Hardware-Assisted Simulated Annealing with Application for Fast FPGA Placement

Michael G. Wrighton, André M. DeHon
California Institute of Technology
Computer Science, 256-80
Pasadena, CA 91125
{wrighton, andre}@cs.caltech.edu

ABSTRACT

To truly exploit FPGAs for rapid turn-around development and prototyping, placement times must be reduced to seconds; late-bound, reconfigurable computing applications may demand placement times as short as microseconds. In this paper, we show how a systolic structure can accelerate placement by assigning one processing element to each possible location for an FPGA LUT from a design netlist. We demonstrate that our technique approaches the same quality point as traditional simulated annealing as measured by a simple linear wirelength metric. Experimental results look ahead to compare quality against VPR's fast placer when considering the minimum channel width required to route as the primary optimization criteria. Preliminary results from an FPGA implementation show the feasibility of accelerating simulated annealing by three orders of magnitude using this approach. This means we can place the largest design in the University of Toronto's "FPGA Placement and Routing Challenge" in around 4ms.

Categories and Subject Descriptors

B.7.2 [**Integrated Circuits**]: Design Aids - *Placement and routing*.

C.1.3 [**Processor Architectures**]: Multiple datastream architectures – *Array and vector processors.*

General Terms

Algorithms, Performance, Experimentation

Keywords

Field-programmable gate arrays, simulated annealing, placement, design automation, reconfigurable computing

1. INTRODUCTION

The long runtimes demanded by placement and routing tools plague users of Field-Programmable Gate Arrays (FPGAs). In an era when synthesis can take minutes, placement runtimes can be hours or days for the largest designs. This problem is not mitigated by faster hardware as (1) FPGAs are getting larger faster than uniprocessor CPUs are delivering more, useful MIPS [1] and (2) traditional algorithms for placement generally scale superlinearly in device size. Consequently, we expect this problem only to increase with time. We further envision a day when the requirements for reconfigurable computing applications will include the capability to place and route a design in milliseconds or even microseconds. Recent developments show how to accomplish the latter [2, 3] of these, but placement has remained troublesome.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
FPGA'03, February 23-25, 2003, Monterey, California, USA.
Copyright 2003 ACM 1-58113-651-X/03/0002…$5.00.

Previous work on software placement technology has identified ways to speed up placements as much as 50x by sacrificing about 30% of the quality of reference tools [4, 5]. To give the reader an intuition of the runtimes involved consider that an 8000 LUT design can be placed by VPR with its `-fast` option in about 270 seconds on a Pentium IV 2.2 GHz workstation. So an accelerated software placer such as [5] would place the same design in roughly 5 seconds. Five seconds is still not fast enough for many applications requiring dynamic reconfiguration.

Since FPGAs and ASICs are becoming large enough to implement hundreds if not thousands of independent processing elements, it makes sense to investigate whether massive parallelism can be used to reduce the time required to find a quality placement. This would allow us to solve the problem such that the technology for placement would scale with our ability to fabricate larger and larger devices.

A traditional simulated annealing-based approach, aiming for best possible quality, to the placement problem sequentially performs swaps of random cell locations, accepting non-greedy moves with exponentially decreasing probability based on the current "temperature" of the anneal and the delta that would occur in the overall cost function. Following some cooling schedule, the temperature is reduced. At the outset of the cooling schedule, virtually all moves are accepted. As the end of the cooling schedule is reached, only moves that would improve the overall cost function are accepted. For suitable cost functions and sufficiently slow cooling, it can be shown that simulated annealing produces optimum placements [6].

Our novel approach is to build a parallel compute engine with a systolic architecture optimized for the placement problem. We assign each possible position for a LUT to a separate processing

element ("PE"). Then we consider all possible local swaps in parallel and accept those that would reduce local contributions to the cost function or that are randomly chosen based on the current "temperature." In Section 3, we sketch the structure for spatial computation that allows the PEs to communicate only locally, yet still drive towards a globally optimal solution. The state and logic requirements for the PEs are detailed in Section 4. In Subsection 4.5, we show how our approach provides the same search strategy as simulated annealing. Section 5 describes the benchmarks we performed to evaluate our solutions against a traditional simulated annealing for linear wirelength and against VPR. Further, we provide preliminary results showing that hardware implementations are practical in Section 6. Section 7 discusses how these results might be applied to dynamically reconfigurable computing systems and large-scale logic emulators. Finally, in Section 8, we document the future work that will be required to fully understand and optimize the power of our approach.

2. PRIOR WORK

As previously mentioned, within the traditional domain of sequential software placers, 50x improvements in speed are possible with only moderate quality losses. While further improvements are likely possible in this area, it seems unlikely that sequential software placers will increase in speed fast enough to keep up with the ever-increasing number of LUTs available on a chip.

We were inspired to consider massively parallel approaches to the placement problem by the force-directed algorithm described first for printed circuit board layout by Goto [7]. Force-directed algorithms for placement treat the placed netlist as if the wires are springs pulling connected components together. The algorithm models the behavior of those springs subject to constraints (*e.g.,* only one LUT allowed placement at a given location on the chip). Force-directed algorithms can give acceptable results, but often terminate trapped in local minimas [8]. We believed that the local nature of the algorithm was well suited to our needs and it formed the basis of our initial experiments. In [9, 10] a scheme to assign one SIMD processor to each cell of an ASIC design was described to accelerate force-directed placement. This work demonstrated that it is feasible to obtain orders of magnitude speedup by assigning every circuit component its own processing element. Unfortunately, this design depended on a large-scale supercomputer, broadcasts to update cell positions, and swaps between arbitrary positions. We show that this style of solution is now feasible using FPGAs and demonstrate how to limit the design to local operations in order to achieve scalability. We further show how the massively parallel design can avoid local minima.

An evaluation of parallel placement algorithms for FPGAs was documented in [11]. The authors attempted two parallelization strategies: First, they parallelized VPR's simulated annealing by allowing parallel swaps on an SGI Origin shared-memory multiprocessor. This yielded no meaningful speedup due to the overhead of synchronizing the processors. They were successfully able to achieve near-linear speedups by allowing each of the separate processors of an IBM-SP2 distributed memory multiprocessor to work on the simulated annealing of the entire device, periodically distributing the best placement to the others, an approach they called "Markov Chains." By accelerating the cooling schedule, any desired speedup is obtainable. With six processors used for a factor six speedup, however, the cost function reached 160% of the uniprocessor VPR result. We improve on this work by using FPGA-based PE's to perform parallel swaps between nearest-neighbor PEs. This allows us to employ efficiently many more processors by exploiting massive and cheap local bandwidth, avoiding synchronization overhead.

3. SOLUTION SKETCH

The basic idea behind our approach to hardware-assisted placement is that we assign each physical space where a LUT could reside its own PE. That PE is responsible for keeping track of which LUT it contains and which LUTs connect to its inputs and outputs. The PE knows its own position and keeps an estimate (assume a perfect estimate for now) of the positions of the connected LUTs. Simultaneously, the PE's negotiate with their neighbors to see which swaps would locally improve the total placement cost. Each PE assumes that its contribution to the current placement cost is the sum of the Manhattan wirelengths required to route to the estimated positions of LUTs connected to its inputs and outputs.

If we assume that position estimates are correct, then this is only slightly different from the traditional, sequential simulated annealing approach to the problem: The cells make local cost reducing swaps to greedily search for a local minima. By adding a suitable element of randomness (which is introduced in Subsection 4.3.3) to the swaps, we avoid local minima by using a simulated annealing-style cooling schedule.

As we are developing a scalable solution, we cannot make the perfect information assumption for hardware implementations. That is, we would prefer not to fully update the state of the placement engine every time a group of swaps is considered; simple update schemes after each set of potential swaps could require *O(N)* time. To deal with this, we insert a "position update chain," which snakes through the array of PE's. Every update interval this chain is shifted to update position estimates across the chip. The position chain carries identifiers for the LUTs (id's) as well as their positions in the current placement. Each time we cycle through the position chain communicating current LUT locations, the algorithm considers all possible local swaps a number of times (proportional to the size of the design). Note that this results in a design that can be implemented as a systolic array; all the processing elements are connected only to their nearest neighbors.

This raises several questions:

- What is the effect of information staleness on the algorithm? How many swaps can be taken between updates before the staleness causes problems?
- By what mechanism do the PE's determine with which neighbor to negotiate a swap at any given time?
- How long will the algorithm take to converge?
- How does the potential for unbounded fanout affect the practicality of creating a systolic data structure?
- By what mechanism is a netlist loaded into and then removed from the PE array?

The following sections answer these questions and detail the architecture.

4. THE DETAILS

To explain the details of our solution, we first describe the top-level structure of a hardware implementation. We then follow with an explanation of an individual PE. We continue with the pseudocode for the algorithm, which follows naturally from the circuit structure. This leads to a mathematical justification of how the algorithm performs a search through the solution space that is analogous to simulated annealing.

4.1 Top-Level Structure

To begin, let us start with a top-level view of an array of cells. As seen in Figure 1, the cells, which directly represent the 2-D placement, are connected to each of their four nearest neighbors. A position chain snakes throughout the design, enabling propagation of state information. We also use the position chain for bootstrapping an initial random placement into the design and shifting out a final placement. Every $H \times W$ shift cycles, the cells shift out their own current information, which is their location and the id number of the LUT they contain. On other shift cycles, the cells simply shift out the data received on the previous cycle, updating position estimates for any matching cells in their locally maintained connectivity lists. Note that all cells communicate only with their nearest neighbors.

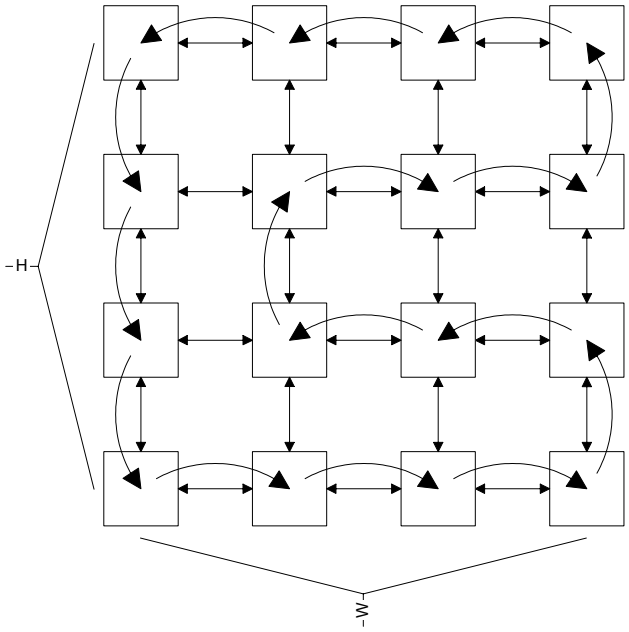

Figure 1. Interconnect between PEs and position chain. Arced arrows represent path of the position chain, straight arrows show nearest-neighbor interconnect for swapping.

4.2 Individual Processing Elements

For a view of an individual processing element, see Figure 2. Note that it includes locally all the components needed to manipulate its part of the placement. A content addressable memory ("CAM") stores the list of LUTs connected to inputs and outputs of the LUT currently at the cell's position. It provides indexing into a memory that stores estimated locations of those LUTs. The control logic includes accumulators and comparators to compute the delta cost that would be associated with swapping a cell. A randomness generator provides for the stochastic behavior that gives us simulated annealing's search strategy. Finally, a state machine manages the cell's communication with its neighbors.

It is important to consider the complication introduced by potentially unbounded fanout in the design netlist we wish to place. Since it is not feasible to allow enough memory for very large fanout at every cell, we need to limit the amount of fanout considered by the algorithm. Experimentally, we verified that the effect of truncating the list of connected cells to 12 did not significantly degrade result quality. We arrived at the number 12 by reserving four slots for LUT inputs (since most architectures are based on 4-input LUTs), realizing that the average fanout would consequently be about four and then allowing a factor 2x to assure only cells with unusually high fanout would be truncated. A more thorough solution might be to configure the synthesis tools to generate LUT mappings with bounded fanout to fit into the allotted memory on the PE's.

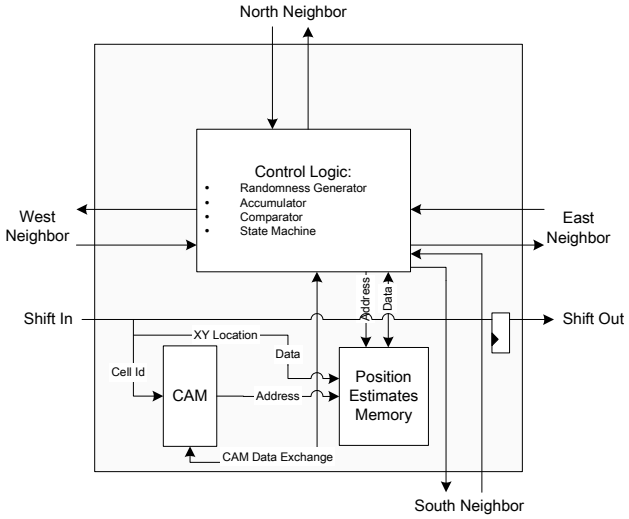

Figure 2. Internal structure of a processing element.

4.3 Algorithm Pseudocode

The pseudocode for the optimization algorithm naturally follows:

4.3.1 High-Level Pseudocode

```
randomly place the design into the PE array
for interval in 0 to TMAX
  for each node PE do in parallel
      loop NUMBEROFCELLS times do
          PE.LOADPOSITIONCHAIN();
          UPDATE PE.connectedCell.positions;
          PE.SHIFTOUTCURRENTPOSITION();
      loop SWAPSPERINTERVAL times do
       for four phases do
          PE.SWAPIFAPPROPRIATE();
return the placement stored in the PE array
```

The four phases ensure that a node will negotiate swaps with its four nearest neighbors on alternating cycles. The next subsection explains the details of this.

4.3.2 Swap Phases

We guarantee both that all four possible swap directions will be considered and that every swap will be agreed to by exactly the two cells involved using a phase scheme. This is done by pairing suitably complementary directions between odd and even cell locations in both dimensions. Boundary cases are easily handled with special cells that never swap off the side of the array. Figure 3 shows the swap phase concept graphically.

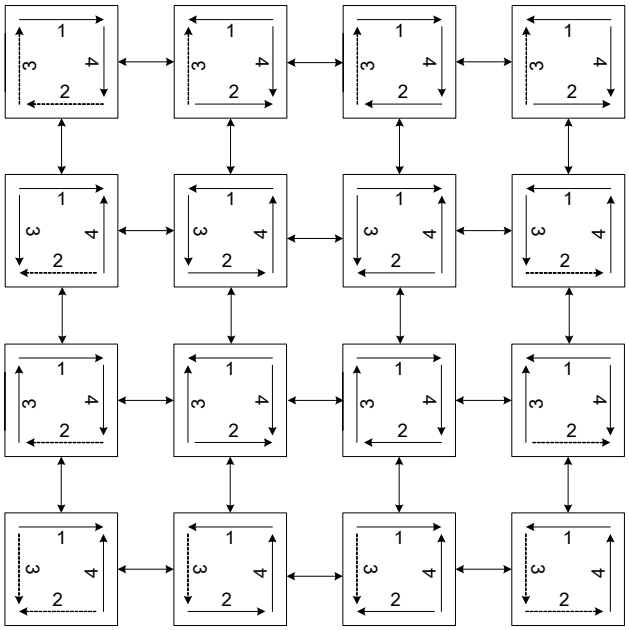

Figure 3. Phase scheme assures that PE's consider swaps in pairs of two. The numbers represent which direction the PE will negotiate at a given phase. Boundary conditions are handled by not actually doing any swap at all.

4.3.3 The swapIfAppropriate Function

The `swapIfAppropriate` function determines what criteria the algorithm optimizes. A purely greedy `swapIfAppropriate` function would be:

```
if hypotheticalCost < currentCost then
   return TRUE;
else
   return FALSE;
```

After some experimentation, we found that purely greedy functions land the algorithm very quickly in a local minimum. We settled on a stochastic `swapIfAppropriate` function, which provides the search behavior of simulated annealing.

Our `swapIfAppropriate` function is:

```
if P undefined then
   P ← 1;
P ← P − 1/(4×TOTALINTERVALSTORUN)
if RANDOM() < P then
   return TRUE;
else if hypotheticalCost <
        currentCost then
   return TRUE;
else
   return FALSE;
```

The local contribution to the cost metric is the sum of the estimated linear Manhattan wirelength to each of the fanin and fanout nodes referenced in the CAM. Readers will notice that this `swapIfAppropriate` differs from a traditional simulated annealing in that the magnitude of the change that would occur from a swap is not considered. Furthermore, linearly varying P is not what a traditional simulated annealing algorithm would do; it is merely a simple approach to use in a preliminary hardware implementation. Later in this section, we argue based on random walks that approaches both using and not using delta cost magnitude provide qualitatively similar explorations of the design space.

4.4 IO Placement

A final issue is how to handle IO placement. Our initial approach, and the one on which we base our results, is to perform a systolic placement on the LUTs themselves and then use a simple greedy algorithm to place the IO pads. Our architecture has some number of IO pads ("slots") along each side of the chip at the ends of every row and column of LUTs. We place each IO that will fit into a slot closest to the LUT to which it is connected. For example, a LUT located at (3,6) on a 10 × 10 array would have its output placed on the left side of the chip six places from the bottom, if a space were available at that group of pins. Then we iteratively increase the amount of error from this "ideal" position until all IOs have been placed. If our LUT at (3,6) could not have its output placed on the left at position six, the next iteration through the loop would attempt to place it on the left side at either position five or seven.

We make no claim this algorithm is ideal, merely that it is simple to implement and yields reasonable results. Since our designs are relatively small, the computational overhead of performing the IO placement with a sequential processor is reasonable.

Another approach is to place the IO pads along with the LUTs. Then, as the cooling schedule progresses, give the pads a synthetic force towards the side of the chip to which they are closest. Finally, use a greedy algorithm to assign them to IO slots.

4.5 Mathematical Analysis

A traditional, design automation formulation of simulated annealing works as follows [8]:

```
T ← TEMPINIT
while (T>TFINAL)  do
     while (useful to cool at T)
        do
          s1 ← random location;
          s2 ← random location;
          cost ← DELTA(s1,s2);
          if (cost<0) then
             SWAP(s1, s2);
          else
             if (EXP(-cost/T) > RANDOM()) then
                SWAP(s1, s2);
     T=NEXT(T); // make smaller
```

The key features of this algorithm:

- allows non-greedy moves to avoid becoming trapped in local minima
- the acceptance probability for non-minimizing moves decreases exponentially with increasing cost

- starts out allowing almost any move ($T_{\to\infty}$ → accept all swaps)
- as T decreases, less likely to accept moves ($T_{\to 0}$ → accept no non-minimizing swaps)
- if we stay at a temperature long enough, all elements should, with high probability, be within a temperature defined cost radius around the optimum location
- as the algorithm progresses T decreases, shrinking the cost radius from the optimal location

The non-decreasing cost moves allow the algorithm to move the design through a series of non-minimizing configurations in its search for global minima.

In this case, our cost function for a swap will be:

$$\begin{aligned} cost = & \sum_{i \in all\ ios\ for\ s1} \left| L(s1.io(i)) - L(s2) \right| \\ & + \sum_{i \in all\ ios\ for\ s2} \left| L(s2.io(i)) - L(s1) \right| \\ & - \sum_{i \in all\ ios\ for\ s1} \left| L(s1.io(i)) - L(s1) \right| \\ & - \sum_{i \in all\ ios\ for\ s2} \left| L(s2.io(i)) - L(s2) \right| \end{aligned}$$

Where $L(s1)$ is the current location of element $s1$ and $s1$.io(i) is the ith element connected to $s1$. We can see this is separable into two parts – the delta cost for moving $s1$ to $s2$ and the delta cost for moving $s2$ to $s1$. (*N.B.* This introduces an inaccuracy if $s1$ or $s2$ drives a pin on the other. The worst consequence possible is a small, local oscillation in the system).

$$\begin{aligned} cost = & \ cost(s1 \text{ move to } L(s2)) \\ & + cost(s2 \text{ move to } L(s1)) \end{aligned}$$

$$\begin{aligned} cost(si \text{ move to } loc) = & \sum_{k \in all\ ios\ for\ si} \left| L(si.io(k)) - loc \right| \\ & - \sum_{k \in all\ ios\ for\ si} \left| L(si.io(k)) - L(si) \right| \end{aligned}$$

It is, therefore, useful to simplify this and consider what happens for the case of moving $s1$ to $L(s2)$ (which exactly corresponds to the case where location $s2$ is empty so its delta cost is 0). Figure 4 plots $e^{-(cost/T)}$, the probability that a node is moved to a distance *cost* from its cost minimizing position.

In our implementation, we want to exploit fast, high-bandwidth local communication. Consequently, we force all swaps to be nearest neighbor swaps. We manage to achieve the same aggregate behavior as sequential simulated annealing using our probabilistic swapping scheme.

To see this, it is useful, again, to focus on the probability locus of a single cell. Assuming for the moment that the cell starts in a cost minimizing position, let us look at where the cell is likely to be after a number of these swaps. Each swap can move away from the minimizing position with probability P_s. After N swaps, the likelihood that our cell is distance d away from the cost minimizing position is:

$$P(d) = \binom{2m+d}{m+d} P_s^{\,m+d} (1-P_s)^m$$

That is, in order to be distance d away, we need to make d more non-minimizing moves than cost minimizing moves. There are $N=2m+d$ choose $m+d$ different ways we can make $m+d$ non-minimizing moves. (To be concrete consider N=3 and d=2; There are 3 choose 2 = 3 ways to make 2 non-minimizing steps and 1 minimizing step. The probability of making these steps is $P_s \times P_s \times (1\text{-}P_s)$ for an aggregate probability of $3 \times P_s^2 \times (1\text{-}P_s)$.) Note here that m is a free variable. As m increases, we have a higher probability of actually finding the cost minimizing position, so that the assumption that we start in a cost minimizing position becomes more likely to be satisfied. In addition, as m increases, we get law of large numbers effects reducing the variance and making the series of discrete swaps converge to this continuous distribution. As shown in Figure 5, the probability locus that this generates has the same characteristics as the traditional simulated annealing exponential cost function (See Figure 4). P_s plays an analogous role to temperature (T). As $P_s \to 1$, all swaps are accepted (compare $T \to \infty$); as $P_s \to 0$, only cost-minimizing moves are accepted (compare $T \to 0$).

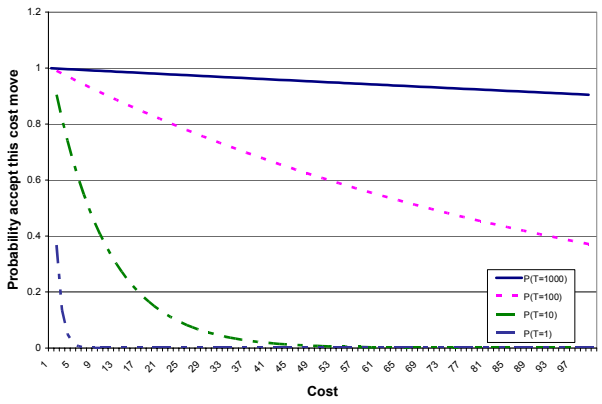

Figure 4. Probability that a node moves to a distance *cost* from its current position for a traditional simulated annealer.

In both cases, there may not be a single cost minimizing location, so the location radius surrounds a cost-minimizing equipotential region rather than a single point.

If the cost function is the same as the distance function, then these two formulations give identical probability loci for suitably corresponding values of P_s and T. Strictly speaking the cost function is not the same shape as the distance function, so the exact, detailed shape of the probability locus will differ. What is important, however, is that both formulations allow the component to explore the radius around the cost-minimizing position with an exponentially decreasing probability as we get away from the minimum and with a drop-off rate that can be incrementally tightened over the course of the algorithm. Our local swap formulation is closer to real, physical annealing than the traditional, design automation formulation since real, micro-scale particles will make local moves in order to explore the state space.

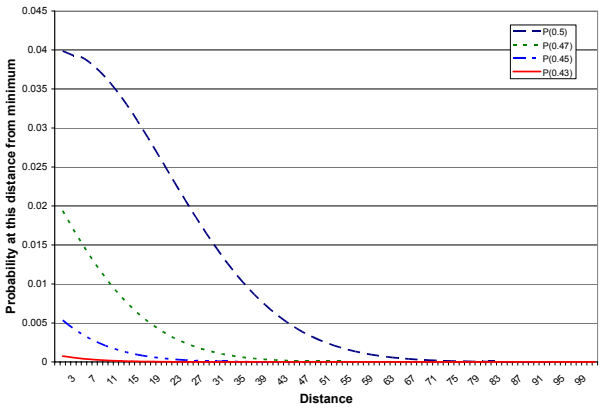

Figure 5. Probability that a node will be a given distance from the greedy, local cost-minimizing position for the local swap annealler.

5. BENCHMARKING RESULTS

We employ a two-fold approach to benchmarking the systolic placer. First, we demonstrate that it is tunable to provide result quality that is close to traditional simulated annealing when optimizing total linear wirelength. Having chosen appropriate constants for the algorithm, look at the channel width which this achieves and argue that the systolic placer offers a new point on placement technology's time-quality envelope.

5.1 Methodology

For the purpose of experimentation, we implemented both a traditional, linear wirelength annealer (with a cooling schedule similar to the one used by VPR) and a simulation of the systolic algorithm in Java, working within the framework offered by a simple mesh-connected FPGA structure. We compared placement quality results *as measured by total estimated linear wirelength* between the two for the well-known Toronto20 benchmark suite used for the "FPGA Place and Route Challenge" [12, 13]. We experimentally explored the variables of SWAPSPERINTERVAL and number of intervals we should allow the algorithm to run before declaring that the algorithm is effective for a range of design sizes.

Measuring placement quality solely in terms of this rough metric is not a perfect predictor of the end metric (*e.g.* channel width required to route, critical path delay). Therefore to gain insight into the final quality our designs, we ran the placements generated with the systolic placer through Versatile Place and Route (VPR) [14] with the `-route_only` and `-fast` option and compared the results to the case where VPR handles both the placement and routing with the `-fast` option. According to the VPR manual [15]:

> -fast: Sets the various placer and router parameters so that a circuit will be placed and routed more quickly, at the cost of some (~10 – 15 %) degradation in quality.

Our own comparison of `-fast` option with the default VPR behavior for the Toronto20 suite found that runtimes were 9x faster and the channel utilization was 27% worse on average, while critical path delays were 4% better when performing both the place and route steps. Our use of the `-fast` option explains why our reported VPR results are worse than those which are published on the FPGA Place and Route Challenge's webpage.

Since our results trade quality for decreased runtime, we believe that it is appropriate to compare them to the fastest generally available software algorithm.

5.2 Intervals Required Exploration

In order to tune the algorithm for the benchmark set, we begin by finding an appropriate number of cooling steps. In order to do this, we make the perfect information assumption and take four swaps (stepping through all the phases). Figure 6 shows that after going through 400 cooling steps, the improvement provided is incremental and reaches quality nearly as good as that of the traditional annealing algorithm for several designs in the benchmark suite.

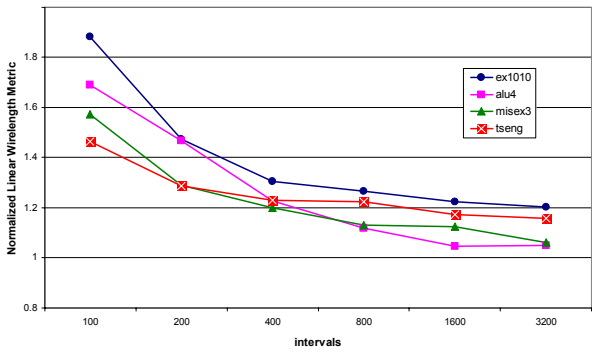

Figure 6. Exploration of intervals to run for several netlists. Result quality is normalized to the quality from a traditional annealing algorithm.

5.3 swapsPerInterval Exploration

After determining that 400 intervals will give reasonable results for the designs in the Toronto20 suite, we move on to adjust the SWAPSPERINTERVAL to see if further quality improvement is possible at the chosen number of intervals. To increase the number of swaps performed, it is preferable to increase the SWAPSPERINTERVAL rather the number of intervals, because each additional interval involves *O(N)* clock cycles to update the state, where *N* is the size of the placement. We hypothesize that performing $O(\sqrt{N})$ swaps at every interval is logical since there is a maximum distance of $2\sqrt{N}$ cells between a LUT's position at the outset of the interval and its "ideal" position in the placement at that time.

Figure 7 shows, for each member of the Toronto20 suite, the relative result quality against the sequential simulated annealing program at several SWAPSPERINTERVAL as a ratio of total estimated linear wirelengths.

The chart indicates staleness doesn't become a meaningful problem at the SWAPSPERINTERVAL we considered. It appears that a SWAPSPERINTERVAL value of $.08\sqrt{N}$ should be reasonable for all of the netlists.

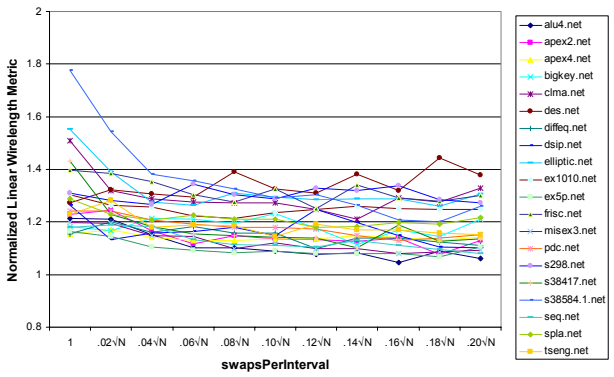

Figure 7. Analysis of swapsPerInterval vs. quality relative to sequential simulated annealing.

5.4 Spatial Annealing Quality and Scaling

Figure 8 summarizes the quality achieved by our systolic placer using a SWAPSPERINTERVAL of $.08\sqrt{N}$ and running for 400 intervals. This shows that we are able to optimize wirelengths within 25% of the software annealer for many of the designs. It further suggests that the algorithm does not deteriorate too badly for large designs

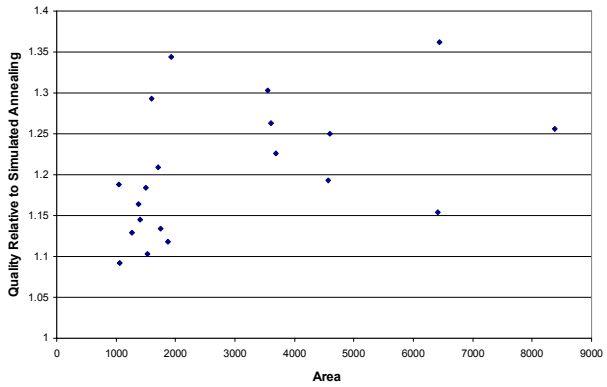

Figure 8. Exploration of effect of design size on relative placement quality.

5.5 Comparison Against VPR Results

In order to look ahead and understand our placements' quality against a fully optimized software placer, we ran the placements generated by our tool (using 400 intervals and a SWAPSPERINTERVAL of $.08\sqrt{N}$) through VPR's router with its `-route_only`, `-fast`, and `-route_algorithm breadth_first` options. We report the channel requirements for three passes through the flow. The architecture is an array of simple 4-LUTs each paired with one output flip-flop. Note that VPR is optimizing directly for channel width, whereas our current systolic placer is only optimizing the linear wirelength metric; as such, we believe, much of the quality loss is due to the simpler cost function rather than from the systolic placement algorithm.

We also ran the benchmarks through the full VPR flow with the `-fast` and `-place_algorithm bounding_box` option, having instrumented the tool to give us the amount of time spent in its placement routine. We compiled VPR with GCC version 3.04 and an optimization setting of `-O3`. Our workstation was an unloaded machine with 2 GB of RAM and a 2.2 GHz Intel Xeon processor with 512 KB of cache. Assuming an FPGA clock rate of 100 MHz and 150 clock cycles to be equivalent to one of the algorithm's steppings through the four phases (we show this to be reasonable in Section 6), we compute an approximate runtime for the spatial implementation. Note that the hardware approach would therefore require approximately 4.2 ms to place the largest design, clma. We compare this time against the amount of processor time required by VPR. This gives an estimation of the speedup offered by our approach that is very conservative for several reasons:

- We are using 400 intervals when some of the designs could be placed over a shorter cooling cycle as shown in Figure 6.
- We expect our FPGA implementation can be optimized substantially both to increase the clock rate and to reduce the number of clock cycles required on every interval.

Table 1 shows the results of our comparison. Clearly, the systolic placer's results are inferior to VPR's, but for some of the designs, the results would likely be acceptable and perhaps preferable given the amount of speedup involved. Note that our systolic scheme optimizing a simple wirelength model yields placements that are about 35% worse than VPR's when measured in terms of minimum channel width required to route. If we take the best of three runs, an average penalty of 29% is attainable. Both placements achieve roughly the same critical path delay; however, neither algorithm is directly attempting to minimize delay. We believe this is substantially faster than a software placer that achieves the same quality level and that we will be able to improve the quality of this approach to compete more seriously against VPR by improving the cost function.

5.6 Speedup Justification

It is worth pointing out that the speedup a hardware-assisted, systolic approach like ours will give can be much greater than that of simply employing a system with a very large number of traditional processors tied to a single memory. This is for several fundamental reasons:

- Local memories and nearest neighbor interconnect allow for extremely high system bandwidth. Instead of requiring massive amounts of expensive shared memory bandwidth, we rely on cheap systolic bandwidth between nearest neighbors.
- There is no added overhead caused by cache thrashing on processors manipulating large data structures or synchronization overhead associated with maintaining cache coherency between processors.
- Specialized datapaths build directly the computation required for this algorithm, providing the right level and structure for parallelism within the local swap calculation and avoiding the overhead associated with structures that are more general.

6. HARDWARE IMPLEMENTATION

In order to understand the feasibility of employing our concept for a systolic placer in practical systems, we are developing an FPGA implementation.

6.1 Preliminary Design Completed

We have created an implementation of the basic systolic placer cell in VHDL and targeted the Xilinx Virtex2 FPGA. Synthesizing the design with Synplify Pro 7.2 and placing and routing the netlist with the Xilinx ISE 5.1i toolset have led us to believe that a design, which requires about 400 LUTs for each processing element and runs at over 100 MHz, is attainable. We were able to use special Virtex "SRL16E" primitives to create reasonably-sized distributed memories and CAM's within the cell [16]. We hold a "shadow" copy of the CAM contents in a RAM to reduce the amount of data that needs to pass between cells. This gives a 3x reduction in swap time at the expense of a modest amount of device resources. As a source of pseudo-random numbers, we employ linear feedback shift registers initialized on power up to random values (i.e. not the same from PE to PE) [17]. A smaller, faster design is likely achievable with further refinement.

We arrived at 150 cycles required for each iteration through the swap phases by the following calculation:

Clock Cycles		Cell Operations
4×		For each direction consider swaps
	5	Compute total current and hypothetical costs with neighbor
	1	Decide whether or not to swap with neighbor
	12	Swap or don't swap the RAM, computing current and hypothetical costs for the next direction
	18	Swap or don't swap the CAM
134	Cycles total	

This only adds up to 134 cycles per iteration; but we round up to 150 because we expect further pipelining to increase this number slightly. We do not include the amount of time to read in the initial placement or output the final placement.

7. APPLICATIONS

We are also examining several practical applications for the technology. Reconfigurable computing is a paradigm that is on the horizon and will need fast placement to enable key capabilities. Logic emulations systems are constrained by long placement runtimes today.

7.1 Application of Hardware Solution to Reconfigurable Computing

In SCORE [18, 19], designs are segmented into pages at netlist generation time (which is often compile time). These page graphs, which may be of arbitrary size, are then scheduled onto a particular device at runtime. This yields a two-level hierarchy, which makes dynamic, run-time configuration more manageable. The fixed-size compute pages become the atomic unit of placement and reconfiguration at runtime. As long as a physical SCORE compute page is larger than one of our placement engine processing elements (~400 LUTs as described in the previous section), a SCORE device is capable of performing *inter*-page placement on itself. That is, we can directly configure the physical SCORE device to be a placement engine that is exactly large enough to place the set of pages that can run on the physical device in a single SCORE timeslice. This works for any number of SCORE pages; the critical parameter is simply that each SCORE page be powerful enough to implement the placement engine's processing element. This adds no additional hardware to the SCORE device; it will simply require a discipline where the runtime system reserves configuration memory space to store the configuration of the placement processing element. Each time a new set of virtual pages requires placement, the runtime system will direct the device to switch to the placement configuration and perform the placement.

Further, in applications where we do want to generate new page configurations at runtime, we can use the whole SCORE device to perform one or more *intra*-page placements as well. This works directly when the SCORE device has equal or greater physical compute pages than it has LUTs within a physical compute page. In this case, we configure a suitable subset of the SCORE device with the placement processing elements and use that to compute the intra-page placement for the LUTs within the compute page. Very large SCORE devices that have more physical compute pages than there are LUTs within a compute page may be able to place multiple compute pages simultaneously or place a compute page while operations continue on other portions of the array. Note that it only takes devices with 400×400=160,000 LUTs to be large enough to place a 400 LUT compute page. Xilinx's XC2VP125 (the largest documented Virtex-II Pro part) has over 100,000 LUTs [20], suggesting devices of suitable size to do single, intra-page placement are not far off in the future.

7.2 Application to Logic Emulation Systems

In order to accelerate VHDL and Verilog RTL simulations of large ASIC designs, several companies (*e.g.* Quickturn, IKOS) market arrays of FPGAs used for rapid prototyping. These accelerators attempt to allow simulations, which would take hours, days, or years using conventional uniprocessors, to run in seconds, minutes, or hours. Unfortunately, with software placement tools, it takes hours to place each of the FPGAs in these systems. Since these systems typically employ hundreds to thousands of FPGAs, it would take days to weeks for a single workstation simple to place the devices in order to run the accelerated simulation! This long placement time reduces the benefit and utility of these accelerators. As a partial mitigation, emulation vendors typically ship dozens of workstation-class computers along with a single accelerator box in order to reduce the FPGA place and route time. In contrast, our approach shows how one could use the FPGAs in the emulation engine itself to perform more rapidly the *intra*-FPGA placement. That is, as long as the emulation engine has 400 or more FPGAs, we can use that collection of FPGAs to place each single FPGA quickly.

8. FUTURE DIRECTIONS

This approach to simulated-annealing is very new and requires substantial theoretical and experimental work before being practical for applications. Thus, we anticipate significant future work.

8.1 Hardware Implementation and Testing

The most obvious direction our work will take is to get the hardware implementation optimized and working on a real FPGA. It is further important to chain together our fast placement solution with a hardware-assisted routing solution such as [3]. Such a platform could demonstrate the feasibility of runtime placement and routing, making dynamically

reconfigurable computing systems more practical than previously shown.

We would also like to demonstrate that our design can be mapped over multiple FPGAs to handle designs of arbitrary complexity. This would be economical for hardware emulation platforms, which already include large numbers of devices.

8.2 Open Questions

The most important open question is how much can be done to improve the quality of the placements generated by our algorithm.

8.2.1 Improved Performance

We believe that there are several obvious directions we can explore to improve the performance of this algorithm. Most obviously is that we should develop an adaptive cooling schedule. Readers familiar with simulated annealing know that most of the change in an objective function's value occurs during a relatively small band of temperatures. Further, it is also clear that some designs require substantially less cooling time than others do. It should be reasonable to move to a termination condition that halts the algorithm as soon as the placement cost is stable. This will speed up many placements. Further, more sophisticated position chain topologies may reduce the overhead for state updating. As pointed out in Subsection 5.4, for a wide variety of designs, we anticipate very good scalability in runtime as design size varies. We are hoping to experiment with the algorithm on designs larger than those available in the Toronto20 suite, the largest of which consumes only about 10% of a contemporary large device.

8.2.2 Cost Functions

While the linear wirelength cost function offers a simple way to demonstrate the concept of massively parallel annealing, we expect to get better results with a cost function that actively minimizes congestion or critical path delay. Recent work suggests spatial approaches to both problems may be practical [21, 22]. We hope to explore the practicality of implementing other cost functions, especially bounding-box, spatially. A bounding-box metric would make comparisons with VPR's placer more direct.

It would also be interesting to examine how other applications of simulated annealing could map to a spatial structure. The scheduling problem stands out as one logical candidate.

9. SUMMARY

We have shown substantial speedups to simulated annealing solutions to the placement problem using spatial computation hardware. Our chief contributions to the literature are:

- A formulation of a local-swap variant of simulated annealing and a demonstration that it is suitable for FPGA placement where the cost metric is linear wirelength
- The design of a direct spatial analog of the placement solution space, which performs this version of simulated annealing using only local communications
- Experimental demonstration of substantial speedups over state-of-the-art software placers for the placement of moderate-sized designs

Table 1-Comparison of Systolic placements to VPR Placements

Netlist	**Size *(LUTs)***	**VPR** `-fast -bounding box`		**Systolic Placer Channel Width *(3 runs)*** *(% of VPR result)*			***Clock Cycles***	***Speedup*** *Assuming 100 MHz and 150 cycles to consider four swaps*
		Channel Width	*Runtime*	*Minimum*	*Maximum*	*Average*		
alu4	1522	11	4.455	12 (109%)	13 (118%)	12.33 (112%)	9.27E+05	481
apex2	1878	13	6.57	14 (108%)	15 (115%)	14.33 (110%)	1.07E+06	615
apex4	1262	14	3.7	16 (114%)	16 (114%)	16.00 (114%)	7.35E+05	503
bigkey	1707	8	6.86	11 (138%)	12 (150%)	11.14 (139%)	1.54E+06	444
clma	8383	14	110.23	23 (164%)	28 (200%)	24.33 (174%)	4.16E+06	2649
des	1591	9	6.37	13 (144%)	16 (178%)	15.00 (167%)	2.05E+06	310
diffeq	1497	9	4.9	13 (144%)	15 (167%)	14.33 (159%)	8.94E+05	548
dsip	1370	7	4.99	9 (129%)	10 (143%)	9.50 (136%)	1.55E+06	323
elliptic	3604	13	22.04	21 (162%)	23 (177%)	21.67 (167%)	1.88E+06	1170
ex1010	4598	12	34.65	14 (117%)	17 (142%)	16.00 (133%)	2.33E+06	1490
ex5p	1064	15	3.41	16 (107%)	17 (113%)	16.67 (111%)	6.48E+05	526
frisc	3556	15	21.94	21 (140%)	22 (147%)	21.33 (142%)	1.83E+06	1197
misex3	1397	13	3.99	14 (108%)	14 (108%)	14.00 (108%)	8.63E+05	463
pdc	4575	20	32.77	24 (120%)	26 (130%)	25.00 (125%)	2.33E+06	1408
s298	1931	9	5.88	17 (189%)	19 (211%)	18.33 (204%)	1.07E+06	550
s38417	6406	10	71.54	12 (120%)	14 (140%)	13.00 (130%)	3.21E+06	2227
s38584.1	6447	11	78.07	13 (118%)	15 (136%)	13.75 (125%)	3.25E+06	2404
seq	1750	13	6.11	15 (115%)	17 (131%)	15.67 (121%)	9.98E+05	612
spla	3690	17	22.87	22 (129%)	24 (141%)	23.00 (135%)	1.89E+06	1213
tseng	1047	8	3.19	11 (138%)	12 (150%)	11.33 (142%)	6.50E+05	490
Total Channels		241		311 (129%)	345 (143%)	326.73 (136%)		

ACKNOWLEDGMENT

This research was funded in part by the DARPA Moletronics program under grant ONR N00014-01-0651 and by the NSF CAREER program under grant CCR-0133102.

REFERENCES

[1] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger, "Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures," *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pp. 248--259, 2000. <http://www.cs.utexas.edu/users/cart/publications/isca00.pdf>.

[2] A. DeHon, R. Huang, and J. Wawrzynek, "Hardware-Assisted Fast Routing," presented at IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), Napa, CA, 2002. <http://www.cs.caltech.edu/research/ic/abstracts/fastroute_fccm2002.html>.

[3] A. DeHon, R. Huang, and J. Wawrzynek, "Stochastic Spatial Routing for Hypergraphs, Trees, and Meshes," presented at Eleventh ACM International Symposium on Field-Programmable Gate Arrays, Monterey, CA, 2003. <http://www.cs.caltech.edu/research/ic/abstracts/fastroute_fpga2003.html>.

[4] C. Mulpuri and S. Hauck, "Runtime and quality tradeoffs in FPGA placement and routing," *Proceedings of the Ninth International Symposium on Field programmable Gate Arrays*, pp. 29--36, 2001. <http://www.ee.washington.edu/faculty/hauck/publications/RuntimeTradeoffs.pdf>.

[5] Y. Sankar and J. Rose, "Trading quality for compile time: ultra-fast placement for FPGAs," *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, pp. 157--166, 1999. <http://www.eecg.toronto.edu/~jayar/pubs/sankar/fpga99sankar.pdf>.

[6] D. Mitra, F. Romeo, and A. Sangiovanni-Vincentelli, "Convergence and Finite-Time Behavior of Simulated Annealing," *Advances in Applied Probability*, vol. 18, pp. 747-771, 1986.

[7] S. Goto, "An Efficient Algorithm for the Two-Dimensional Placement Problem in Electrical Circuit Design," *IEEE Transactions on Circuits and Systems*, vol. CAS-28, pp. 12-18, 1981.

[8] K. Shahookar and P. Mazumder, "VLSI cell placement techniques," *ACM Computing Surveys (CSUR)*, vol. 23, pp. 143-220, 1991.

[9] P. Banerjee, *Parallel Algorithms for VLSI Computer-Aided Design*, Chapter 3. Englewood Cliffs, NJ: PTR Prentice Hall, 1994.

[10] E. I. Horvath, R. Shankar, and A. S. Pandya, "A Parallel Force Directed Standard Cell Placement Algorithm." Technical Report. Dept. Computer Science, Florida Atlantic University, Boca Raton, FL., 1992.

[11] M. Haldar, A. Nayak, A. Choudhary, and P. Banerjee, "Parallel algorithms for FPGA placement," *Proceedings of the tenth Great Lakes Symposium on VLSI*, pp. 86--94, 2000. <http://www.ece.nwu.edu/cpdc/Match/Pubs/glvlsi2000.malay.ps>.

[12] V. Betz, "The 'FPGA Place-and-Route Challenge'." Toronto, 2001. <http://www.eecg.toronto.edu/~vaughn/challenge/challenge.html>.

[13] V. Betz, J. Rose, and A. Marquardt, *Architecture and CAD for deep-submicron FPGAs*. Boston: Kluwer Academic Publishers, 1999.

[14] V. Betz and J. Rose, "VPR: A New Packing, Placement and Routing Tool for FPGA Research," presented at International Workshop on Field Programmable Logic and Applications, London, 1997. <http://www.eecg.toronto.edu/~vaughn/papers/fpl97.pdf>.

[15] V. Betz, *VPR and T-VPack User's Manual (Version 4.30)*, 2000. <http://www.eecg.toronto.edu/~vaughn/vpr/vpr.html>.

[16] J.-L. Brelet and B. New, "XAPP203: Designing Flexible, Fast CAMs with Virtex Family FPGAs," Xilinx Application Note, 1999. <http://www.xilinx.com/xapp/xapp203.pdf>.

[17] P. Alfke, "XAPP052: Efficient Shift Registers, LFSR Counters, and Long Pseudo-Random Sequence Generators," Xilinx Application Note, 1996. <http://www.xilinx.com/xapp/xapp203.pdf>.

[18] E. Caspi, M. Chu, R. Huang, J. Yeh, J. Wawrzynek, and A. DeHon, "Stream Computations Organized for Reconfigurable Execution (SCORE): Extended Abstract," presented at Conference on Field Programmable Logic and Applications, Villach, Austria, 2000. <http://www.cs.berkeley.edu/projects/brass/documents/score_fpl2000.html>.

[19] E. Caspi, M. Chu, R. Huang, J. Yeh, Y. Markovskiy, J. Wawrzynek, and A. DeHon, "Stream Computations Organized for Reconfigurable Execution (SCORE): Introduction and Tutorial," 2000. <http://www.cs.berkeley.edu/projects/brass/documents/score_tutorial.html>.

[20] Xilinx, "Virtex-II 1.5V Field Programmable Gate Arrays: Data Sheet." San Jose, CA, 2002. <http://www.xilinx.com/partinfo/ds031.pdf>.

[21] B. Hu and M. Marek-Sadowska, "Congestion Minimization During Placement Without Estimation," presented at IEEE/ACM Internation Conference on Computer Aided Design, San Jose, CA, 2002.

[22] T. Kong, "A Novel Net Weighting Algorithm for Timing-Driven Placement," presented at IEEE/ACM International Conference on Computer Aided Design, San Jose, CA, 2002.