Adaptive Delay Estimation for Partitioning-Driven PLD Placement

Michael Hutton, *Member; IEEE*, Khosrow Adibsamii, *Member; IEEE*,
and Andrew Leaver, *Member; IEEE.*

Abstract—**This paper describes the application of weighted partitioning techniques to timing-driven placement on a hierarchical programmable logic device. We will discuss the nature of placement on these architectures, the details of applying weighted techniques specifically to the PLD CAD flow, and introduce the new concept of adaptive delay estimation using *phase local* to increase performance. Empirical results show that these techniques, in a fully complete system with large industrial designs, give an average 38.5% improvement over the un-improved partitioning-based placement tool. Approximately two thirds of this benefit is due to our improvements over a straightforward weighted partitioning approach.**

Keywords— **programmable logic, FPGA, PLD, timing-driven placement**

I. Introduction

ONE fundamental problem with weighted-partitioning algorithms for timing-driven placement is that early partitioning steps have very poor visibility of the eventual critical path. At the same time many architectures, hierarchical PLDs in particular, are architecturally well suited to partitioning-based placement, and we would like to use partitioning as a tool.

We consider the problem of timing-driven placement in the context of such hierarchical PLDs, using the APEX family of devices from Altera as an example. In this paper we show that partitioning is a viable flow for placement in hierarchical devices, and that relatively straightforward techniques, combined with several new twists on these techniques, can lead to dramatic improvements in design performance.

The use of recursive weighted partitioning for timing-driven placement is not in itself new. For example [1], [2] and later [3] describe general algorithms for recursive partitioning with scaled weights applied to critical nets. Other methods or enhancements have included probability distributions [4] or iterative rip-up and re-place of critical paths [5]. For PLDs, Frankle [6] discussed the idea of slack-allocation, or budgeting, using an iterative algorithm, where slack is defined as (constraint - delay) for any path in the network. This approach is related to the zero-slack methodology described in [7]. Ou and Pedram [8] add bounds on the number of times a net may be cut in successive partitions.

Simulated annealing methods [9], [10], [7], [11] have also been used, including the more recent VPR placer by Betz and others [12], [13], [14] specifically for PLDs. Senouci *et. al.* [15] address the issue of timing-driven placement on hierarchical targets. Though this work provides many interesting ideas, the algorithm presented is $O(n^2)$ and does not immediately meet our run-time goals. Many more advanced and complex placement algorithms have also been proposed using analytic methods [16] or variations on linear programming.

The place and route problem for PLDs differs from that for ASICs. The most pervasive difference is that for ASICs one typically wants to minimize total wirelength as a proxy for chip area. PLDs, in contrast, are prefabricated. One can roughly consider that wires are free up to the fixed line-count, and infinitely expensive beyond that point. Other differences are that PLDs have more predictable fixed delays (because wire-sizing and buffering is fixed) and that many decisions are very architecture specific. The output of place and route for PLDs is the settings for all switches required to program an implementation of the routed netlist into the prefabricated hardware.

The goals of this paper are to describe a production-quality application of a recursive-partitioning approach to placement. Though other algorithmic approaches have achieved better results than partitioning on standard-cell and gate-array architectures, we feel that our specifically hierarchical architecture is particularly suited to partitioning. Furthermore, some of the more subtle issues with production quality system (multiple clock-domains, combinational cycles, user-defined cliques) can be abstracted easily as illegal moves in partitioning but are difficult to introduce into some other other algorithms (simulated annealing being an exception). A further contribution of this paper is to introduce new ideas in the recursive partitioning flow which triple the benefit over a straightforward flows such as [1], [3] and to discuss the often overlooked effects of parameter tuning on the quality of results.

A preliminary version of this paper appeared in [17].

II. Background

A. Hierarchical PLD Architectures

Figure 1 shows a diagram of the APEX 20K400 programmable logic device with a hierarchical routing architecture. The basic logic-element (LE) consists of a 4-input LUT and DFF pair, typical of many PLDs. LEs have special-purpose carry and cascade logic to their neighbours for implementing arithmetic functions. Synthesis maps all logic into this element. Groups of 10 LEs are grouped into a logic-array-block or LAB. Interconnect within a LAB is a complete crossbar.

Research supported by and performed at Altera Corporation, 101 Innovation Drive, San Jose, CA 95134. All authors affiliated with Altera. Contact {mhutton,kadibsam,aleaver}@altera.com.

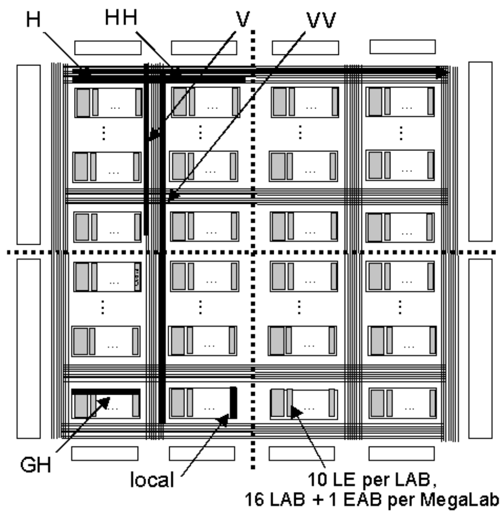

Fig. 1. Hierarchical APEX20K programmable logic device

Groups of 16 LABs and one EAB (2K memory block) form a MegaLab. Interconnect within a MegaLab uses a GH (MegaLab global H) line, then switches into a LAB. The top-level architecture is a 4 by 26 array of MegaLabs, so the device has 16640 4-LUT LEs. Communication between MegaLabs is accomplished by global H (horizontal) and V (vertical) wires spanning a half-chip and connected by programmable switching buffers. We denote a full-horizontal connection by HH.

Delays can be categorized as "local" for same-lab, GH+local within a MegaLab, V+GH+local for the same octant, V+H+GH+local for same quadrant, up to HH+VV+GH+local for a full corner to corner delay. As a rough metric, a local can be considered 1 unit of delay, and any other wire type 2 units. Thus the ratio of shortest to longest paths in the architecture is 1:11. Note that delays in a hierarchical architecture cannot be approximated by geometric (e.g. Manhattan) distance. The delay effects of fanout are largely removed by the hierarchical nature of the device, because a multiple fanout net is re-buffered at every switch-point. Fanout on individual wires is mostly beyond the granularity of placement and is considered a routing issue, since any problematic nets could also be split onto leftover prefabricated wires.

A natural algorithm for placement in this type of architecture is recursive partitioning as illustrated in Figure 2. In Phase I, we partition the netlist into left and right horizontal halves, minimizing cut-size. Cuts at this phase result in HH wires. Phase II bi-partitions each half independently, minimizing cuts or H wires. Phase III partitions into top and bottom half (VV wires), constrained by previous partitions. Phase IV is a k-way partition into individual rows (constrained by the existing partition hierarchy) to minimize V lines, and Phase V is the final partition of MegaLabs into individual LABs.

B. *Critical Paths*

For this paper, nodes consist of primary inputs and outputs, and registered or combinational LEs. A *path* is a directed path from a delay source (register or input) to a delay destination in the underlying graph and its *delay* is the sum of the delays on its constituent nodes and edges.

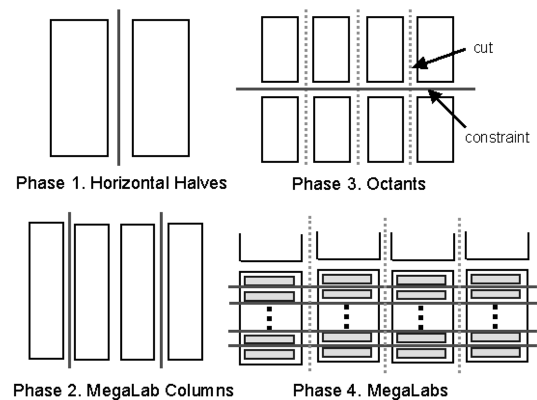

Fig. 2. Basic recursive partitioning algorithm

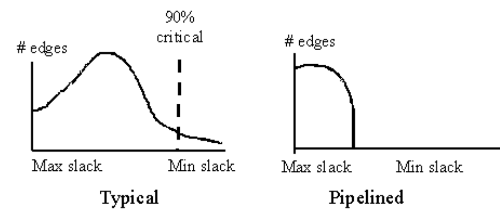

Fig. 3. Idealized slack profile.

Given an annotation of nodes and edges with their electrical delay we can compute, for each node and edge, the maximum delay from a source or to a destination, and hence the length of the longest path through that node or edge. Given a target delay (constraint) on an A to B path, we define the *slack* for that path as the most stringent constraint minus the actual delay of the path. For a given node or edge, we define the *slack* of the node or edge to be the minimum slack over all paths which contain the node or edge.

A *critical path* is one which has the minimum slack over all paths. In the special case of an ideal single-clock synchronous design this is equivalent to the longest path. Otherwise static timing-analysis is $O(pn)$ time, where p is the number of interacting path types, and n is the number of nodes in the graph. Many of the details which must be solved in a production system involve the added complexity of multiple clock-domains and constraints.

Define a k% critical path as any path with (100-k)% of the worst-case slack, measured over the (max-slack - min-slack) range of all paths. Thus a "90% critical" path has slack equal to 10% of the most critical path's delay. In other terminology, the path has a *slack ratio* of 0.1. The purpose of scaling is so that the algorithm can optimize min-slack rather than quitting as soon as all positive slack is achieved in the netlist. Similarly a k% critical node (edge) is defined as a path with slack (100-k)% of the worst-case node (edge) over the same range. Note that, by definition, all nodes (edges) on a k% critical path are k% or more critical.

Figure 3 shows two idealized slack distributions or pro-

files. In both cases the y-axis is the number of edges, and the x-axis is decreasing path slack. The left side shows a typical slack profile for the majority of netlists, with relatively few very critical edges, whereas the right side shows the pipelined case.

The logical method for changing a partitioning-based placement tool to a timing driven one is to add weights to the more critical edges. However, the real problem is in choosing these weights appropriately, particularly in the early stages of placement.

III. Adaptive Delay Estimation: Phase Local

Consider two paths early in placement, one with length 4 and the other with length 9. In the unit-delay model, we would under-estimate the cost of cuts on the shorter path and cut it many times. Using 11 for cuts and 1 for non-cuts we underestimate the potential delay of the longer path in earlier stages. It is not the idea to use weighted partitioning which is important in the overall algorithm, but rather how the weights are chosen, and how the timing model is parameterized.

This motivates our concept of *phase-local*. The phase local is an estimate of possible delays which, using statistical characteristics of previously placed netlists, gives the most probable result of later placement phases. This allows the standard algorithm outlined above to achieve a higher quality placement without that algorithm actually being modified. An additional feature of this technique is that by combining pessimistic and typical estimators one can further increase the efficiency of the overall algorithm. As a further benefit the estimator is both conceptually understandable and easy to implement in software.

A. The Phase Local

Definition 1: For a hierarchical architecture with h levels of hierarchy define the *phase-local* or *ploc* for phase i to be a weighted average of the probabilistic delays of all stages $i+1$ to h.

We have calculated, for a range of benchmark designs, a distribution representing for each “local” connection after phase i, and the resulting average number of H, V, GH and local wires occurring for that connection after complete placement. For our example device:

$$\text{ploc}(1) = f_1(\text{HH+VV+gh+loc, H+VV+gh+loc, H+V+gh+loc, ..., gh+loc, loc}),$$

$$\text{ploc}(2) = f_2(\text{H+VV+gh+loc, H+V+gh V+gh+loc, ..., gh+loc, loc}),$$

and so on. The value *ploc*(i) refers to the value of a local delay in the i'th partitioning phase.

For discussion purposes we will assume a simple linear combination of future cut-delays weighted by their empirical probability of occurrence i.e. the function f_1 is prob(HH)*delay(HH) + prob(VV)*delay(VV), ... For f_2 the HH cuts are now known exactly and do not appear in the calculation. All probabilities adjust accordingly. Future work would involve dynamic calculation of *ploc* based on timing constraints and characteristics of the current design (e.g. degree of pipelining).

B. Pessimistic Phase Local

In the same manner as we statistically (or otherwise) calculate the typical or expected *ploc*, we calculate the *pessimistic ploc*, defined as the weighted average based on the 95th percentile wires in the experimental trials (rather than the average, or 50th percentile wires).

The *ploc* value defines the expected delay of an edge, and hence determines the expected critical path delay for the netlist as a whole. The pessimistic *ploc* determines the resulting delay when edges on a path receive worse than average behaviour in future cuts (which will always be true for some paths and edges).

C. Choosing which edges to weight

By judicious choice of delay annotation we can further refine the benefits of using the *ploc*. The following pseudo-code outlines an algorithm which is significantly more successful than the basic approach at marking the “correct” edges. The critical percentage (*cpct*) is a parameter which will be discussed further in Section IV.

For phase i with delay(x), *ploc*(i), and pessimistic *ploc*(i) as defined above:

1. Delay-annotate the netlist using *ploc* for all unknown connections, and determine the critical path slack.
2. Define threshold = *critical_slack* + *cpct* * (*max_slack* - *min_slack*).
3. Delay-annotate netlist using *pessimistic_ploc*.
4. For each wire x in the netlist, mark x as critical if (x.slack_ratio < threshold).
5. Weight critical edges and partition.

The purpose of the two delay annotations is to ensure that we mark all potentially critical edges, in addition to those which are actually critical in the current predicted path. Essentially we mark all edges which, under a non-optimal future partition, will be within *cpct* of the current estimated critical path. By doing so, we minimize the possiblity that the critical path will change unexpectedly as a result of an unidentified edge being cut.

Using *ploc* alone, we find that we roughly double (12% to 24%) the benefit in overall design performance over simply annotating the critical edges based on a static timing analysis. This gain is virtually free, since the use of *ploc* is a minor addition to the algorithm and incurs an insignificant compile-time penalty.

IV. Parameterization and Phase Loops

In a heuristic approach of this type, parameterization is important. Our implementation includes a number of threshold values, and configurable values such as the number of re-tries. These values are particularly important in PLDs: over-optimizing early-on gives false savings (because wire-counts are fixed for the device) yet hurts future partitions.

Two important parameters are *cpct*, the critical percent threshold, or dividing line between critical and non-critical edges and *npct*, the threshold for maximum raw percentage of edges which may be weighted in the current phase.

As is common with heuristic algorithms, backtracking or re-doing a given phase with more information can be very useful. Rather than random multi-starts we refine the weightings in each loop. We give those edges which the partitioner did poorly on, or which are newly critical as a result of the partition, greater weight in the subsequent attempt. We refer to this as *phase loops*.

After empirical experiments to sweep many values, we use *cpct*=90% and *npct*=15%. We perform 3 phase loops on the first 3 phases and 2 on the more expensive k-way partitions of Phase IV and afterwards. We scale weights in the critical range linearly, so that a *cpct*-critical edge is lightly weighted and a 100% critical edge is maximally weighted. The exact parameters are not particularly relevant outside of our architecture, as they reflect our own preferences between run-time and quality of results, and more importantly are dependent on the PLD architecture itself.

We use a FM-based partitioner [18] for all bi-partitions and a Sanchis-based k-way hypergraph partitioning algorithm [19] for the k-way portions of the algorithm. Though various authors have reported recursive bipartitioning as generally superior to k-way [20], we found that resource balancing issues (drivers and switches rather than just wires) related to the PLD architecture make the k-way better for our specific problem. We used FM and Sanchis because we already had these available in production-quality code, and time-constraints prevented experimentation with more advanced techniques such as multi-level partitioning (e.g. hMetis [21]) in a first version. In future work we will extend hMetis or other multi-level algorithms to support carry-chains, RAM, secondary-signal networks and other architectural features with the hope for even greater gains.

We made modifications to the FM and Sanchis algorithms, most notably to support a wide variety of timing and other constraints, and to support arbitrary edge weights. Using multiple bins rather than a single move pool for the Sanchis algorithm was very important in order to deal with conflicting independent cost functions which arise from complex timing and pre-placed macros, carry-chains and logic/memory placement. We also generalized the loose-net removal ideas of Cong *et. al.* [22] for use in a k-way partitioner.

V. Overall Algorithm

Our complete algorithm follows the recursive partitioning phases given in Section II. The timing-driven aspects lie in the addition of edge weights to the partitioner as discussed in Sections III and IV. The resulting algorithm for a generic phase is thus as follows:

1. Compute *ploc*, and delay annotate the netlist as shown in the algorithm of Section III(C).
2. Compute a timing analysis and resulting edge-slacks.
3. Apply scaled weights to at most *npct* critical edges with slack less that *cpct*, as discussed in Section IV.
4. Perform the weighted partition.
5. Adjust the critical edge set and weights for phase_loops, as discussed in Section IV.
6. Repeat steps 1 to 5 for the parameterized number of phase_loops allocated to this phase, then choose the best result and continue to the next phase.

Not discussed in this paper are error-recovery steps. If a given step fails to find an acceptable solution, we will choose to return to a previous phase for a more (or less) aggressive solution.

The enforcement of carry-chain legality and user assignments are maintained through legality constraints passed on to the partitioner which forbid or penalize certain moves. Similarly, we count and balance secondary signals and carry chains as an intrinsic part of the move cost function.

A. Empirical Results

To examine the effects of our timing-driven improvements we compiled 20 large industry designs through the software. These designs contain between 60% and 100% of the number of LEs in the example device (16,640 4-input LUTs). The example device accomodates between 30K and 60K 2-input gates of user-logic and up to 104 2K-bit memories. The results are shown in Table I.

Though the production software attempts to meet all timing constraints, we will use simply "fmax of the slowest clock" as a metric of quality. We note that since all designs successfully fit in the PLD, the concept of wirelength is not very important. Run-time is reported both for the total flow and for only the place and route portion of the flow. All numbers are expressed as percentage change from the non-TDC version of the algorithm.

On average, we improve global design performance by 38.5% at a cost of 265% run-time, a very acceptable trade-off from a production point of view. An less aggressive tuning of the algorithm (essentially minimizing the number of phase loops) provides half the benefit at approximately half the cost.

Dividing up the improvements, roughly one third of the benefits come from weighted partitioning itself, a further third from the new concepts related to *phase local*, and the remaining third from the tuning issues related to *phase loops* and solving for the most appropriate parameterization by empirical trials. These are the incremental improvements found in the quality of results through the development of the software.

The division of benefit is worthy of note: the straightforward approach of weighted partitioning provides significantly worse results than an optimized system with tuned parameters and the additional application of the phase local concept.

Ideally we would like to contrast the results of this algorithm to more recent algorithmic techniques published in the literature. However, public-domain code typically cannot handle most of the issues (multiple timing constraints, cascade and carry chains, RAM, asynchronous paths) that are present in industrial designs and architectures.

	Less Aggressive			More Agressive		
	Δfmax	Δftime	Δttime	Δfmax	Δftime	Δttime
des01	24.1%	-7.6%	-6.8%	39.8%	12.7%	10.3%
des02	29.9%	33.3%	11.0%	54.7%	406.7%	82.2%
des03	-5.3%	69.9%	67.4%	26.9%	-0.5%	-1.0%
des04	-0.3%	0.0%	7.1%	40.4%	60.0%	60.7%
des05	2.8 %	42.9%	20.0%	6.1%	71.4%	50.0%
des06	23.9%	95.7%	66.2%	32.8%	178.3%	123.1%
des07	0.3%	520.0%	471.4%	1.1%	100.0%	96.4%
des08	4.8%	-92.0%	-80.6%	32.5%	68.8%	60.7%
des09	-34.8%	0.0%	-1.8%	-34.6%	63.6%	8.9%
des10	15.6%	88.9%	66.0%	24.9%	816.7%	586.0%
des11	137.8%	233.3%	95.1%	173.3%	666.7%	285.4%
des12	6.6%	945.5%	678.3%	50.2%	197.0%	139.1%
des13	8.7%	416.1%	320.0%	12.1%	741.9%	575.0%
des14	46.2%	361.2%	218.4%	56.5%	197.4%	114.7%
des15	78.5%	62.5%	13.7%	70.4%	368.8%	84.9%
des16	30.2%	72.0%	51.2%	48.7%	144.0%	93.0%
des17	9.1%	138.1%	8.4%	11.3%	209.5%	24.2%
des18	13.5 %	20.0%	12.5%	20.9%	52.0%	32.5%
des19	41.0%	37.5%	33.9%	61.4%	266.7%	230.4%
des20	25.1%	162.5%	152.6%	41.0%	684.1%	632.6%
avg.	22.9%	160.0%	110.2%	38.5%	265.3%	164.5%
med.	14.6%	70.9%	42.5%	36.3%	187.6%	89.0%

TABLE I

Fmax (performance) and compile-time results for full and partial TDC algorithm

VI. Conclusions

In this paper we have presented a discussion of timing-driven compilation for hierarchical programmable logic devices, and have given an algorithm to effect timing-driven compilation. Though we used the example of an APEX 20K device as motivation, the work and results are applicable both to hierarchical architectures in general, or to recursive partitioning approaches on any architecture.

The contributions of this paper include the introduction of *phase_local* for estimating critical path delay for partitioning steps which have not yet occurred. Phase local roughly doubles the benefit over a straightforward weighted partitioning approach. A further third of the benefit comes from appropriately parameterizing the algorithm and adapting the selection of critical paths across multiple phase attempts (*phase loops*), which underlies the flow. Together, these allow us to better target the true critical nets and achieve a higher-quality solution.

The benefits of the complete algorithm are clear and significant: we report a 38.5% average (36.3% median) improvement in register to register performance with acceptable (3.7X average, 2.9X median) compile time penalty. A less aggressive tuning of the algorithm gives half the performance gain, with half the compile-time cost.

The algorithm herein represents the only published placement method for hierarchical FPGA or PLD devices at current density ranges and containing modern device features. Though we report on designs for a 16,600 LE part, the software successfully operates on designs reaching 50,000 LEs with feasible compile time.

References

[1] M. Burnstein and M. Youssef, "Chip layout optimization using critical path weighting," in *22nd ACM/SIGDA Design Automation Conference (DAC)*, 1985, pp. 124–130.

[2] A. E. Dunlop, V. D. Agrawal, D. N. Deutch, M. F. Jukl, P. Kozak, and M. Wiesel, "Chip Layout Optimization using Critical Path Weighting," in *21st ACM/IEEE Design Automation Conference (DAC)*, 1984, pp. 133–136.

[3] M. Marek-Sadowska and S. P. Lin, "Timing driven placement," in *IEEE Int'l Conf. on Computer-Aided Design (ICCAD)*, 1989, pp. 94–97.

[4] W.E. Donath, R.J. Norman, B.K. Agrawal, S.E. Bello, S.Y. Han, J.M. Kurtzberg, P. Lowy, and R.I. McMillan, "Timing-driven placement using complete path delays," in *27th ACM/IEEE Design Automation Conference (DAC)*, 1990, pp. 84–89.

[5] S. Sutanthavibul and E. Shragowitz, "Dynamic prediction of critical paths and nets for constructive timing-driven placement," in *28th ACM/IEEE Design Automation Conference (DAC)*, 1991, pp. 632–635.

[6] J. Frankle, "Iterative and adaptive slack allocation for performance-driven layout and FPGA routing," in *29th ACM/IEEE Design Automation Conference (DAC)*, 1992, pp. 536–542.

[7] P. S. Hauge R. Nair, C. L. Berman and E. J. Yoffa, "Generation of Performance Constraints for Layout," *IEEE Trans. on CAD*, vol. 8, no. 8, pp. 860–874, 1989.

[8] S.L. Ou and M. Pedram, "Timing-driven placement based on partitioning with dynamic cut-net control," in *37th ACM/IEEE Design Automation Conference (DAC)*, 2000, pp. 472–476.

[9] C. Ebeling, L. McMurchie, S. Hauck, and S. Burns, "Placement and routing tools for the Triptych FPGA," *IEEE Trans. on VLSI*, vol. 3, no. 4, pp. 473–482, 1995.

[10] S.K. Nag and R.A. Rutenbar, "Performance-driven simultaneous placement and routing for FPGAs," *IEEE Trans. on CAD*, vol. 17, no. 6, pp. 499–518, 1998.

[11] W. Swartz and C. Sechen, "Timing-driven placement for large standard cell circuits," in *32nd ACM/IEEE Design Automation Conference (DAC)*, 1995, pp. 211–215.

[12] V. Betz, *Architecture and CAD for speed and area optimizatoin of FPGAs*, Ph.D. thesis, University of Toronto, 1998.

[13] A. Marquardt, V. Betz, and J. Rose, "Timing-driven placement for FPGAs," in *ACM/SIGDA Int'l Conference of FPGAs (FPGA00)*, 2000, pp. 203–213.

[14] P. Leventis, "Placement algorithms and routing architecture for long-line based FPGAs," Bachelor's Thesis, University of Toronto, 1999.

[15] S.A. Senouci, A. Amoura, H. Krupnova, and G. Saucier, "Timing-driven floorplanning on programmable hierarchical targets," in *ACM/SIGDA Int'l Conference of FPGAs (FPGA98)*, 1998, pp. 85–92.

[16] R. S. Tsay and J. Koehl, "An Analytic Net Weighting Approach for Performance Optimization in Circuit Placement," in *28th ACM/IEEE Design Automation Conference (DAC)*, 1991, pp. 636–639.

[17] M. Hutton, K. Adibsamii, and A. Leaver, "Timing-driven placement for hierarchical programmable logic devices," in *ACM/SIGDA Int'l Conference of FPGAs (FPGA01)*, 2001, pp. 3–11.

[18] C. M. Fiduccia and R. M. Mattheyses, "A Linear Time Heuristic for Improving Network Partitions," in *19th ACM/IEEE Design Automation Conference (DAC)*, 1982, pp. 241–247.

[19] L. Sanchis, "Multiple-way network partitionoing," *IEEE Trans. on Computers*, vol. 38, no. 1, pp. 62–81, 1989.

[20] J. Cong and S.K. Lim, "Mutliway Partitioning with Pairwise Movement," in *IEEE Int'l Conf. on Computer-Aided Design (ICCAD)*, 1998, pp. 512–516.

[21] V. Kumar G. Karypis, R. Aggarwal and S. Shekhar, "Multilevel Hypergraph Partitioning: Application in the VLSI Domain," in *34th ACM/IEEE Design Automation Conference (DAC)*, 1997, pp. 526–529.

[22] J. Cong, H.P. Li, S.K. Li, T. Shibuya, and D. Xu, "Large scale circuit partitioning with loose/stable net removal and signal flow based clustering," in *IEEE Int'l Conf. on Computer-Aided Design (ICCAD)*, 1997, pp. 441–446.