Timing-Driven Placement for Hierarchical Programmable Logic Devices

Michael Hutton, Khosrow Adibsamii and Andrew Leaver
Altera Corporation
101 Innovation Drive
San Jose, CA 95134
{mhutton, kadibsamii, aleaver}@altera.com

ABSTRACT
In this paper we discuss new techniques for timing-driven placement and adaptive delay computation for hierarchical PLD architectures.

Our algorithm follows the natural recursive k-way partitioning-based approach to placement on such devices. Our contributions include a specification of the overall TDC (timing-driven compilation) algorithm, an analysis of heuristics such as a variant of multi-start partitioning, a new method for adaptive delay computation, and a discussion of the structure of critical paths and sub-graphs on modern PLD designs.

This algorithm has been implemented in a production quality commercial tool, and we report on the results with and without the implementation of the new techniques. The basic result is a substantial 38.5% average (36.3% median) improvement in register-to-register performance across a range of real designs in modern density ranges, at a cost of approximately 3.65X average (2.88X median) place-and-route CPU time. (These improvements and costs are relative to the same tool prior to the efforts described in this paper.) A partial implementation of the new algorithm shows approximately half the performance gain, with approximately half the compile time cost.

Keywords
CPLD, FPGA, algorithm, heuristic algorithm, partitioning, placement, timing-driven placement, programmable logic.

1. INTRODUCTION
We consider the problem of automatic placement of a netlist (graph of 4-input LUTs and FFs) into a PLD architecture that is fundamentally hierarchical in nature. This type of architecture differs significantly in structure from a flat gate-array or island-style part. The goal of the work described is to quickly and easily introduce performance-driven compilation into an existing flow based on a recursive k-way partitioning placement algorithm for such a device. Though we will describe the work in terms of the APEX family of devices from Altera, and in particular the 16400 LE 20K400, the techniques and results should be interesting for any strictly hierarchical device, or for alternative architectures which utilize recursive partitioning as a placement technique

Our work was intended for production software, and thus subject to a number of constraints which might not be present in a purely research context.

Firstly, compile time is important. Though our primary goal is to increase design performance with an expected cost in compile time, an $O(n^2)$ algorithm would represent multiple days of compilation time for a 16000 LE part, and would be completely infeasible on a 50000 LE part (the largest device currently available). Thus we concentrate only on approaches which can be implemented in near-linear time. For our example 16,000 LE part, a 2 hour compile time should be rare, and typical time should be an hour or less.

Secondly, our delay annotation and timing analysis must support multiple delay constraints, both global and point-to-point. This means the algorithm must maximize the minimum delay slack (constraint minus actual delay) rather than simply the length of the longest delay path (because the slowest clock may not be the most critical), and must be amenable to more than one constraint.

Thirdly, the algorithm must incorporate a number of important modern architectural features. Carry and cascade-chain structures are of absolute importance. They modify (or "would modify" if they were not present) the critical path of a majority of designs, perturb the distribution of critical paths, and affect the effectiveness and hence choice of algorithm.

Lastly, we constrain ourselves to a solution that can be implemented in a short amount of person time. Since the goals of our project were to quickly introduce better performance from the tool, this means working within the basic confines of an existing recursive k-way partitioning placement. Though it is feasible in general to explore completely different approaches (e.g. simulated annealing, quadratic or force-directed placement, etc.) these would require a longer-term project in which large parts of the code could be re-written.

The structure of the paper is as follows. In Section 2 we survey some related previous work. Sections 3 and 4 introduce our problem and target architecture and outline a basic placement algorithm. Sections 5 to 8 discuss the distribution and structure of critical paths, introduce our overall changes to effect timing-driven compilation, define our adaptive delay computation, and

FPGA 2001, February 11-13, 2001, Monterey, California, USA.

finally give the complete algorithm. Section 9 gives empirical results demonstrating the efficacy of our techniques, and we conclude in Section 10.

2. RELATED WORK

Early papers concentrated primarily on gate-array or standard-cell like structures, and relatively small devices. Donath et. al. [5] though primarily discussing delay calculation for timing-driven compilation use a binomial probability distribution to identify possible routes from A to B in a gate-array-like part. Sutanthavibul and Shragowitz [16] compute the k most critical paths and iteratively re-place and re-route these paths until timing convergence is achieved. They use a constructive placement method, and also use path slack calculation to score these paths. Frankel [7] describes a "limit bumping" algorithm. Basically this is a slack allocation method, whereby slack calculated for a path is distributed evenly among its constituent nets. Bennett et. al. [2] follow a largely similar approach, covering more implementation details of a complete system. The primary target device in all these cases is a grid or gate-array-like continuous architecture, and except the last, all of these works were created for very low-density designs by today's standards.

Swartz and Sechen [17] (TimberWolf) and Ebeling et. al. [6] use simulated annealing as a tool for timing-driven placement. In the former, the target is standard-cell devices, in the latter the Triptych FPGA architecture. Nag and Rutenbar [11] also use simulated annealing. They attempt the placement and routing problem simultaneously with the goal of achieving better performance, but at a much higher computation time.

More recent work by Betz [3] and by Marquardt, Betz, and Rose [10] is the VPR tool. VPR is a simulated annealing based tool for placement into island-style FPGAs. The tool uses carefully crafted and tuned cost functions to achieve high-quality timing and placement solutions in reasonable time; these make it the current "champion" among academic tools for placement of MCNC and other public-domain benchmark circuits. However, published versions of VPR do not handle carry-chains or multiple delay constraints, and apply primarily to non-hierarchical, island-style architectures. Extensions to this work [9] have addressed row-based architectures, but not fully hierarchical devices.

Senouci et. al. [15] address the issue of timing-driven placement on hierarchical targets. Though this work is interesting for understanding the structure of critical paths and cones, a primary operation in their algorithm is to compute predecessor cones of all delay destinations. Since they consider only relatively small combinatorial circuits this computation is bounded by O(n * number of primary outputs), which is feasible. However, typically industrial designs have a significant proportion of flip-flops, so for sequential circuits this computation is inherently $O(n^2)$ in the design size and thus too expensive. An additional path-based work by Sawkar and Thomas [14] is also inherently $O(n^2)$, and does not suit our purposes for reasonable run-time.

Recently, Ou and Pedram [12] gave a timing-driven placement algorithm for gate arrays based on a mixture of partitioning and quadratic placement. One of the heuristic goals in their algorithm is to minimize the number of times a net is cut in successive partitions.

3. HIERARCHICAL ARCHITECTURES

The problem of placement into a hierarchical device is fundamentally different than placement into a flat or "island" style device. Though logic-cell delays can be assumed to be identical, island-style interconnect is closely approximated by geometric proximity, whereas delay in the hierarchical part is based upon hierarchical containment, which follows a step function rather than a continuous growth. Basically if two signals are within the same hierarchy level, it doesn't really matter where within that hierarchy they are. Similarly, geometrically close cells incur greater delay to get to other locations outside their hierarchical boundary than to distant cells within their hierarchical boundary. Depending on the degree of hierarchy in the part (since we assume signals are buffered at connection points), high-fanout nets may or may not be of major concern.

Thus, unlike a gate-array or standard-cell like part, a hierarchical architecture has a "natural" placement algorithm based on recursive partitioning. We will discuss this in terms of our example device, which we describe now.

3.1 The APEX Programmable Logic Device

Figure 1 shows a diagram of the APEX 20K400 programmable logic device, a commercial product from Altera Corporation. The basic logic-element (LE) is a 4-input LUT and DFF pair . Groups of 10 LEs are grouped into a logic-array-block or LAB. Interconnect within a LAB is complete, meaning that a connection from the output of any LE to the input of another LE in its LAB always exists, and any signal entering the input region can reach every LAB.

Groups of 16 LABs form a MegaLab. Interconnect within a MegaLab requires an LE to drive a GH (MegaLab global H) line, a horizontal line the width of the MegaLab, which switches into the input region of any other LAB in the same MegaLab. Adjacent LABs have the ability to interleave their input regions, so an LE in LAB *i* can usually drive LAB *i*+1 without using a GH line. A 20K400 MegaLab contains 279 GH lines.

The top-level architecture is a 4 by 26 array of MegaLabs. Communication between MegaLabs is accomplished by global H (horizontal) and V (vertical) wires, which switch at their intersection points. The H and V lines are segmented by a bi-directional segmentation buffer at the horizontal and vertical centers of the chip. We will denote the use of a single (half-chip) H or V line as H or V and a double or full-chip line through the segmentation buffer as HH or VV. In the APEX architecture, H and V lines cannot drive LEs directly; they drive a GH which then functions as in the MegaLab case above (so an H delay implies an additional GH delay). Because of segmentation of the H and V lines, further hierarchies of same-row, same-MegaLab-column and same-quadrant are implied for both a routability and timing purposes. The 20K400 contains 100 H lines per MegaLab row, and 80 V lines per LAB-column (80*16 per MegaLab column).

We can thus categorize the delay of a point-to-point connection as "local" for same-LAB connections, GH+local for same-MegaLab, H+GH+local for adjacent horizontal MegaLabs, V+GH+local for same octant of the chip, and V+H+GH+local for same-quadrant of the chip. The maximum (worst-case) delay is an HH+VV+GH+local connection.

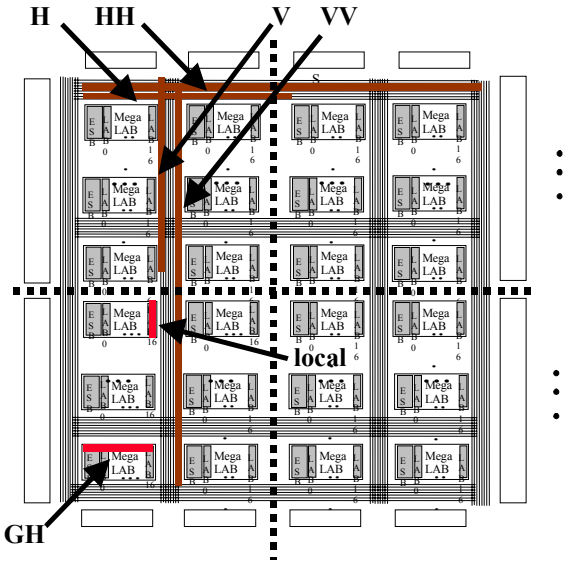


Figure 1. APEX device, top-level view, showing horizontal (H), vertical (V), MegaLab-horizontal (GH) and LAB-local (local) routing lines.

Roughly[1], a local consumes 1 unit of delay, a GH 2 units of delay, H or V 2 units of delay, and a HH or VV 4 units of delay. Thus the ratio of the shortest (local) to longest (HH+VV+GH+local) connection is roughly 1 to 11.

The LEs within a LAB can be configured with additional circuitry to form a carry-chain, which is significantly faster than general-purpose interconnect for implementing various arithmetic functions. The particular structure is not all that important to the software, as long as we can properly estimate delay and place arithmetic cells in the appropriately constrained locations. A carry chain wire is considered to be part of the logic and thus, for a legal placement, can be considered to have constant delay. Multi-LAB carry-chains (greater than 10 cells) form important placement problems, because they require the entire carry-chain to be placed in relative position.

Each MegaLab also has a 2K-bit configurable RAM, which is accessible in much the same way as a LAB. The delay through a RAM is also considered to be a fixed value independent of placement. However the choice of which MegaLab contains a particular logical RAM and the routing of address and data lines to the RAM are part of the placement problem.

The schematic details of carry-chains, cascade chains, and embedded RAM are beyond the scope of this paper (see [1] for details).

[1] The use of approximate delays is for easier exposition and because the ratios change with different aspect ratios across the range of the APEX family. Since this particular device is an example only, the interested reader can refer to the Altera data-book [1] or Altera software for exact wire delays on this or other specific parts and speed-grades.

4. BASIC PLACEMENT ALGORITHM

The natural way to place a netlist in this type of part is to do a recursive k-way partition. Each step will be referred to as a *phase* in the overall algorithm. Though the effectiveness of this type of algorithm is greatly affected by implementation details, the basic flow is as follows.

Phase 1: Partition the netlist into a left and right horizontal halves, minimizing the overall cut-size (number of HH wires).

Phase 2: Bi-partition each half independently, dividing the netlist into horizontal quarters (MegaLab columns) to minimize cut-size (H lines).

Phase 3: Partition the entire netlist into a top and bottom half, minimizing cut-size (VV wires) while respecting the MegaLab columns previously assigned. The result is 8 octants.

Phase 4: Partition the top and bottom netlists 13-ways into rows to minimize the number of connections (V wires) while respecting the previous vertical partitions.

Phase 5: Partition MegaLabs into LABs, trying to balance V-line usage in individual columns, and the input-region of each LAB.

This recursive partitioning approach is illustrated in Figure 2.

In each of these partitioning steps, the implementation details must count and balance secondary signals (clock, clear, enable), RAMs, and logic cells separately, as the resulting partition must be correct for all types of logic. Because carry-chains and other data-path elements have regular output, care must be taken to balance their relative placement to balance local congestion. As mentioned earlier, this can be a particularly difficult problem, since we often see more arithmetic functions in modern large designs.

A recursive partitioning approach has a number of advantages: The complexity of the algorithm is buried in the partitioning subroutine, a well understood problem in CAD which can be regularly updated independent of multiple architecture families. Graph partitioning algorithms are typically fast (linear) time, and thus the overall algorithm is also linear. Further, the partitioner can be de-coupled from the architecture-specific details and easily updated with new partitioning technology. The overall algorithm is simple and easy to maintain, even in the presence of heterogeneous resources and other constraints. Because the production system has to deal with numerous such additions to deal with a real architecture, and is regularly modified to deal with new architecture families and devices, these latter issues are particularly important.

As previously mentioned, the primary goal of this work was to quickly and easily introduce a timing-driven algorithm within the context of the existing place and route flow.

5. CRITICAL PATHS

We assume familiarity with a netlist defined as either a directed hypergraph (i.e. nodes and nets) or as a directed graph, and with the terms fanout and fanin applied to a node. For this paper, nodes consist of primary inputs and outputs, combinatorial 4-LUTs and registered 4-LUTs (LUT-DFF pair), carry or cascade-chain elements, and ESB memories.

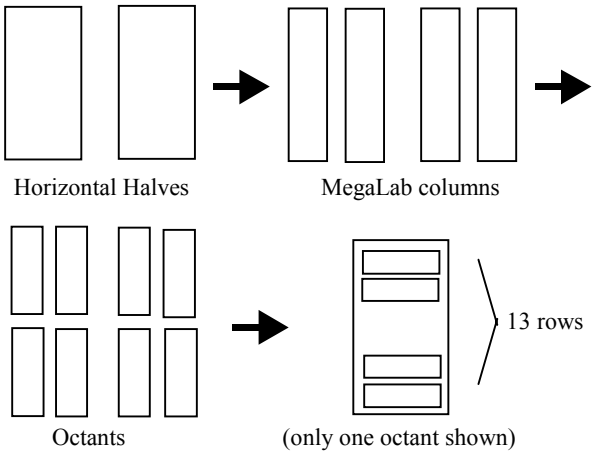


Figure 2. Illustration of recursive partitioning placement for APEX devices.

A *delay source* in a netlist is either a primary input or the output of a registered LUT. A *delay destination* is either a primary output or the output of a registered LUT. A *path* is a directed path from a delay source to a delay destination, and its length or delay is the sum of the delays on its constituent nodes and edges. Typically, since a large proportion of nodes (LUTs) are associated with a DFF, there are a quadratic number of source-destination pairs in a nelist. Both theoretically and empirically there are an exponential number of paths in a netlist and they cannot be enumerated efficiently. However, it is not hard to count either the number of paths or the number of paths going through a given node or wire, or to determine the length of the longest path through a node or wire.

Logic delays are always known for the architecture. Since interconnect delays for the architecture are known for a fixed distance and load, a fully placed netlist has a deterministic timing analysis. We assume a partially placed netlist has a known timing analysis only if the placement algorithm gives the timing analyzer an estimate of unknown interconnect delays for edges not yet routed.

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* delay 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. All these values are well-defined in a synchronous netlist, and computable in linear time with several graph traversals. Asynchronous circuits, either by design or arising from false paths, are common in practice but are beyond the scope of this discussion.

A critical path is one which has the minimum slack over all paths. In a single-clock system with a single global fmax (register-register) delay constraint, this is equivalent to the path with the maximum delay. In a multi-clock system with different fmax

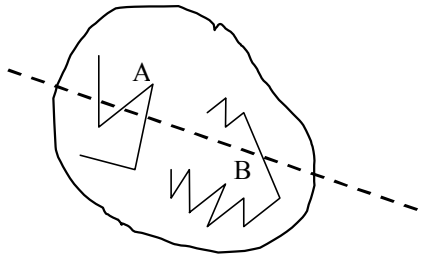


Figure 3. Two different length paths, early in placement.

specifications for different clocks, or with specific point-to-point path constraints, minimum slack is *not* equivalent to maximum delay. The goal of a timing-driven compilation is to maximize the worst-case (minimum) slack, not to minimize the worst-case (maximum) delay. However, for simplicity we will sometimes equate the two in discussion.

Since our goal is to modify a recursive k-way partitioning based algorithm, the obvious approach is to change the un-weighted partitioner into a weighted partitioner, and assign greater weight to cutting an edge which is on a critical path. The basic difficulty with this approach is in identifying the critical paths before the netlist has been placed.

6. ADAPTIVE DELAY ESTIMATION: PHASE LOCAL

The greatest issue with recursive partitioning as an algorithm for timing-driven placement is that the true path delays are not known in the early stages of placement. Thus the algorithm can minimize the number of critical nets which are cut early on, yet these nets are not actually the true critical nets by the time we are multiple levels deep in the placement. This problem is intrinsic whether or not the target architecture is hierarchical, but is exacerbated by non-uniform delays in a hierarchical part.

As an example, consider the two paths shown in Figure 3 during an early phase of the placement. If delay annotation assumes a unit-delay model, then the shorter path A will be largely ignored, and thus cut as many as 3 times. In a hierarchical device, this can be catastrophic to overall delay. However, if we consider the cut edge or net at its true value (e.g. 11 units) and all non-cuts as a local delay (1), the partitioner would successfully balance the number of cuts to (say) two in each path. When the various components of the B path are cut in later phases (unless the remainder of the path fits in the lowest hierarchy this would be required), the initial cuts plus the later cuts will result in a poor timing. In order to identify the correct critical nets for the partitioner, we really need to know which nets will later receive the less expensive cuts of lower hierarchy boundaries.

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 further benefit the estimator is both conceptually understandable and easy to implement in software.

6.1 Phase Local

Definition: For a hierarchical architecture with h levels of hierarchy define the phase-local 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. Though not crucial, it is of practical benefit to calculate this value statistically.

For our example of the APEX 20K400,

phase_local(1) = f_1(HH+VV+gh+loc, H+VV+gh+loc, H+V+gh+loc, …, gh+loc, loc),

phase_local(2) = f_2(H+VV+gh+loc, H+V+gh V+gh+loc, …, gh+loc, loc),

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

We found that a simple linear combination of future cut-delays weighted by their empirical probability of occurrence was sufficient for the first version of the algorithm, 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. We hope to do future experimentation whereby we tailor the definition of phase_local to individual characteristics of the design. This is because the probabilities of future cuts are also dependent on the relative slack distribution of edges, on net-size (fanout), on placement constraints (which we currently honour in the placement, but don't take into account for the phase local on other nets) and on the degree of pipelining. Because of pipelining, a design with a short unit-delay has a very different slack profile and cut probability distribution than one with much longer unit-delay and many carry-chains.

6.2 Pessimistic Phase Local

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

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

6.3 Choosing which edges to weight

By judicious choice of delay annotation we can further refine the benefits of using the phase local. The following pseudo-code outlines an algorithm which is much 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 7.

For phase i with delay(x), phase_local(i), and pessimistic_phase_local (i) as defined above:

1. Delay-annotate the netlist using phase_local for all unknown connections, and determine the critical path slack.
2. Define threshold = critical_slack + *cpct* * slack_range.
3. Delay-annotate the netlist using pessimistic_phase_local.
4. For each wire in the netlist
   if (x.slack < threshold) then
   mark x as critical.
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 the phase_local alone, we find that we get an average 10% or more benefit in overall design performance. This gain is virtually free, since the use of phase_local is a minor addition to the algorithm and incurs an insignificant compile-time penalty.

7. PHASE LOOPS

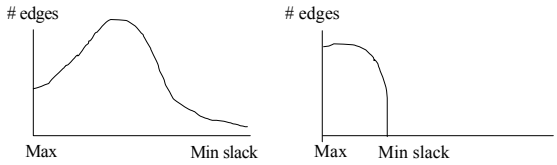
Though phase-local allows us to better target the true critical edges, we find that an additional heuristic is required to identify the changing nature of critical paths during the compilation.

Before we can discuss modifying the weights put on critical nets, we need to discuss critical path and slack distribution in more detail.

7.1 k%-Critical paths and edges

The most interesting empirical fact about the majority of designs which we encounter is that most of the netlist is not critical. Though there are specific exceptions (e.g. heavily pipelined designs with very short unit-delay), it is typical that the placement of the majority of cells is only important to the goal of routabiltiy, and does not affect timing. This is the basic motivation behind a recursive-partitioning approach, and for using a hierarchical architecture in the first place – if we can keep the critical path within hierarchy levels, and have only non-critical nets cross boundaries then we have an overall more efficient solution.

The critical path (or paths) is the path with the least slack (as defined in Section 5). 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. 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. (Note that this is necessary for benchmarking purposes, but any algorithm could be parameterized to quit once user timing constraints are met.) Similarly a *k% critical node* (*edge*) is defined as a path with slack (100-*k*)% of the worst-case node



a) Typical b) Pipelined (rare)

Figure 4. Idealized slack profile.

(edge) over the same range. Note that, by definition, all nodes (edges) on a *k*% critical path are *k*% or more critical.

Figure 4 shows two idealized slack distributions or *profiles*. In both cases the y-axis is the number of edges, and the x-axis is decreasing path slack. The situation of Figure 4(a) is the typical slack profile, where a relatively few number of edges are close to critical, and the majority of edges are in the 40% to 60% critical range. As we partition the netlist, edges are cut (delays introduced), and the curve tends to shift to the right. Our goal is to minimize the shifting of the tail as much as possible by cutting non-critical edges (in the heavy part of the curve), but not the critical edges in the tail. Some strongly pipelined netlists, such as shown in Figure 4(b) can have flat and short delay profiles, with upwards of 25% of edges critical. However, we find that the majority of netlists (more than 90% of the designs we have been seeing) are the form of Figure 4(a), and can be successfully attacked by judicious labeling of the most critical edges or nets.

7.2 Critical-edge Weighting

It is reasonable to define a parameter which specifies the boundary between critical and non-critical edges, weighting all edges with slack less than this value, and not weighting edges which have slack above that value. However the tuning of such a parameter is very important to the quality of the solution. The goals of routability and minimizing critical cuts are conflicting, and marking too many edges can cause the design to fail placement altogether. If too many edges are marked as critical, then we also increase the chance that a very critical edge is missed; if too few than we could end up cutting a potentially critical edge.

After experimentation, we chose to use two basic parameters to control the weighting operation: Define *cpct* or critical-percent-threshold as the dividing line between critical and non-critical edges in the current phase. Similarly, define *npct* as the maximum raw percentage of edges which may be weighted in the current phase. In the first version of the algorithm, we settled on 90% for *cpct* and 15% for *npct* after considerable empirical analysis. Some experiments have been done to consider values which change dynamically with the structure of the problem, and we expect to incorporate at least some aspects of dynamic parameterization at a later date.

Edge-weighting is not a binary consideration. We find it more beneficial to scale the weights within the critical range smoothly, so that a *cpct*-critical edge is virtually unweighted, whereas a 100% critical edge is heavily weighted. In the initial version of the algorithm we chose a simple linear weighting between 1 and a maximum value (1000 for the particular density at hand).

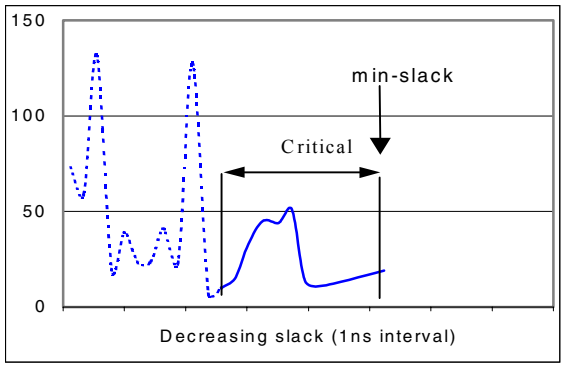


Figure 5(a). Initial identification of critical edges.

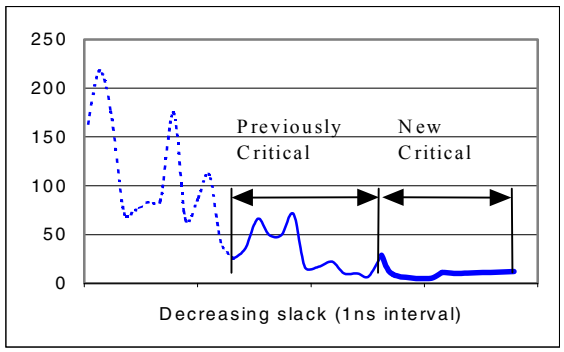


Figure 5(b). Identification of secondary critical-edge set after first partitioning attempt.

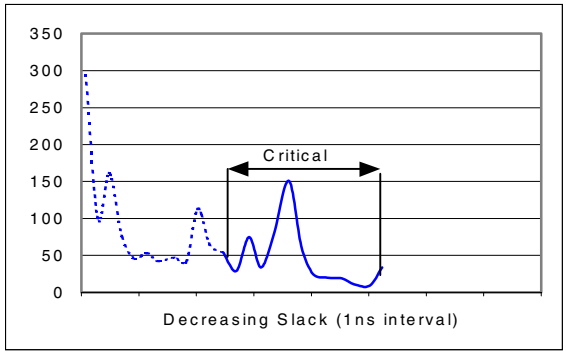


Figure 5(c). Result of second partition, meeting the original estimated min_slack.

The exact tuning of parameters such as *cpct* and *npct*, and constants chosen for weighting ranges are largely architecture and density dependent. Initial guesses at such parameters can achieve about half the gain of tuned parameters. This shows both that the general strategy is effective and also that tuning is required for the best possible solution.

7.3 Edge-Slack Migration

To this point we have assumed that the choice of critical edges using phase-local and the above parameterization is correct. In addition to the fact that the operation is heuristic, and can choose the critical edges poorly, we find that the critical path profile will

often migrate or change, due either to better or worse than predicted partitions, or simply due to the idiosyncrasies of particular netlists.

To solve this problem, we perform the operation of each phase several times (depending on the phase, and the results achieved). In-between successive attempts at a phase we re-analyze the distribution of critical paths, modify weights, introduce new (previously unweighted) edges into the critical set, and repeat the partitioning step to attempt a better result. This is quite different from simple multi-start partitioning, because we modify the problem, rather than simply partitioning multiple times with different initial partitions.

Figure 5(a) shows the tail end of the slack profile at the beginning of the current phase on an example netlist. The edges above *cpct* critical are identified as critical, and weighted as previously discussed. The critical set shown is from the *cpct* threshold down to min-slack. Our goal is to cut as few as possible of the edges in this range.

After partitioning we perform a new timing analysis and analyze the results, as shown in Figure 5(b). Without changing the actual phase-local, note that we can only do worse than the original min-slack (by cutting a critical edge). Thus the paths which do contain critical cuts are now beyond the min-slack predicted by phase_local. We then introduce new weighted edges into the critical set based on the previous threshold and re-partition. Previously weighted edges remain.

Figure 5(c) shows the final result. By weighting the new critical edges and repartitioning, we have no cut-edges in the critical set, and have achieved the best possible minimum slack for this phase.

8. OVERALL ALGORITHM

The basic algorithm follows the recursive partitioning phases as outlined in Section 4. The timing-driven aspects lie in the addition of edge weights to the partitioner as discussed in the previous sections.

The resulting algorithm for a generic phase is thus as follows:

1. Compute phase_local, and delay annotate the netlist as shown in the algorithm of Section 6.
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 7.
4. Perform the weighted partition.
5. Adjust the critical edge set and weights for phase_loops, as discussed in Section 7.
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 cliquing and user assignments are maintained through legality constraints passed on to the partitioner which forbid certain moves. Similarly, we count and balance secondary signals, carry chains and cliques as an intrinsic part of the move cost function.

8.1 Netlist Partitioning

Since all of the above discussion involves breaking down the timing problem into a series of weighted graph partitioning problems, it is important to discuss some of the implementation details of our partitioning subsystem.

Our partitioner currently uses a Sanchis K-way partitioning algorithm [13]. We chose this algorithm because it is simple to implement and is reasonably fast. We made a number of improvements and additions to the published algorithm, most notably to support a wide variety of constraints. We also support arbitrary edge weights between an output pin / input pin pair on a net. This is because timing algorithms (and ours in particular) want to optimize point-to-point delays rather than the delay of the entire net.

To improve partitioning quality, we have generalized the loose-net removal ideas of Cong *et. al*. [4] for use in a k-way partitioner. We found that loose net removal was particularly good at minimizing the cutweight (that is, the sum of the weights on all cut edges), and it also gives us a useful reduction in the average cutsize. We found, however, that a number of implementation details were necessary to deal with compile-time issues.

In order to deal with legality issues (secondary signals, carry chains, compound cells or cliques, etc.) which cause move-gains to be non-linear, we have implemented a multi-heap version of the Sanchis move pool which allows us to more efficiently search for best moves in the presence of competing metrics.

8.2 Tuning for Less Aggressive TDC

Since our overall algorithm is a combination of many different additions – phase_local, loose nets, looping, etc, we find that it is inherently tunable. That is, we can trade off improved performance against compile time. With modifications to reduce the number of loops and turn loose-net code off in later stages when it is less effective, we are able to produce a median result which splits the difference between full and no TDC effort. This allows users who are easily achieving their timing requirements with the less aggressive software to minimize their compile time.

9. EMPIRICAL RESULTS

To illustrate the effects of our timing-driven improvements, we compiled 20 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). All designs contain carry-chains, and about half use embedded memory features of the device.

As mentioned in the previous section, there are three different settings which we compare: "Off" means the basic algorithm of Section 4 is used. "Full algorithm" means the complete implementation of the techniques described in this paper were applied, and "less aggressive" refers to the partial implementation described in the previous section.

Table 1. Fmax (performance) and compile time results for full and partial TDC algorithm

	LESS AGGRESSIVE			FULL ALGORITHM		
design	**Δ fmax**	**Δ ftime**	**Δ ttime**	**Δ fmax**	**Δ ftime**	**Δ ttime**
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%
average	22.9%	160.0%	110.2%	38.5%	265.3%	164.5%
median	14.6%	70.9%	42.5%	36.3%	187.6%	89.0%

We report the percentage change[2] in fmax, and total compile time. Fitter time represents the time spent specifically in the place-and-route tool. Total time represents the "user experience" – including time spent in netlist extraction, synthesis and the final timing analysis. Fmax for this benchmarking exercise, is defined as speed at which the slowest clock in the device will operate for the design.

Although the software is capable of placing pins in more advantageous positions for both performance and routability, it is usually the case that user pin-assignments are fixed beforehand by board layout. To model this, we pre-process each design with a random assignment of pins.

Table 1 shows the overall results. On average, the full implementation of our algorithm shows a 38.5% improvement in fmax at a cost of 265% place and route time, or 3.65X the "no TDC" compile time. Since outliers typically tend to dominate the average, a more descriptive metric of the tradeoff is the median, which shows 36.3% better fmax at a cost of 2.87X run-time. In terms of total compile time the typical or median user sees a 1.89X compile time penalty.

With the less-aggressive setting we see roughly half the performance gain (22.9% average, 14.6% median), and comparably half the compile-time penalty. Since this is a tuned option, that behaviour is by design rather than coincidence.

We note that in one case (3 for less aggressive) the timing-driven algorithm achieved worse results than the "no TDC" version. We expect outliers to disappear once the algorithm has been tuned for special cases. Similarly, there are several cases where the TDC algorithm reports better compile time. We attribute this primarily to more aggressive partitioning in the TDC flow achieving a more routable placement and shortening routing time.

Though the work reported on here contrasts the "no TDC" case to the "with TDC" case, we should point out a preliminary version of TDC in the previous version of the software achieved roughly 5-10% speedup for 2X compile time cost. Relative to that algorithm our TDC is about 30% better for 2X compile time (i.e. there was a release of the production Quartus software in-between the start and finish of the TDC project as a whole.)

10. CONCLUSIONS

In this paper we have presented a discussion of timing-driven compilation for hierarchical programmable logic devices, and 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.

As components of the algorithm, we gave a new method (phase_local) for estimating critical path delay for partitioning steps which have not yet occurred, and for adapting the selection of critical paths (phase_loops) across multiple phase attempts, which underlies the flow. Together, these allow us to better target the true critical nets and achieve a higher-quality solution. We also discussed various implementation details that improve the underlying graph partitioning algorithm.

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.

The work presented in this paper was implemented for the February 2000 (Ver. 00.02) release of Altera's Quartus software. In Quartus the parameterizations discussed here are referred to as

[2] Raw or absolute fmax is a function of process generation, speed-grade, and designer objectives. Since this is a research rather than marketing communication we chose also not to display raw compile times. Typical compile times on our desk-top machines for this particular device range from as little as 15 minutes to upwards of several hours, depending completely on the difficulty of the particular design chosen. Thus a 3X slowdown in the overall compilation time is a very acceptable price for the performance gains we report.

TDC effort "off", "normal" and "extra-effort" compiler settings. Interested readers can independently verify or experiment with the production software, but should be aware that the code is under constant improvement and the performance vs. cost parameterizations and the algorithms themselves may be modified significantly in later versions.

11. ACKNOWLEDGMENTS

The original algorithm for partitioning-based placement that we modified is due to John Tse and others at Altera. David Karchmer and Dan Stellenberg provided many new hooks for timing analysis and delay estimation which were required for this work. Thanks to David Karchmer, Jay Schleicher, David Mendel and Mario Khalaf for contributing their ideas and discussion.

12. REFERENCES

[1] Altera Corp. Device Data Book, 1999.

[2] D.W. Bennett, E.F. Dellinger, W.A. Manaker, Jr, C.M. Stern, W.R.Troxel and J.T. Young, "Frequency-Driven Layout and Method for Field-Programmable Gate Arrays", US Patent #5,659,484, Aug. 19, 1997.

[3] V. Betz, "Architecture and CAD for Speed and Area Optimization of FPGAs", Ph.D. Dissertation, University of Toronto, 1998.

[4] 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", Proc. IEEE Int'l Conf. On Computer-Aided Design (ICCAD), pp. 441-446, Nov , 1997.

[5] 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 Proc. 27th ACM/IEEE Design Automation Conference, pp. 84-89, 1990.

[6] 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, Dec 1995.

[7] J. Frankle, "Iterative and Adaptive Slack Allocation for Performance-Driven Layout and FPGA Routing", in Proc. 29th ACM/IEEE Design Automation Conference, pp 536-542, 1992.

[8] M. Hutton, "A Method for Adaptive Critical Path Delay Estimation During Timing-Driven Placement for Hierarchical Programmable Logic Devices", US Patent Pending.

[9] P. Leventis, "Placement algorithms and routing architecture for long-line based FPGAs", Bachelor thesis, University of Toronto, 1999.

[10] A. Marquardt, V. Betz and J. Rose, "Timing-Driven Placement for FPGAs", in Proc. ACM/SIGDA FPGA Conference, FPGA00, pp 203-213, 2000.

[11] S.K. Nag and R.A. Rutenbar, "Performance-Driven Simultaneous Placement and Routing for FPGAs". IEEE Trans. On CAD for Integrated Circuits and Systems, Vol. 17, No. 6, pp. 499-518, June 1998.

[12] S-L Ou and M. Pedram, "Timing-driven Placement Based on Partitioning with Dynamic Cut-net Control", in Proc. 37th ACM/IEEE Design Automation Conference, pp 472-476, 2000.

[13] L. Sanchis, "Multiple-way network partitoining", IEEE Trans. On Computers, Vol. 38, No. 1, Jan 1989.

[14] P. Sawkar and D. Thomas, "Multi-Way Partitioning for Minimum Delay For Look-Up Table Based FPGAs", in Proc. 32nd ACM/IEEE Design Automation Conference, pp. 201-205, 1995.

[15] S.A. Senouci, A. Amoura, H. Krupnova and G. Saucier, "Timing-Driven Floorplanning on Programmable Hierarchical Targets", in Proc. ACM/SIGDA FPGA Conference, FPGA98, pp 85-92, 1998.

[16] S. Sutanthavibul and E. Shragowitz, "Dynamic Prediction of Critical Paths and Nets for Constructive Timing-Driven Placement", in Proc. 28th ACM/IEEE Design Automation Conference, pp. 632-635, 1991

[17] W. Swartz and C. Sechen, "Timing-Driven Placement for Large Standard Cell Circuits", in Proc. 32nd ACM/IEEE Design Automation Conference, pp. 211-215, 1995.