Parallel Placement for Field-Programmable Gate Arrays

Pak K. Chan
Department of Computer Engineering
University of California
Santa Cruz, California 95064 USA
pak@cse.ucsc.edu

Martine D.F. Schlag
Department of Computer Engineering
University of California
Santa Cruz, California 95064 USA
martine@cse.ucsc.edu

ABSTRACT

Placement and routing are the most time-consuming processes in automatically synthesizing and configuring circuits for field-programmable gate arrays (FPGAs). In this paper, we use the negotiation-based paradigm to parallelize placement. Our new FPGA placer, NAP (Negotiated Analytical Placement), uses an analytical technique for coarse placement and the negotiation paradigm for detailed placement. We describe the serial algorithm and report results. We also report findings related to parallelizing NAP under a multicast networking and multi-threaded operating system environment; the parallel placer is tolerant to multicast packet loss as well as out-of-order packet delivery. Our parallel placer exhibits little performance degradation while attaining speedups of 2 using 3 processors.

Categories and Subject Descriptors

B.7.1 [**Integrated Circuits**]: Types and Design Styles—*Gate arrays*; B.7.2 [**Integrated Circuits**]: Design Aids—*Placement and routing*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed programming*

General Terms

Algorithms, Performance, Design

Keywords

Parallel placement, FPGAs, Timing-driven placement, Analytical placement

1. INTRODUCTION

Placement and routing are the most time-consuming processes in automatically synthesizing and configuring circuits for field–programmable gate arrays (FPGAs). Given a circuit represented as a connection of logic blocks, the problem of placement for FPGAs can be stated as that of assigning each logic block to a unique physical resource while achieving a given overall performance. As the capacity of FPGA devices as well as the size of FPGA users' circuits grow, there is a great demand to place designs rapidly. There are several approaches towards this problem. One can sacrifice/trade the quality of the placement for the amount of time it takes to place the circuit [9], or put effort into accelerating a serial placer. The first approach seems to imply a user will know the proper quality/time tradeoff of the placer *a priori*. The authors of [6] apply different techniques for parallelizing simulated annealing to accelerate VPR[2]. These techniques require the parallel placer to run on expensive shared-memory machines (SGI Origin) or specialized distributed memory multiprocessors (IBM-SP2).

In contrast, we approach this problem by designing a placer that can be easily parallelized in a ubiquitous network environment. Our algorithm combines an analytical technique [7, 10] with the negotiation paradigm [4]. In the negotiation paradigm, blocks are allowed to occupy the same location and the cost of overcommitted locations are increased (*history_cost*) until the conflicts are resolved. This technique is key to our parallelization since it allows the algorithm to tolerate inaccuracy in each processor's knowledge of the current positions of the blocks.

We discuss our serial placement algorithm in Section 2, and present some experimental results in Section 2.7. In Section 3, we describe the parallel placement algorithm and its implementation and provide speedup results. We ascertain the quality of the serial placer, and use a network of low cost workstations to run the parallelized placements. Each parallel placer has two threads; one thread is responsible for communication via multi-cast while the other thread is placing the blocks.

2. A NEW PLACEMENT ALGORITHM

In this section we present a placement method (NAP) which is a hybrid of the well known analytical placement technique and the negotiation method in an iterative framework. Analytical placement is used to minimize the weighted quadratic displacement between blocks while the negotiation paradigm is used to resolve the competition of blocks for locations. The negotiation method has been shown to converge for bipartite matching problems such as the problem of assigning blocks to locations [4].

Covering the netlist with trees allows us to calculate the analytical placement efficiently and hence repeatedly. It also provides the means to partition the task for parallel processing. Figure 1 gives a high-level description of the serial algorithm. The input is the netlist of I/O and logic blocks; the output is the assignment of blocks to locations. For simplicity, we first assume that the positions of the I/O blocks are given and fixed, that is, the pads are locked. Section 2.6 describes the case when pad locations are not fixed.

FPGA'03, February 23–25, 2003, Monterey, California, USA.

```
NAP(netlist, FPGA architecture)

    Cover the netlist with trees
    while the placement is not feasible
        for each tree T_i
            Calculate analytical placement V_i of T_i
            for each node n_j in T_i
                find position for n_j in window centered at V_{i,j} minimizing
                    clone_cost + history_cost + shared_cost + Iteration(mod 2)Δwirelength
            endfor
            Update delays and criticalities
        endfor
        Update history costs
    endwhile
end
```

Figure 1: Placement algorithm.

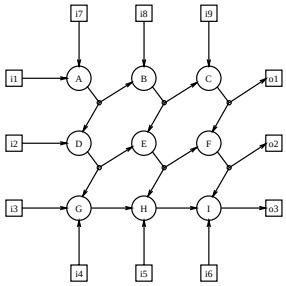

Figure 2: Netlist example.

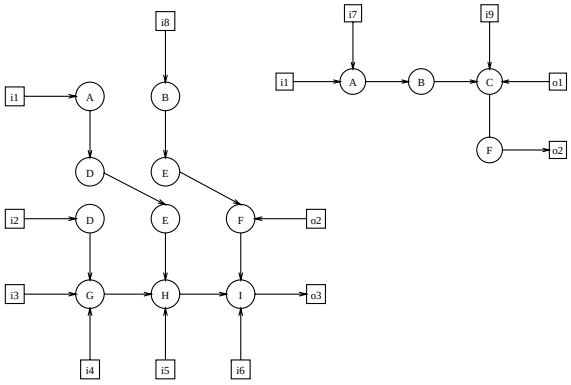

Figure 3: Tree cover of netlist example.

2.1 Covering with trees

The first step is to cover the netlist with trees. The root of the tree is an output I/O block, the leaves are I/O blocks and the internal nodes are logic blocks. The tree covering must ensure that each I/O block, logic block and source-to-sink connection appears in at least one tree. This may require that blocks appear multiple times, either as nodes in the same tree or separate trees. For this reason we refer to the tree nodes as *clones* of blocks. Figure 2 shows a netlist corresponding to a 3×3 grid. Its tree cover is shown in Figure 3. The edges are oriented from child to parent in Figure 3 which can be opposite from the signal direction as is the case for the edge from o1 to C in Figure 3.

Design	IOs	logic blocks	edges	clones	trees
alu4	22	1522	5408	6243	8
apex2	41	1878	6692	8384	3
apex4	28	1262	4479	6528	19
des	501	1591	6110	8440	245
ex1010	20	4598	16078	26473	10
ex5p	71	1064	4002	8903	34
misex3	28	1397	4968	6343	43
pdc	56	4575	17193	28988	40
seq	76	1750	6193	7857	33
spla	62	3690	13808	24347	46
g50	200	2500	5100	6823	2

Table 1: Tree sizes.

Each source-to-sink connection (and hence block) must be represented in at least one tree, so that the delay of the circuit can be properly optimized in the placement phase. But since the running time of the placement phase depends on the number of clones in the cover, the algorithm should try to minimize the total number of clones. Minimizing the cover is in itself an NP-hard problem. Since we wish to minimize the size of the cover principally to reduce the running time of the placement phase, the time spent on generating the cover must be balanced against the goal. We use the heuristic approach given in Figure 4.

The netlist is an acyclic graph since latches are decomposed into input and output nodes. Hence the longest paths can be calculated in linear time using the DAG ordering. The first path selected is the minimum longest input-to-output path rather than the maximum; using the maximum often results in a single large tree. After the first path, the trees are created by adjusting the cost of the nodes and edges so that we can find a path from the inputs with the most missing (not yet covered) blocks and edges. Table 1 shows the size and number of the trees in the covers produced for the benchmarks used in our experiments.

2.2 Collapsing the Clones

In addition to the usual constraint that the blocks not overlap, feasibility now also requires that each logic block's clones have been placed in a single location: *collapsed*. To achieve this we calculate the *center of gravity* of the clones of each logic block (if it has more than one clone). The distance from a clone to its block's

```
Cover_with_Trees(netlist)

    Calculate the longest path for each input to any output
    B ← 2* Number of nodes
    T1 ← a minimum longest input to the output path
    while there are missing nodes or edges
        Set cost of missing nodes to −B
        if there are no missing nodes
            Set cost of missing internal edges to −B
        else  Set cost of missing internal edges to the cost of their source node
        Calculate the minimum cost paths from each node to any output
        Traverse a minimum cost path starting from the input with the minimum
            pathcost until either an output or a node with positive path cost is
            reached
        if an output is reached
        then create a new tree containing the traversed path
        else graft the traversed path onto the tree at the final node
    endwhile
    Add any missing external edges onto the trees
end
```

Figure 4: Covering algorithm.

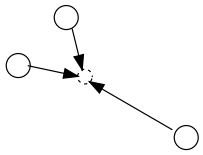

Figure 5: Three clones and their center of gravity: the small dotted circle.

center of gravity is part of the cost function in fine placement (inner **for-loop**) and is also used in the analytical placement as described in Section 2.3. Figure 5 shows three clones and their center of gravity (dotted node). Each clone is encouraged to move towards the center.

To quickly calculate the center and determine if a logic block's clones have collapsed, we maintain two sums for each coordinate and update them incrementally:

$$L_x = \sum_{i=1}^{n} x_i,$$

and

$$S_x = \sum_{i=1}^{n} x_i^2$$

where $x_1, x_2, \ldots, x_n$ are the x-coordinates for the block's clones. L_y and S_y are likewise maintained for the y-coordinates. We calculate the center of gravity by dividing L_x and L_y by the number of clones. Since the variance of the coordinates,

$$\frac{1}{n}\sum_{i=1}^{n}(x_i-\bar{x})^2 = \frac{1}{n}\left(\sum_{i=1}^{n} x_i^2\right) - \bar{x^2} = \frac{1}{n^2}(nS_x - L_x^2)$$

will be 0 only when all the coordinates are identical, we can detect in constant time whether a logic block has collapsed.

2.3 Analytical Placement

We use an analytical placement technique to determine a coarse placement for each tree during each iteration. In principle, the analytical placement problem is to solve the matrix equation:

$$G \cdot \vec{v} = \vec{b}$$

where $\vec{b}$ is the vector of positions of the I/Os, $\vec{v}$ represents positions of the clones of the internal blocks, and G is the edge-weight matrix of the graph. This computation can be interpreted as computing the node voltages of a resistive network with the I/Os as current sources, the edge-weights as conductances, and the positions of the clones as node voltages [5]. Figure 6 shows the calculation for a tree with 5 nodes and 5 I/Os. The conductance of each edge is determined by the criticality of the edge. Because our graph is a tree, $\vec{v}$ can be calculated quickly in linear time by traversing the tree in lieu of a numerical approach.

G also includes an edge for each clone to its center of gravity, which has a fixed position during this calculation and so is treated as an additional current source. The edge to the center of gravity has conductance 0 initially, but its value is increased to help the clones of a logic block collapse to a single location as the algorithm progresses.

This analytical placement computation yields an optimal solution with respect to a weighted quadratic wirelength objective, but the cells may overlap. A negotiation placement paradigm which resolves these overlaps is discussed below.

2.4 Fine Placement

Fine placement of each clone is achieved by using the voltage calculated as described above as the center of a window and selecting a location within the window which minimizes the cost function:

$$\begin{aligned} \mathit{clone\_cost} &+ \mathit{history\_cost} \\ &+ \mathit{shared\_cost} \\ &+ \mathit{Iteration}(\bmod 2)\Delta \mathit{wirelength} \end{aligned}$$

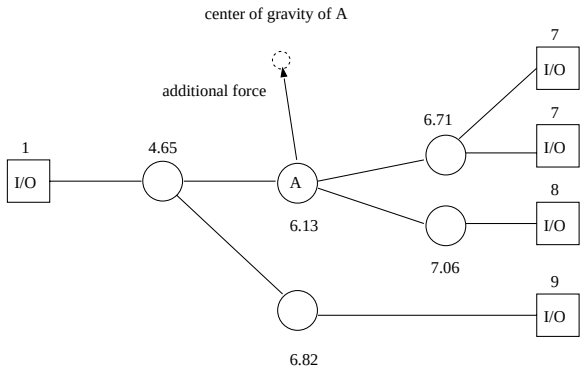

Figure 6: Analytical placement. I/O blocks are current sources. Each edge has resistance of one Ohm. Values on the nodes are voltages.

clone_cost is the distance to the logic block's center of gravity. The *history_cost* of a location is increased at the end of every iteration after which that location is over-committed. The *shared_cost* is the number of other logic blocks that currently occupy that location (have at least one clone present). The weight of the shared and clone components as well as the size of the window are adjusted as the algorithm proceeds to achieve feasibility.

The spans of each net are maintained so that the change in wirelength of the nets can be quickly evaluated. The centers of gravity of the clones are used as the coordinates of the logic blocks for the wirelength calculations.

The delays of all the edges (source-to-sink connections) are updated after each tree has been placed, and their criticalities are updated for use in the analytical placement. The delay of a source-to-sink connection is the sum of the delays associated with a shortest path through the routing architecture; in this case, the sum of the delays for the pips, switches and wire segments.

2.5 Scaling for large array sizes

The algorithm is applied hierarchically by clustering the grid locations into which the clones must be placed. For example, a 60×60 grid using a cluster size of 3 becomes a 20×20 grid with each grid position (slot) capable of holding 9 logic blocks. This improves the efficiency by allowing placement constraints to be resolved at a coarser level. All delay and wirelength calculations use the full coordinate system; the coordinates of the grid position at the center of the slot are used as the position for each clone in that slot. Once feasibility or near feasibility is attained at the coarser level, the current placement is expanded with each grid position inheriting an interpolated version of the history of its slot. For the first few iterations after expansion, clones are constrained to remain within the slot boundaries in order to reestablish an equilibrium.

2.6 Moving I/Os

In the preceding sections we have assumed the positions of the I/O blocks were given and fixed. This might be the case in practice, but when the placement of the I/Os is not constrained, their placement should be determined as part of the placement optimization. In order to calculate the analytical placement described in Section 2.3 the I/O positions must be known.

Initial positions for the I/Os are calculated from the 2^{nd} and 3^{rd} smallest eigenvectors of the Laplacian derived from the netlist [3]. The fanout netmodel was used to convert the netlist to a graph.[1]

[1] A net with f sinks was represented in the graph by f edges each of weight $1/f$.

g50	fixed pads			moving pads		
	x,y wl	delay	time	x,y wl	delay	time
VPR	1.25 1.18	1164	14.23	1.15 1.99	1323	15.0
NAP	1.06 1.07	1128	17.35	1.10 1.16	1278	29.37

Table 3: Results of a 50×50 grid netlist with fixed and moving pads. The optimal solution has wirelength 0.98 0.98 and delay 999.

The two eigenvectors are used to provide a two-dimensional embedding of the circuit. The embedding is first rotated so that the maximum number of outputs are on the right and lower sides (in the range $[-135^o, 45^o]$) to match the direction of the architecture. Each I/O is then projected to the boundary of the array preserving its angle with respect to the the center of the embedding. If this position is occupied, the I/Os on either side (whichever requires less shifting) are shifted to make room.

Inside the placement loop, positions of the I/Os are recalculated every 50^{th} iteration. The position which minimizes delay with respect to the current positions of the logic blocks is determined. If this position is occupied, the position with the least current occupants within a small window is selected. If there is significant overlap after all I/Os have been considered, then the I/Os are *flattened* to remove any overlaps. *Flattening* consists of examining each over committed slot and shifting the I/Os on either side to make room for the excess.

2.7 Performance of NAP

To evaluate our placement algorithm we ran NAP using the architecture and benchmarks from the FPGA Place-and-Route Challenge [1] and compared the results with VPR version 4.3 run with the `path_timing_driven` option. All ten combinational benchmarks were placed by VPR. NAP was run with both fixed pad locations using the pad locations produced by VPR and also without fixed pad locations. For each benchmark the array size is determined by the smallest square array that will accommodate both the logic blocks and I/O pads (`des` is the only design for which the array size is constrained by the number of I/O pads). The placements are validated by a separate program which takes as input the placement output file and also reports the wirelength and delay. Table 2 gives the results in term of wirelength, delay and running time in minutes on a SUN Ultra 5 360Mhz. We also ran VPR and NAP on a synthetic design for which the optimal wirelength and delay is known: a grid of size 50×50 similar to the circuit in Figure 2. Table 3 shows the results of VPR and NAP placing this design on a 50×50 array (100% full).

The results in Tables 2 and 3 are reported in terms of the additive delay model used in NAP. As VPR is optimizing a capacitive delay model we also evaluated the placements using VPR with the `route_only` and `timing_driven` options. Before attempting to route VPR reports the Estimated Critical Path; these results are reported in Table 4 under the heading "Est." The placements were routed with the minimum number of tracks achieved in the FPGA Place-and-Route Challenge [1] and as expected, none of them were successfully routed as these placements were optimized for delay. We also routed each placement with 20%, 50% and 100% additional tracks beyond the minimum [1] and report these results in Table 4. A routing failure is indicated by RF. In Table 5, the results from Table 4 are normalized to the best result achieved within each routing experiment.

Design	Array size	VPR			NAP					
		Moving Pads			Fixed Pads			Moving Pads		
		x,y wl	delay	time	x,y wl	delay	time	x,y wl	delay	time
alu4	40	3.35 2.62	352	11.84	3.42 2.78	318	8.36	3.34 2.90	361	9.23
apex2	44	4.06 3.59	412	22.21	4.12 3.69	376	9.13	3.92 3.98	437	10.19
apex4	36	5.25 3.99	330	13.44	4.86 4.15	342	6.99	4.19 4.81	396	13.80
des	63	4.11 3.90	377	20.58	4.28 4.06	458	6.33	5.21 4.77	369	6.17
ex1010	68	3.76 3.81	821	84.22	3.74 4.03	782	54.51	4.03 4.19	754	67.00
ex5p	33	5.45 5.22	300	7.24	5.37 5.28	312	27.91	5.03 5.73	300	16.70
misex3	38	3.91 3.40	369	10.23	3.83 3.62	303	6.12	3.75 3.70	309	6.08
pdc	68	4.89 5.21	728	90.37	4.90 5.18	609	69.09	4.84 5.23	676	62.61
seq	42	3.81 4.07	339	17.58	3.75 3.90	326	9.78	3.75 3.82	352	11.33
spla	61	4.02 4.02	590	59.77	4.38 4.08	553	52.31	3.91 4.51	562	46.82

Table 2: Results for VPR in timing mode and NAP in terms of wirelength, delay and running time in minutes. NAP was run both with fixed pads (obtained from the VPR placement) and with moving pads starting with the spectral pad placement.

	VPR				NAP							
	Moving Pads				Fixed Pads				Moving Pads			
Design	Est.	+20%	+50%	+100%	Est.	+20%	+50%	+100%	Est.	+20%	+50%	+100%
alu4	78.6	80.4	80.4	80.3	72.3	82.0	76.2	74.4	81.3	88.6	84.6	82.2
apex2	89.8	90.6	90.6	90.6	84.6	87.3	88.4	86.0	97.3	104.3	98.6	99.1
apex4	73.0	RF	88.3	73.0	75.3	RF	77.2	77.2	86.7	RF	96.3	87.9
des	82.6	82.9	82.9	83.5	98.1	98.5	98.4	98.5	80.6	83.5	82.0	82.0
ex1010	171.6	187.0	175.7	176.6	166.4	172.7	167.3	167.3	159.6	RF	165.5	161.7
ex5p	67.0	RF	68.5	68.5	68.8	100.7	70.6	70.6	68.7	RF	69.3	69.0
misex3	81.8	108.9	82.0	82.0	68.7	82.6	70.0	69.4	69.4	108.7	70.6	71.5
pdc	154.7	175.7	156.0	159.0	132.8	181.3	137.5	136.3	144.2	157.1	146.1	144.9
seq	75.3	80.4	76.6	76.6	73.1	105.3	78.4	74.5	77.3	104.1	79.8	81.0
spla	127.8	RF	129.1	128.5	120.0	RF	123.7	122.6	121.8	RF	124.2	125.5

Table 4: Results for VPR in timing mode and NAP in terms of critical path as measured by VPR. The estimated critical path is in 10^{-9} seconds, as well as the routed result with 20%, 50% and 100% more tracks than the minimum [1].

	Est.			+20%			+50%			+100%		
design	VPR	NAPf	NAPm	VPR	NAPf	NAPm	VPR	NAPf	NAPm	VPR	NAPf	NAPm
alu4	1.09	1	1.12	1	1.02	1.1	1.06	1	1.11	1.08	1	1.1
apex2	1.06	1	1.15	1.04	1	1.19	1.02	1	1.12	1.05	1	1.15
apex4	1	1.03	1.19	RF	RF	RF	1.14	1	1.25	1	1.06	1.2
des	1.02	1.22	1	1	1.19	1.01	1.01	1.2	1	1.02	1.2	1
ex1010	1.08	1.04	1	1.08	1	RF	1.06	1.01	1	1.09	1.03	1
ex5p	1	1.03	1.03	RF	1	RF	1	1.03	1.01	1	1.03	1.01
misex3	1.19	1	1.01	1.32	1	1.32	1.17	1	1.01	1.18	1	1.03
pdc	1.16	1	1.09	1.12	1.15	1	1.13	1	1.06	1.17	1	1.06
seq	1.03	1	1.06	1	1.31	1.29	1	1.02	1.04	1.03	1	1.09
spla	1.06	1	1.01	RF	RF	RF	1.04	1	1	1.05	1	1.02

Table 5: Relative results in terms of critical path for VPR, NAP with fixed pins and NAP with moving pins. Each placer's result is divided by the minimum result within the experiment.

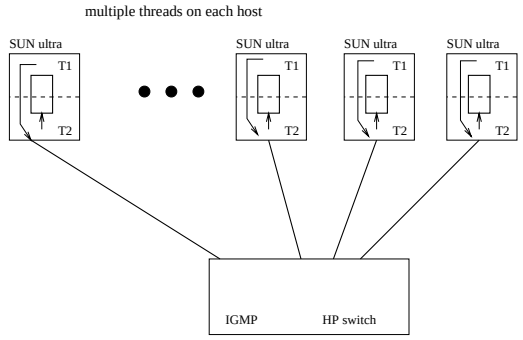

Figure 8: Multi-cast and Multi-threaded parallel processing.

3. PARALLEL VERSION

In the parallel version of our placement algorithm, each processor is assigned a subset of the trees to place. Each processor communicates the positions of the clones in its trees when they are modified. Hence each processor has information about the positions of all of the clones necessary to calculate the positions of the clones in its assigned trees. Figure 7 gives a high-level description of the parallel algorithm.

The trees are allocated by sorting them into decreasing size and allocating them one by one to the processor which currently has the subset of smallest size. This is a static allocation.

Each processor keeps track of the iterations of every processor to avoid getting too far ahead of the others (out of synch). When the number of shared locations and uncollapsed blocks drops below a certain threshold, a designated processor takes over and completes the placement on its own. This is necessary for convergence of the negotiation paradigm [4].

3.1 Communications aspects of P-NAP

In a negotiation-based algorithm, there are no constraints on the number of tenants which can occupy a resource; instead the cost of using a resource with multiple occupants is the mechanism used to resolve resource conflicts. Our experience with our parallel router has revealed that tight synchronization among the processors is not required, as long as the costs are updated eventually and correctly [4].

Available technology to support inter-processor communication exists at the computer network level and board level. Processors communicate through "sockets" at the network level with loose synchronization. Threads programming and corresponding operating system support are widely available to facilitate fine-grain threads to communicate tightly. We exploit both of these opportunities in our placer.

Currently our parallel placer runs on loosely coupled SUN Ultra 5 workstations. Each processor is assigned a subset of the tree cover to place and the processors exchange messages to communicate location changes of the clones. The negotiation paradigm tolerates a large degree of latency in the accuracy of this information, since in all but the final phases of the algorithm this information is used to evaluate the cost of resources; it's not crucial to know that 12 rather than 8 clones want a particular location. Furthermore, each processor only informs others of the *change* of the location of a clone. This is a very important design feature of the parallel placer and will be explained further. In order to complete placement successfully, the algorithm does need to know with complete accuracy the resource utilization in the final phases, and so one processor takes over near the end.

Our current parallel implementation uses multi-casting for interprocessor communication. Using IP multicasting (UDP sockets) allows a processor to generate a single message which is then broadcast at the switch level (almost) simultaneously to all of the other processors (within a limited "scope"). The amount of traffic is dramatically reduced since each update message now requires only 1 message rather than $N-1$ where N is the number of processors [11]. The overhead involved in IP multicasting is about 4 times less than unicast sockets. This feature helps the processors remain in closer synchronization during the execution of the algorithm.

We have conducted experiments with the IPv4 multi-cast routing and have experienced less than 1 percent drop in packets, but with a ten-fold increase in speed. The round trip delay of a multi-cast packet is about 0.25 msec, and all the processors receive the same message approximately at the same time. This is in sharp contrast to TCP uni-casting which is reliable but delivers with much larger delay variations.

The main obstacle in using IP multicasting is its unreliable nature. IP multicasting is unreliable because packets may be dropped or arrive out-of-order. There is extensive research in the networking area related to recovery of packet losses in multicast networking. The recovery techniques generally involve timeout, retransmission as well as sending ACK and NACK mechanisms [11].

Our solution to this reliability problem is to take advantage of the inherent tolerance in the negotiation paradigm. First, each packet has a sequence number. Hence out-of-order packets are detected and tolerated temporarily and subsequently forgiven if they don't involve movements of the same clone, as illustrated in Figure 9. Due to the decomposition of the netlist into trees and the fact that clones are placed as the trees are traversed, the chance that a series of out-of-order packets involve the same clone is almost nil. Further, minor packet loss is harmless if a subsequent packet related to the same clone is received; as illustrated in Figure 9. So far, we have observed IP multicast reliability problems only in the first category with no fatal effects.

At a finer-grain level, our parallel placer is implemented as multiple threads. There is one programming thread that is responsible for receiving, checking, and buffering incoming packets [8]. There is another main thread that is responsible for placement and sending the updates out as packets. This thread partitioning assures minimum interprocess communication overhead and improves the robustness of the parallel placer. The access and update of the packet buffer is coordinated using "locks" for mutual exclusion.

3.2 Results of parallel NAP

In P-NAP, processors communicate with each other by sending and receiving messages. The network serves as the medium for transferring messages. There is no guarantee regarding the order or timing of the arrival of the messages to a processor. This is a source of nondeterminism in the parallel placer. Table 6 shows results of P-NAP using multiple processors. Since a processor computes in nanoseconds and each communication takes milliseconds, it is not surprising to see that speedups are more prominent in the larger designs (`spla, pdc`) than in the smaller ones. Although our parallel placer has not been fine-tuned, we are glad to see that it converges for the very dense circuit `ex1010`. Typically, we observe no substantial degradation of delays and wirelengths; one surprise is that P-NAP yields better delay than the serial NAP for some circuits.

4. CONCLUSIONS

We have developed a negotiated-analytical placement algorithm (NAP) and implemented a serial placer. The NAP algorithm has been parallelized to run on a network of workstations. Experiments

```
P-NAP(netlist, FPGA architecture)

    Cover the netlist with trees
    Allocate trees to processors
    while the placement is not feasible
        for each tree T_i assigned to me
            Calculate analytical placement V_i of T_i
            for each node n_j in T_i
                find position for n_j in window centered at V_{i,j} minimizing
                    clone_cost + history_cost + shared_cost + Iteration(mod 2)Δwirelength
                if the position of the node changed
                    send out the new position of the node
            endfor
            Read and process all of the messages in the buffer
            Update delays and edge weights
            Update clone centers
            Send out the current delay
        endfor
        Update history costs
        Send out a message with the current iteration
        while(my iteration is > avg_iteration +2)
            Read and process messages
        endwhile
    endwhile
end
```

Figure 7: Parallel placement algorithm.

Design	1 proc			2 proc			3 proc		
	x,y wl	delay	time	x,y wl	delay	time	x,y wl	delay	time
alu4	3.42 2.78	318	8.36	3.46 2.77	340	5.63	3.46 2.90	343	4.68
apex2	4.12 3.69	376	9.13	4.08 3.79	386	6.69	4.09 3.89	410	6.12
apex4	4.86 4.15	342	6.99	4.99 4.16	339	5.69	4.89 4.26	354	4.54
des	4.28 4.06	458	6.33	5.06 4.80	398	3.85	4.99 4.82	410	3.96
ex1010	3.74 4.03	782	54.51	3.95 4.42	776	47.81	3.85 4.19	758	44.30
ex5p	5.37 5.28	312	27.91	5.89 5.70	298	21.26	5.96 5.85	291	22.5
misex3	3.83 3.62	303	6.12	3.93 3.86	309	5.41	4.04 3.76	306	5.03
pdc	4.90 5.18	609	69.09	4.98 5.25	650	40.78	4.98 5.32	657	37.65
seq	3.75 3.90	326	9.78	3.87 3.96	350	7.92	3.80 3.91	339	7.61
spla	4.38 4.08	553	52.31	4.47 4.22	572	30.48	4.50 4.08	533	26.53

Table 6: Results of multiple processors running P-NAP in terms of wirelength, delay and running time in minutes. P-NAP was run with fixed pads (obtained from the VPR placement).

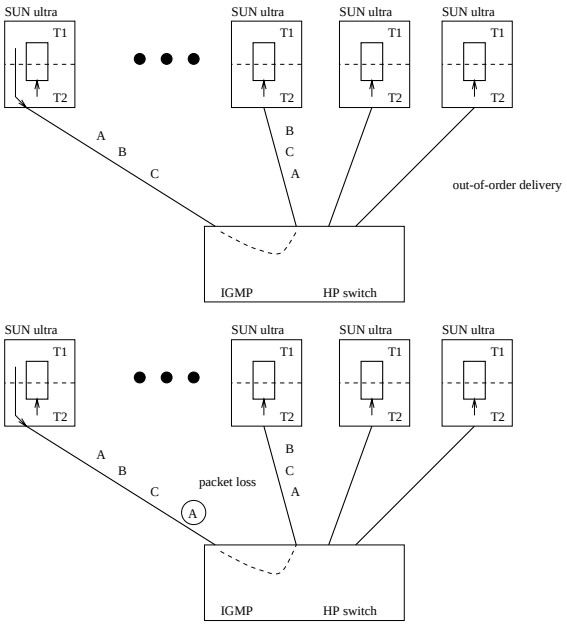

Figure 9: Packet delivered out-of-order and packet loss.

have demonstrated that our serial as well as parallel placers perform well with benchmark circuits. Challenges ahead include fine-tunning the parallel placer to attain more speedup and controlling the non-determinism.

5. ACKNOWLEDGMENTS

The authors thank Xilinx, UC MICRO, and SUN Microsystems for their support.

6. REFERENCES

[1] V. Betz. FPGA Place-and-Route Challenge. `http://www.eecg.toronto.edu/~vaughn/challenge/challenge.html`.

[2] V. Betz and J. Rose. VPR: A New Packing, Placement and Routing Tool for FPGA Research. In *Seventh International Workshop on Field Programmable Logic and Applications*, pages 213–222, London, England, 1997.

[3] P. K. Chan, M. Schlag, and J. Y. Zien. Spectral *k*-way ratio-cut partitioning and clustering. In *Advanced Research in VLSI, 1993 University of Washington Symposium on Integrated Systems*, pages 123–142, MIT Press, Cambridge, MA, March 15–16 1993.

[4] P. K. Chan, M. D. F. Schlag, C. Ebeling, and L. McMurchie. Distributed-Memory Parallel Routing for Field-Programmable Gate Arrays. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-19(8):850–862, August 2000.

[5] C.-K. Cheng and E. S. Kuh. Module placement based on resistive network optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-3(3):218–225, July 1984.

[6] M. Haldar, A. Nayak, A. Choudhary, and P. Banerjee. Parallel algorithms for FPGA placement. In *The Great Lakes Symposium on VLSI*, Chicago, March 2000.

[7] J. M. Kleinhans, G. Sigl, F. M. Johannes, and K. J. Antreich. GORDIAN: VLSI Placement by Quadratic Programming and Slicing Optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 10(3):356–365, Mar. 1991.

[8] S. J. Norton and M. D. DiPasquale. *Thread time : the multithreaded programming guide*. Prentice Hall, 1996.

[9] Y. Sankar and J. Rose. Trading quality for compile time: Ultra-fast placement for FPGAs. In *Proceedings of International ACM/SIGDA Symposium on Field-Programmable Gate Arrays*, pages 157–166, Monterey, California, USA, Feb. 1999.

[10] M. Senn, U. Seidl, and F. Johannes. High quality deterministic timing driven FPGA placement. In 10^{th} *International ACM/SIGDA Symposium on Field Programmable Gate Arrays*, Monterey, California, February 2002. Poster Abstract.

[11] W. R. Stevens. *UNIX Network Programming, 2nd ed.* Prentice Hall, 1999.