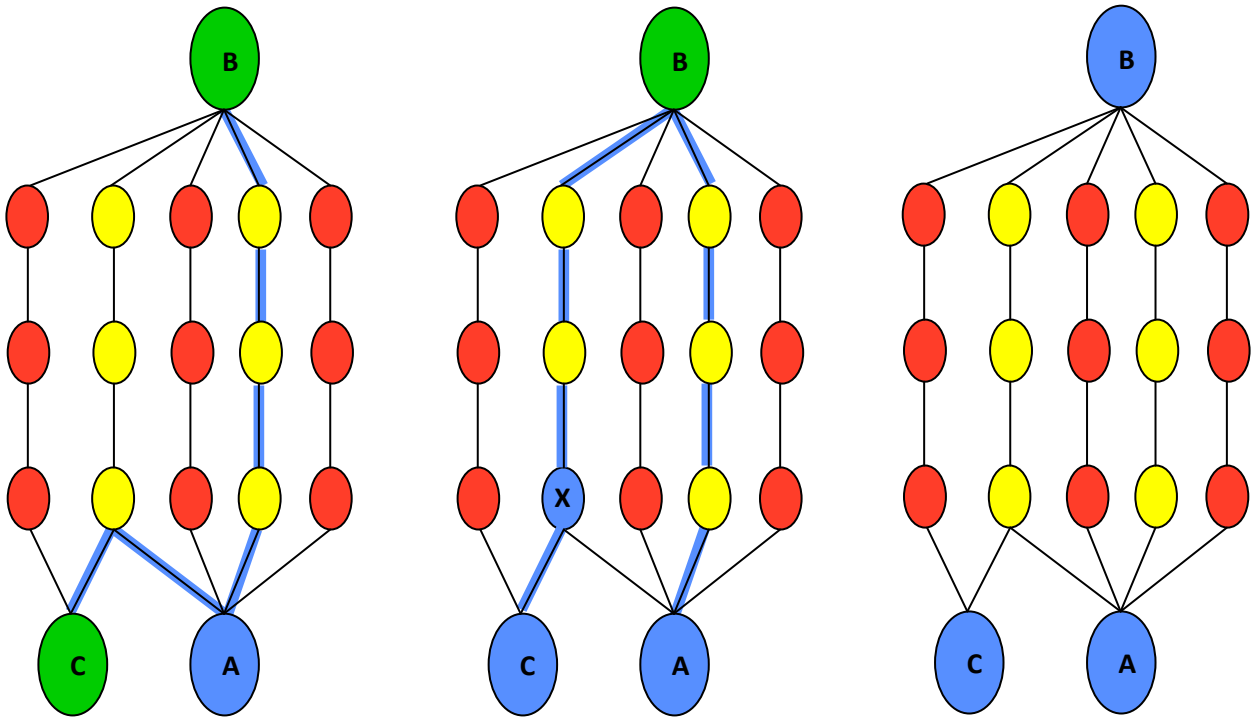


Figure 7. Similar to the formulation in Figure 6 this depicts how one would find an approximate minimum Steiner tree (using Takahashi and Matsuyama's method). After finding the shortest path between a node in the tree and a node outside of the tree, add both the node and all nodes along the path to the tree. This prevents us from double counting as was previously the case. As the intermediary nodes now also act as sources while finding the next shortest path, we have additional opportunities to find short paths.

Cost Metric #3 Steiner Trees (CM3):

The third cost metric is an alternative to the second. It operates similarly with two exceptions. First, rather than starting from a single Wej (as per step #2 in CM2) a search is started at all Wejs simultaneously. When any Wej encounters another they form a connected cluster (including the connected Wejs and the path connecting them together) and the search repeats now growing from the connected cluster and the remaining Wejs.

A small modification to step d. from CM2 is necessary:

d. Move the found Wej from the unvisited list to the visited list, backtrack along the path and add each wire to the visited list as well, add the cost-to-reach to the cumulative cost.

By adding the intermediary routing resources along the path connecting the two Wejs, future searches will start at the Wejs and those intermediary wires. This produces a heuristic Takahashi and Matsuyama Steiner Tree approximation (6).

Test Cases Descriptions:

A modest set of six test cases has been selected to compare the efficacy of the placement tool. Two of the six are commercial designs, while the remaining four are synthetic, built to intentionally stress various attributes of the PSoC architecture. These test cases were specifically chosen because of their difficulty to find valid placements with conventional tools. While the test case set is small, the members were hand selected with careful consideration to ensure Hephaestus was confronted with a variety of different designs. These test cases cover stressing component placement, port placement, the interconnect, and, combinations thereof.

As mentioned while describing the architecture, all instances of analog muxes in these test cases were fused into one static net.

Commercial #1:

Components	GPIOs	Max Fanout	Average Fanout
13	25	13	~2.4

This test case has a massive static net that spans thirteen ports. The remaining nets are small, connecting only two components. Compared to the other designs it has the greatest number of GPIOs and the greatest number of nets. Despite the massive net, the

average fanout across all nets is only approximately 2.4 connections because there are so many two terminal nets. This test case stresses the GPIO ring and large net routing.

Commercial #2:

Components	GPIOs	Max Fanout	Average Fanout
15	17	17	4

The other commercial test case is the largest design of those chosen. It boasts 15 components and the largest net overall, connecting 17 ports. Because of the complicated and dense design, it originally relied upon user-specified fixed port placements to help guide the original Cypress tool to find valid placements. Those forced placements have been revoked to test whether Hephaestus can successfully find valid placements without the need for human intervention. The 13 nets of this design have an average fanout of approximately 4 connections per net. This seems like it should be the most complex of the designs within the selected set of test cases but another happens to be the more difficult. All aspects of the architecture are tested with this design.

Synthetic #1:

Components	GPIOs	Max Fanout	Average Fanout
9	6	7	3

The simplest of the selected test cases, this was chosen to provide a good baseline for the others.

Synthetic #2:

Components	GPIOs	Max Fanout	Average Fanout
13	2	9	5

This test case contains routing with greater fanout. The average fanout is the approximately 5 connections per net, higher than any other test case. This is in large part due to the highly connected operational amplifiers. Between the number of components and the high average fanout, this test case provides a good overall test of dense component placement. Of the tests, this one in particular proves to be the most difficult as the results later show. This is likely in part to the high average fanout combined with the constrained nature of the architecture.

Synthetic #3:

Components	GPIOs	Max Fanout	Average Fanout
4	19	5	~3.7

This test case, while light on components, consists of the greatest number of GPIOs and amuxes. The average fanout is only about 3.7 connections per net, but the five amuxes are moderately well connected (four of which have a fanout of 5, the remaining of 3). This test case stresses the interconnect.

Synthetic #4:

Components	GPIOs	Max Fanout	Average Fanout
7	22	5	2.4

While the component count is rather small, this test case contains a fair number of GPIOs and many small nets. It boasts the least average fanout at ~2.4 connections per net; however, it counterbalances this by containing twenty-one nets. The largest net only has 5

connections. This test case stresses placement by flooding the routing interconnect with many small nets.

Results:

All experiments were performed on a machine running Linux kernel 3.0.0 with an Intel Core 2 Duo P8600 processor @2.400GHz and 4GB of DDR3 SDRAM. The application was compiled with the GNU Compiler Collection's g++ compiler version 4.5.3 with the O2 optimization flag set. The machine was rebooted prior to running the batch of experiments and only the necessary processes and window manager were started beforehand. The experiments vary between choice of move set, choice of cost function, choice of test case, and random initial seed. Each experiment was run in series one after the other and each experiment was run multiple times with the same parameters to average out variation caused by external sources (e.g. background OS processes).

For the sake of brevity, all allusions to Steiner trees should be understood to be referencing the Takahashi/Matsuyama Steiner tree approximation formulation. The data presented in Tables 3 and 4 are the average over both the identical experiments (same parameters and seed), and experiments with differing initial seed. Each experiment was run over 5 different initial seeds, and repeated 5 times.

A third move set which performed random signal assignment to routing resources was included in the experiments but the results are omitted. Without any directedness, this third move set failed to produce any routable placements. This is important to note, but otherwise uninteresting because of the 100% failure rate. Some amount of directedness is necessary to ensure signals are being assigned to adjacent routing resources.

The first set of experiments followed the suggestion put forth by VPR and set the number of moves per iteration according to Equation. 10. Only the fixed function units and GPIOs were included in the count of placeable objects. At this number of moves per iteration, the results were nominally equivalent for all the non-trivial move sets and the two cost functions. Equation 12 shows on average, over all of the test cases, the number of moves

per iteration that were made. The results from these experiments are available in Table 3 and Figure 8. While all designs managed to find valid placements at this number of moves per iteration, we do see a discrepancy in Synthetic #2 between the runtimes of the directed move set which outperforms the undirected move set by roughly at least a factor of two.

$$10 * 25^{1.33} \approx 723 \quad \textbf{(12)}$$

For the second set of experiments the number of moves per iteration was artificially reduced to force faster annealing in an effort to expose weaknesses between move sets and cost functions (Table 4). The suggested number of moves per iteration was reduced to 300 moves. While some move set and cost function combinations continued to perform well under the faster annealing schedule, others began to yield unroutable placements.

Test Case	Undirected & Spanning	Directed & Spanning	Undirected & Steiner	Directed & Steiner
Commercial #1	10.92	9.02	9.73	9.56
Commercial #2	22.22	14.52	18.58	16.78
Synthetic #1	7.84	9.96	9.82	8.12
Synthetic #2	39.21	12.49	32.59	16.00
Synthetic #3	6.92	6.51	7.65	7.52
Synthetic #4	2.87	3.07	3.29	3.83

Table 3. Runtime (seconds) for the test cases for a mix of move set and cost function. None of the tests failed using the number of moves per iteration in Equation 12. Most runtimes were nearly equal except for Synthetic #2 which shows a strong lead while using the directed move set. Directed also performs better in Commercial #2, the other dense design. Spanning outperforms the Steiner cost function nominally in synthetic test cases #3 and #4.

Over all of the identical experiments, the average standard deviation of the runtimes was 0.192 seconds. Those experiments that ended quickly only differed by 10's of milliseconds, whereas the long experiments, especially those not finding a valid placement, could be off by 100's of milliseconds.

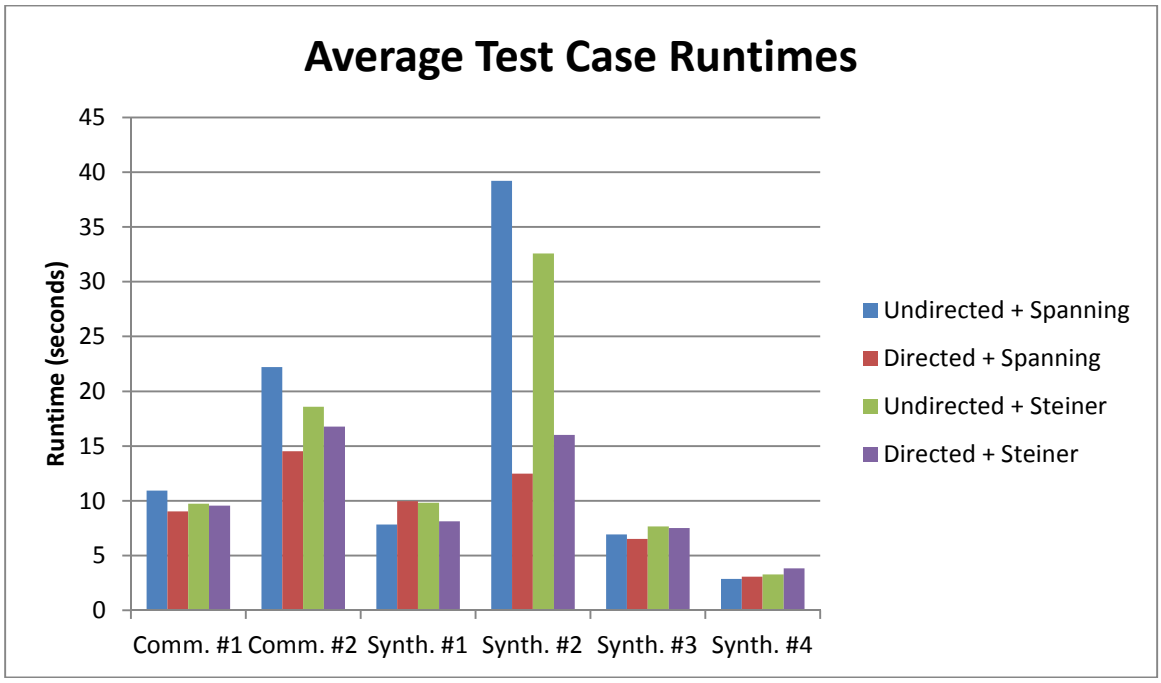


Figure 8. Graphical view of the data from Table 3.

It is also important to be aware that while the experiments using the Steiner tree cost function consistently produce placements with lower overall grade than those placements found using the Spanning tree cost function, this too could be misleading, since they use different cost models. Because the Steiner tree approximation algorithm includes the intermediary routing resources as sources for the next iteration of the algorithm, those intermediary resources are not double counted as is the case when using the spanning tree algorithm.

	Moves / Function	Runtime (s)	Grade	Iterations	Success Rate %
Comm. #1	Undirected, Spanning	7.962	142.40	121.20	100
	Directed, Spanning	4.421	142.40	68.20	100
	Undirected, Steiner	8.878	129.00	102.60	80
	Directed, Steiner	4.992	127.00	73.20	100
Comm. #2	Undirected, Spanning	8.647	148.20	75.20	100
	Directed, Spanning	8.321	146.40	71.80	100
	Undirected, Steiner	7.471	119.80	62.60	100
	Directed, Steiner	8.772	118.00	70.00	100
Synth. #1	Undirected, Spanning	4.513	73.00	107.40	80
	Directed, Spanning	2.509	70.00	67.40	100
	Undirected, Steiner	3.742	67.80	82.80	80
	Directed, Steiner	4.762	66.80	112.20	60
Synth. #2	Undirected, Spanning	9.420	117.20	98.40	100
	Directed, Spanning	3.745	121.00	39.60	100
	Undirected, Steiner	17.176	103.20	161.00	40
	Directed, Steiner	10.582	98.60	103.80	80
Synth. #3	Undirected, Spanning	2.159	100.60	40.00	100
	Directed, Spanning	2.133	97.20	39.40	100
	Undirected, Steiner	2.222	79.20	38.20	100
	Directed, Steiner	2.244	79.20	37.80	100
Synth. #4	Undirected, Spanning	1.817	83.20	98.20	80
	Directed, Spanning	1.445	80.80	70.60	100
	Undirected, Steiner	1.728	76.80	81.20	80
	Directed, Steiner	2.113	71.60	93.20	100

Table 4. Runtime, overall grade, iteration count, and success rate for different combinations of move set and cost function over the set of test cases.

There are several important trends from the results:

- On average, the directed move set yielded placements whose overall grade was less than placements found using the minimal move set.
- On average, a valid placement was discovered after fewer temperature iterations, and thus faster, using the directed move
- With the MST cost function, the minimal move set failed to consistently produce routable placements. The directed move set succeeded in every instance.
- Using the Steiner tree cost function, neither move sets consistently produce routable placements.

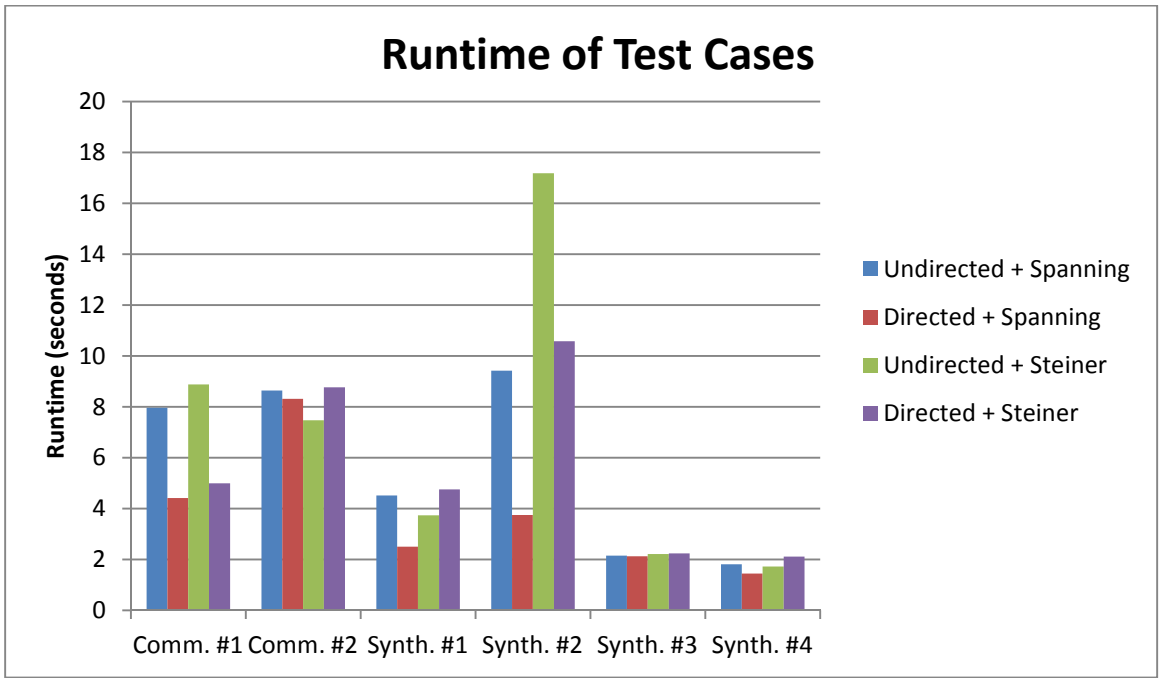


Figure 9. Graphical view of the data from Table 4.

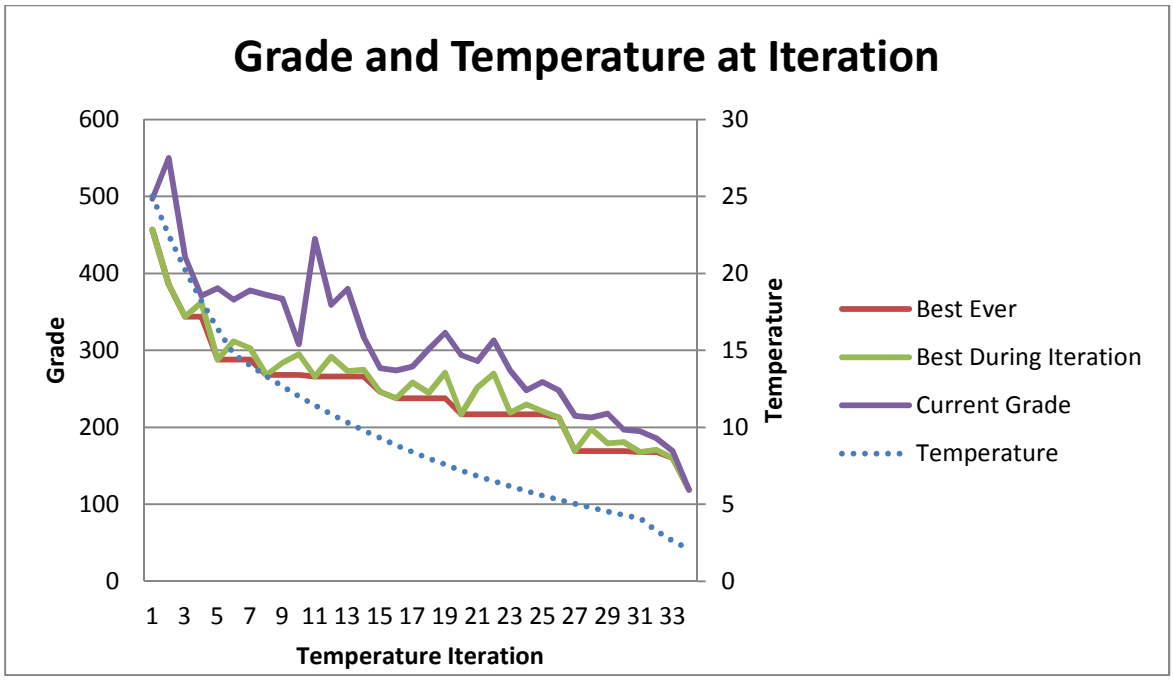


Figure 10. The grade and temperature of the current placement at the end of each temperature iteration. The majority of annealing occurs between iterations 7 and 31. After iteration 31, the temperature drops off and quenching begins. Soon afterwards a valid placement is discovered.

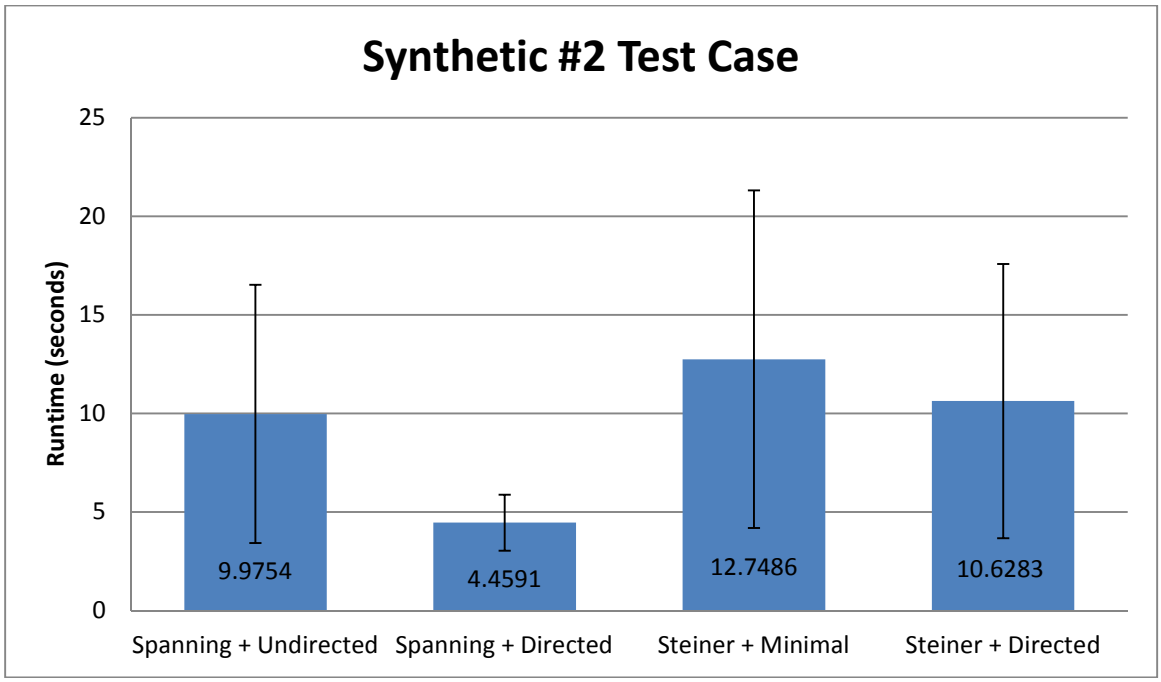


Figure 11. The mean runtime of 100 runs of synthetic test case #2 with bars marking one standard deviation away. Each combination of move set and cost function was run 10 times with 10 different seeds. These values were drawn from a different set of experiments than those recorded in Table 4.

Of the experiments run, the Spanning tree cost function along with the directed move set fared the best. This combination, on average, found solutions faster than the other combinations 50% of the time and was competitive with the best during the remaining runs. While the Spanning tree and directed move set combination always produced routable placements during all experiments, this does not necessarily hold true if larger sets of experiments are run. For example 2 of 30 randomly seeded runs of synthetic test case #2, which caused the most trouble for most combinations, failed to produce routable placements.

Figure 11 shows the result of running each combination of move set and cost function on synthetic test case #2 ten times each with different seeds. Be aware that the data in Figure 11 comes from a different set of experiments than that in Table 3 or Table 4. Synthetic test case #2, one of the more difficult set of experiments as evidenced by the high failure rate,

showed the most variation between combinations. For this particular set of experiments, using the Spanning tree cost function both yielded placements that could be routed more frequently than the Steiner trees, and of those that found valid placements, found them faster. The same is true for using the directed move set over the minimal move set.

Synthetic #2 exhibits the greatest range of diversity between combinations of cost function and move; the results from this particular test case aligns with the results seen in the other test cases. The Spanning and Directed combination clearly excels in comparison to the other combinations. The mean runtime for this particular combination is approximately 4.5 seconds, at least two times better to the next nearest combination (Spanning and Minimal @ ~10 seconds) and nearly three times better than the worst (Steiner and Minimal @ ~12.7 seconds).

Conclusion:

The results of the experiments suggest that it is possible, for sufficiently small and constrained architectures, to excise the need to perform full trial routes as part of the cost function when performing placement. The lightweight, congestion-unaware routing algorithm implemented as part of this research remains architecture adaptive, capable of discovering valid placements given a hyper-graph representing the architecture.

The results support that directedness of the move set is beneficial (and indeed necessary) to finding valid placements. For reasons not completely explored, the Spanning tree cost function outperforms the Steiner tree cost function, yielding valid placements more frequently. The more complex moves introduced into the directed move set are still relatively conservative and only speed up decisions that will need to be made regardless in the future, which forces the algorithm to get through the easy decisions faster. This research did not push the bounds on how directed a move set could become although that is an interesting direction for future work.

Future Work:

One direction that deserves additional exploration is the introduction of more directedness in the annealer's moves. Three such future extensions include:

1. Extending the second move set (MS2) to look two neighbors away during signal extensions and component swaps. By looking out two neighbors ahead, clusters of almost connected wires might be able to bridge their separation by choosing the unowned bridge.
2. Guiding extend moves with the second cost metric (CM2)'s spanning trees. If a signal could be aware of wires owned by other signals yet are included in its minimum spanning trees, those wires could be the perfect target for signal extension.
3. Several simplifications with regard to the complexities of the PSoC architecture were enforced. This was intentionally done to focus on the matter of formulating this alternative placement algorithm. Several such simplifications include fusing muxes into static nets, ignoring mandatory component-to-component constraints (under the assumption that the tool will still find valid placements for those components), and ignoring the existence of other special architectural elements such as at-most-one switch groups, the CapSense components, and voltage references. Future work should investigate properly observing muxes (i.e. placing such muxes atop the routing resources and routing between the shared and independent connections).

Works Cited

1. **Cypress Semiconductor.** *PSoC3, PSoC5 Architecture TRM Document No. 001-50235 Rev. *E.* San Jose, CA : Cypress Semiconductor, 2010.

2. *VPR: A New Packing, Placement and Routing Tool for FPGA Research.* **Betz, Vaughn and Rose, Jonathan.** London, UK : IEEE, 1997. Field Programmable Logic and Applications. pp. 213-222.

3. *Architecture Adaptive Routability-Driven Placement for FPGAs.* **Sharma, Akshay, Hauck, Scott and Ebeling, Carl.** Tampere, Finland : IEEE, 2005. Field Programmable Logic and Applications. pp. 427-432.

4. *FPGA Routing and Routability Estimation Via Boolean Satisfiability.* **Wood, R. Glenn and Rutenbar, Rob A.** s.l. : IEEE Transactions on Very Large Scale Integration Systems, 1998, Very Large Scale Integration (VLSI) Systems, pp. 222-231.

5. *PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs.* **McMurchie, Larry and Ebeling, Carl.** New York : ACM, 1995. Field-Programmable Gate Arrays. pp. 111-117.

6. *An Approximate Solution for the Steiner Problem in Graphs.* **Takahasi, Hiromitsu and Matsuyama, Akira.** 1980, Mathematica Japonica, pp. 573-577.