Iterative and Adaptive Slack Allocation for Performance-driven Layout and FPGA Routing

Jon Frankle
Xilinx, Inc.
2100 Logic Drive
San Jose, CA 95124

Abstract

We give a generalization, called the limit-bumping algorithm (LBA), of a procedure of Youssef et. al. [1] that transforms initial connection delays into upper limits on delay suitable for performance-driven layout. LBA is a simple way to distribute slacks using arbitrary allocation functions. We then show how lower and upper bounds on connection delays can be used in the computation of upper limits for initial layout and for layout improvement.

The methods have been integrated into a delay-sensitive router for FPGAs. In 22 standard benchmark designs (with placements fixed), feasible system clock periods were reduced in every case, by an average of 14% and as much as 32%.

1. Introduction

In recent years, increasing attention has turned to the problem of performance-driven layout. This is largely because with increased chip complexity and switching speeds, wiring delay accounts for an increasing proportion of overall system delay.

The challenge for performance-driven layout tools is to implement designs so that total path delays from start pins (primary inputs or register outputs) to end pins (primary outputs or register inputs) satisfy specified limits. This presentation will focus on register transfers, for which the performance measure is the clock period T. We treat the long-path problem and ignore the related short-path problem. For a completed layout, the minimum feasible period $T_{achieved}$ is given by

$$T_{achieved} = \max_{\pi} \left[clock\_to\_out(start) + \sum_{block\ b \in \pi} Delay(b) + \sum_{connection\ c \in \pi} Delay(c) + setup(end) \right]$$

over data transfer paths π from start to end registers synchronized by the clock. We assume that block delays are fixed, and that the design is free of purely combinational cycles, *i.e.*, has no unclocked feedback.

The difficulty is that layout tools ordinarily work on individual nets, without attending to path requirements. Automated performance optimization demands interaction between the layout process and timing analysis of paths. For example, the results of timing analysis can be used to produce suggested upper limits on connection delays for the layout process, such that any layout that meets the limits would satisfy the performance requirements.

This work makes three contributions: (1) We give a generalization, called the limit-bumping algorithm (LBA), of a procedure of Youssef *et. al.* [1] that transforms initial connection delays into upper limits suitable for performance-driven layout. LBA is a simple way to distribute slacks using arbitrary allocation functions. (2) We show how lower and upper bounds on connection delays can be used in the computation of upper limits for initial layout and for layout improvement. (3) We demonstrate that major improvements in system performance can be achieved by integrating these techniques in an FPGA router.

1.1. General background

Early work [2,3] performed timing analysis using delay estimates, and gave critical signals higher weight during partitioning or higher priority during routing. The importance of interactions between timing analysis and place-and-route steps was highlighted in [4], but the weighting function used to mediate the interactions was not described. At each stage of recursive mincut in [5], non-critical connections got weights in inverse proportion to their slacks, and critical connections got slightly higher weights. In [6], improved performance was obtained by recursive partitioning and global routing with a re-weighting scheme in which a net's weight is simply incremented by one at any stage it is found to be critical; the evolving weights influence both partition costs and routing order.

A common problem with weight adjustment is that at the same time critical connections are improved, other connections can become critical. Another approach has been to perform layout with continuous guidance from

path constraints. In [7], incremental timing analysis is used to evaluate individual moves during placement. Linear programming has been used at each stage of recursive partitioning to track path constraints dynamically during placement [8]. Related work in [9, 10] used quadratic programming and reduced sets of active constraints to produce high-performance placements more efficiently.

The task of transforming the results of timing analysis into guidance (typically weights) for placement is nontrivial, as evidenced by the many heuristics that have been proposed. An interesting formulation in [11] allows the derivation of an *optimum* formula for connection re-weighting; it requires as inputs from timing analysis a set of precise limits on individual connection delays. We will see later that the routing process in some technologies can utilize connection delay upper limits more directly, to good effect. This increases the importance of developing effective techniques to compute such limits. Work on computing limits is reviewed below.

1.2. Integration of path analysis with layout

The slack of a directed path π is defined as $R(\pi) - A(\pi)$, where $R(\pi)$ and $A(\pi)$ are the required and actual total propagation times along π, respectively. A system is said to have a long-path timing problem if for some path, signals fail to propagate through the constituent logic blocks and interconnect within the required time. (This corresponds to a negative slack.) We also define a slack for each individual source-to-load connection c:

$$slack(c) = \min_{paths\ \pi : c \in \pi} [slack(\pi)] \qquad (1)$$

Let R(c) and A(c) be the earliest required and latest actual arrival times at the load pin of c, respectively. An equivalent definition of the slack on connection c is then

$$slack(c) = R(c) - A(c)$$

Computing slacks is straightforward [12]. Two linear-time computations are performed, one in which A(c)'s propagate forward, and one in which R(c)'s propagate backward.

When layout is complete, slacks are readily computed; but during the layout process, connection delays are not yet known. Still, a performance-driven layout system can make use of path analysis by iterating the following steps: estimate delays --> compute resulting slacks --> suggest delay upper limits --> aim to meet upper limits. Our focus is the third step of this loop, in which upper limits are derived from slacks.

1.3. The zero-slack algorithm and variations

Hauge *et. al.* introduced the zero-slack algorithm (ZSA) [13]. This algorithm begins by computing slacks based on a tentative set of connection delays chosen so that they meet the timing requirements. The algorithm increases the delays in this set until they are maximal in the sense that they still meet the requirements, but a further delay increase on any connection would produce a violation. These delays are provided to the layout tool as upper limits.

The ZSA identifies a continuous path segment with minimum non-zero slack. Excess delay is distributed uniformly among connections on that path segment, slacks are updated on other connections that are affected, and the process is repeated until every connection has zero slack.

Excess delays can also be distributed in proportion to physical measures, e.g., capacitance per fanout or capacitance change per fanout [14]. In [1], the slack of a path is budgeted to its connections in proportion to the function

$$weight(c) = LF(c) * AcL(c) \qquad (2)$$

where $LF(c)$ is the delay per unit load on c's source pin, and $AcL(c)$ is the external capacitance of c's load pin; other weighting criteria appear in [15].

Luk [14] sped up the ZSA by omitting the recomputation of slacks on connections whose slacks are altered by delay increases on the minimum-slack segments. This can create intermediate slacks that are negative, unless the increases are appropriately bounded. In practice, all slacks converge to near zero in a few iterations.

Another iterative procedure to budget slacks, called *Iterative-Minimax-PERT*, is described in [1,15]. The weight function in (2) is used to define multipliers f(c):

$$f(c) = \frac{weight(c)}{\max_{\pi : c \in \pi} [weight(\pi)]}, \quad \text{where } weight(\pi) = \sum_{c : c \in \pi} weight(c). \qquad (3)$$

In each iteration, the delay of every connection c is incremented by $f(c) * slack(c)$. It is proven that all slacks decline monotonically and converge toward zero.

In the next section, we give a generalization of *Iterative-Minimax-PERT*, called the limit-bumping algorithm, that accepts arbitrary multipliers $f(c)$ as inputs. We identify simple conditions on the $f(c)$'s that are necessary and sufficient to guarantee that slacks go monotonically to zero. Finally, we introduce new methods for setting both initial delays and f(c)'s. Section 3 describes the application of these procedures to an FPGA routing program. Section 4 presents experimental results.

2. The limit-bumping algorithm

The inputs to the limit-bumping algorithm ("LBA") are a netlist, timing constraints, block delays, and for each connection c, an initial delay I(c) and a multiplier f(c). The outputs of LBA are delays U(c), such that any layout whose connection delays are less than or equal to U(c) will satisfy the timing constraints.

Following [1], at each connection c, the product of f(c) and c's current slack is added to the current delay in each iteration.

Limit-bumping algorithm

```
/* Step 1: initialize. */
U(c) = I(c);

/* Let slack(c, U) denote the slack on connection c, given the
   set of delays U. Increase U until slacks are near 0. */
do {
   /* Step 2: */
   compute slack(c, U) for all connections c;
   close_enough = near_zero (slacks); /* (stop criterion). */
   if (!close_enough)
      /* Step 3: distribute slacks. */
      for every connection c
         U(c) = U(c) + (f(c) * slack(c, U));
   }
while (!close_enough);
```

2.1. Convergence conditions

THEOREM 1: Every term slack(c, U) generated by LBA decreases monotonically to zero for arbitrary inputs with non-negative slack if and only if the multipliers f(c) satisfy

(a): f(c) > 0 for every connection c.

(b): $\sum_{c:c\in\pi} f(c) \le 1$ for every path π.

Proof.

(NECESSITY): If either condition is violated, there will be inputs for which some slack does not decrease monotonically to zero. If (a) is violated, slack(c) may not decrease; if (b) is violated, slack(π) can become negative.

(SUFFICIENCY): If (a) is satisfied, then every positive slack decreases by at least a fixed fraction in each iteration, and hence converges to zero. We need only show that no slack can become negative, *i.e.*, that slack *decrease* ≤ slack.

Decrease in slack (path π)

$= \sum_{c:c\in\pi} f(c) * slack(c)$

$\le \sum_{c:c\in\pi} f(c) * slack(\pi)$ *(From (1): slack(c) = min slack of all paths that include c)*

$\le slack(\pi)$ (by condition (b))

Q.E.D.

From now on, we restrict ourselves to f(c) values that meet the conditions of theorem 1. We can thus safely refer to the multipliers f(c) as fractions.

2.2. Adapting known slack budgeting heuristics

Each heuristic in past implementations of ZSA has distributed path slack among connections in proportion to one or more functions weight(c). Alternatives already mentioned for weight(c) include:

w_1(c) = 1;	[13]
w_2(c) = source fanout (c);	[13]
w_3(c) = LF(c) * AcL(c);	[1]
w_4(c) = fanout(c) * k(c);	[14]
w_5(c) = fanout(c) * k(c) * capacitance(c);	[14]

where k(c) is a delay sensitivity (dt/dC for nets or dt/dR for source-sink pairs).

Any positive function *weight*(*c*) determines a corresponding set of fractions f(c) according to equation (3), which we restate here:

$$f(c) = \frac{weight(c)}{\max_{\pi:c\in\pi}[weight(\pi)]}, \quad \text{where } weight(\pi) = \sum_{c:c\in\pi} weight(c). \qquad (3)$$

Furthermore, fractions defined this way satisfy the conditions of theorem 1 *by construction*.†

LBA is particularly efficient when using fractions $f(c)$ given by equation (3) and values $weight(c)$ that do not change during the layout process. The fractions then need be computed only once. This can be done in linear time, *e.g.*, using depth-first searches. From then on, it is trivial to distribute slacks as soon as they are computed, using a single multiply-and-add per connection. No sorting or recomputing of slacks is necessary during delay distribution, as is required in most previous methods.

When we determine an $f(c)$ *a priori* by equation (3), we divide by the total weight on the static maximum-weight path through c rather than by the total weight on a path segment of minimum slack. Static fractions are thus "conservative", *i.e.*, sometimes smaller than what could be used. The only apparent downside of pre-computing the fractions f(c) is a possible increase in iterations for convergence - but we do not expect convergence to be much slower. Different paths through the same connection typically have similar weight. And when some path *does* have less weight (where pre-computation would be conservative) it is less likely to be critical.

† Consider a path π. If it is the max weight path through every one of its connections, then $\sum_{c:c\in\pi} f(c)=1$ exactly; if it is not, then any connection c that is on a path with greater weight would contribute correspondingly *less* to the sum.

2.3. Uses of realistic lower bounds

The best available lower bounds on connection delays, which we denote L(c), are a reasonable choice of settings for inputs I(c) to LBA. One benefit of using L(c) can be seen by considering what can happen when the input I(c)'s are set to zero, as in [1]. An unsatisfiable upper limit U(c) (< L(c)) could result, even in designs for which layouts exist that meet the timing constraints. To avoid this possibility, we should at least ensure that I(c) ≥ L(c) for every connection.

It is worthwhile to compute slacks based on the delays L(c) in any case. The computation yields a "best conceivable" clock period T_{low}, *i.e.*, what would result if every connection achieved its lower bound simultaneously. This is useful, independent of the time T_{wish} that the designer wants. Even if $T_{wish} < T_{low}$ (what the designer wants is impossible) we probably still want to push the design's performance. The request often reduces to: "make it run as fast as it can".

When timing constraint values have not been specified, we need a way to set them automatically. We set the system goal

$$T_{goal} = G * T_{low} \qquad (G \geq 1) \qquad (4)$$

where the constant G is based on experience with the layout tool. Starting with a conservative T_{goal} can have practical advantages - even if $T_{wish} < T_{goal}$. First, the performance achieved often beats the goal. Second, a solution in hand whose performance falls short may be valuable, especially when improvement methods are available.

2.4. Use of realizable delays

Suppose our tool has produced a layout. We will use upper bounds $D(c)$, namely

$D(c)$ = connection delays achieved in existing layout,

to derive new upper limits U(c) that (1) are consistent with a specified improvement in system performance, and (2) optimize the chances that our layout tool will meet them.

Our approach to performance improvement uses the following new weight function:

$$w_6(c) = D(c) - L(c) \qquad (5)$$

Unlike the other weight functions we have considered, w_6 depends on the current state of the layout, and thus cannot be pre-computed. $w_6(c)$ is the gap between the delay achieved on a connection and its lower bound. It thus measures c's maximum "potential for improvement". Our idea is to set the upper limits U(c) so that the amount by which each delay is asked to improve is proportional to this room to improve. We first show how to use w_6 in the LBA, and then examine a tiny example in detail.

First, set the desired new period T_{goal}, and compute every slack(c, D). Then, compute fractions f(c) according to equation (3), using function $w_6(c)$ for weight(c). Using the resulting f(c) values, apply the variant of LBA described below.

Since we start with the old delays, a requirement of improved performance means that one or more slacks will initially be negative. This destroys the monotonic change property from theorem 1. Although in practice convergence is fast even when negative slacks are allowed [14], we prefer to preserve monotonic change by running LBA twice (in two stages). In stage 1, the delay upper limits U(c) are bumped only for those connections with negative slack. The same multiply-and-add operation is used, with one precaution. Since in this stage the limits are *decreasing*, we must check not to allow any connection's upper limit to be bumped below its lower bound. Step 3 of LBA as revised for stage 1 is thus:

```
/* Step 3: distribute NEGATIVE slacks,
   don't violate lower bounds. */
for every connection c
   if (slack(c, U) < 0)
      U(c) = max (U(c) + f(c) * slack(c, U), L(c));
```

Stage 2 is the ordinary LBA, with input delays taken from the U(c)'s produced by stage 1.

Example: Suppose $D(c_1)$=20, $L(c_1)$=10; $D(c_2)$=20, $L(c_2)$=15; and we have slack -6 to distribute between c_1 and c_2. Since $w_6(c_1)$=10 and $w_6(c_2)$=5, c_1 gets twice as much delay decrement as c_2. The resulting values are $U(c_1)$=16, $U(c_2)$=18.

The delay decrement $\Delta(c)$ asked of connection c is $D(c) - U(c)$. Equation (3) makes these decrements proportional to weight. With weight w_6, the effect is to equate the "approach fractions" of different connections' U(c) values from their achieved delays towards their lower bounds, where

$$approach\_fraction(c) \equiv \frac{(D(C) - U(c))}{(D(C) - L(C))} = \frac{\Delta(c)}{w_6(c)} \qquad (6)$$

(The approach fraction in the example is 0.4 for both c_1 and c_2.)

The approach fraction seems a reasonable measure of how difficult it will be for the layout process to meet a new limit U(c). Weight function w_6 attempts to balance this difficulty among connections as much as possible. It is in this sense that the layout tool's chances for success are optimized.

3. Application to FPGA routing

3.1. Performance-driven FPGA routing

As noted in section 1.1, past work on performance-driven place-and-route has focused more on placement than on routing, because for most technologies, improvements to the placement algorithms have much more impact on performance. In field-programmable gate arrays (FPGAs), however, routing is unusually important. The fraction of delay due to routing in FPGAs is typically from 40% to 60%, which is much greater than that for mask-programmed gate arrays.

The Xilinx XC4000 architecture [16] offers a hierarchy of routing resources distinguished by their path lengths and delay characteristics. General-purpose interconnects (figure 1) pass through switching matrices between the blocks and are best for medium-length, local interconnects.

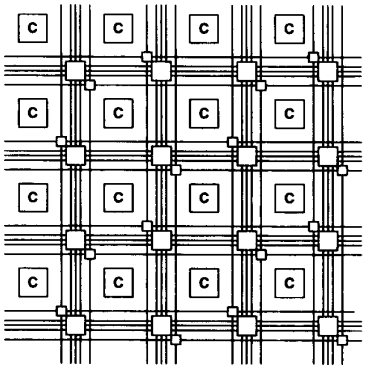

Figure 1. General-purpose wiring.

Other segments bypass alternate switchboxes, allowing longer paths with fewer switches (also shown in figure 1). Long lines (figure 2), which span the entire height or width of the array, are ideal for high-fanout nets and

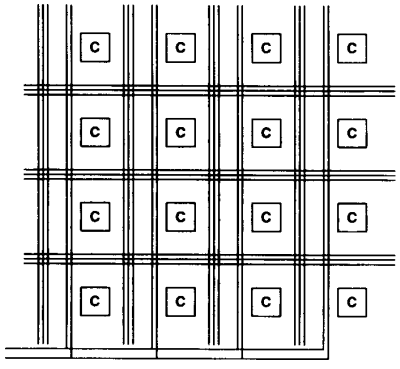

Figure 2. Long-line wiring.

time-critical signals. Special buffers can also be inserted in routing paths to speed up connections. The freedom to select the most appropriate resources for each connection means the router has more influence on delays than in traditional technologies. Improved routing algorithms offer great potential for enhanced performance in FPGAs.

To realize this potential for improvement, a router should be able to:

(1) Route connections within a specified delay limit when possible.

(2) Use different cost functions in different situations. For example, most routing choices must be made not according to delay alone, but according to a combination of delay and resource costs. Special nets such as power/ground signals should be oblivious to delay.

(3) support a retry mode to complete unrouted connections or improve existing routes. In this mode, previously completed connections can become temporarily unrouted.

Such capabilities are supported in existing routers, including a router for the Xilinx XC4000 family. Although this router is sensitive to delays, it has lacked the capability to decide what connection delay limits to aim for. Until now, a single upper limit has been used for all connections. What is needed is an optimized *set* of upper limits on connection delays. Guidance from path analysis lets the router realize its full potential to enhance system performance.

3.2. Initial routing solution

Following section 2.3, slacks are analyzed assuming that connection delays equal their lower bounds L(c), to find the best conceivable clock period T_{low}. Using (4) with G=1.5, our initial goal clock period is

$$T_{goal} = 1.5 * T_{low}.$$

Fractions f(c) are determined from equation (3) using weight function $w_1(c)=1$ (uniform slack distribution). We run LBA and use the resulting upper limits U(c) both to constrain the connections and to determine the order in which they are routed. As in the current delay-sensitive router, connections that require special resources are routed first. The others are routed in decreasing order of L(c)/U(c), so connections with tighter limits go first.

Connections that fail to meet their prescribed limits U(c) are left unrouted. If any connections fail, the limits are relaxed by 20% and the retry procedure is called. If failures still remain, the two steps of relaxing limits and calling retry are repeated until every connection is routed.

3.3. Improvement of an existing routing

We aim to reduce the clock period by one third of the maximum possible decrement:

$$T_{goal} = T_{achieved} - \frac{(T_{achieved} - T_{low})}{3}$$

Fractions f(c) are determined from equation (3) using the new weight function $w_6(c)=D(c)-L(c)$. We run the two-stage LBA from section 2.4 to generate new upper limits U(c). Every connection for which the already achieved delay D(c) exceeds the new U(c) is ripped up.

The connections are sorted in decreasing order of $D(c)/U(c)$, so that those whose delays most exceed their new limits go first. The retry procedure is then called. If unrouted connections remain, a loop is repeated as above in which their limits are relaxed by 20% and retry is called, until all are routed.

4. Experimental results

We experimented with 22 designs: *primary*1, from the MCNC layout benchmarks, and 21 MCNC finite state machine (fsm) examples. No logic optimization was attempted, so comparisons of absolute sizes with published optimization results are not meaningful.

Statistics on the designs are presented in Table 1.

design	elements	connections	ffs	ff transfer pairs	max levels	array size
bbara	137	392	12	45	5	8-by-8
bbsse	132	356	13	39	6	8-by-8
beecount	92	270	9	31	4	8-by-8
cse	240	679	16	55	7	10-by-10
dk14	164	523	7	27	5	8-by-8
dk15	104	323	4	12	5	8-by-8
dk16	459	1332	53	207	5	14-by-14
dk17	256	730	28	89	5	10-by-10
ex2	68	188	18	40	4	8-by-8
ex4	71	157	14	18	4	8-by-8
ex6	181	517	12	48	5	8-by-8
keyb	416	1217	26	66	7	12-by-12
mark1	95	201	16	32	4	8-by-8
opus	91	230	11	32	4	8-by-8
planet	388	1035	50	75	4	12-by-12
primary1	988	2426	269	2318	12	18-by-18
s1	389	1138	20	80	6	12-by-12
sand	630	1762	32	90	8	14-by-14
scf	579	1334	123	171	4	14-by-14
sse	132	356	13	39	6	8-by-8
styr	519	1494	31	95	7	14-by-14
tav	588	1769	18	32	7	16-by-16

Table 1. Statistics on the test designs after conversion to the Xilinx XC4000 family

For each design, we give counts of total elements (pads, function generators, and flip-flops), connections, flip-flops, and pairs of flip-flops involved in data transfer. We also indicate the maximum levels of connections on any one transfer path, and the size of the array of configurable logic blocks (CLBs)† used for layout.

Table 2 shows the minimum feasible clock period achieved for each design obtained by running the router three different ways. $T_{resource}$ is the result when delays are ignored, so that resource utilization alone is optimized. T_{delay} is the result with delay-sensitive routing, when every connection is given an identical upper limit on delay. T_{LBA} is the result using LBA to compute upper limits U(c) for initial routing and for one pass of routing improvement as described in sections 3.2 and 3.3. For comparison, the last column gives T_{low}, the feasible period if every connection could be routed at its lower bound delay. Table 2 also shows percentage improvements: from $T_{resource}$ to T_{delay}, from T_{delay} to T_{LBA}, and from T_{LBA} to T_{low}.

design	T_resource	T_delay (%improve) vs. T_resource	T_LBA (%improve) vs. T_delay	T_low (%improve) vs. T_LBA
sand	94.4 ns	101.0 (-7%)	68.7 (32%)	56.6 (18%)
s1	90.1 ns	66.4 (26%)	50.9 (23%)	44.6 (12%)
styr	83.3 ns	77.8 (7%)	59.8 (23%)	48.2 (19%)
scf	63.4 ns	47.7 (25%)	36.7 (23%)	30.0 (18%)
tav	122.7 ns	96.4 (21%)	73.8 (23%)	57.1 (23%)
dk17	63.2 ns	58.5 (7%)	45.9 (22%)	39.0 (15%)
planet	51.8 ns	48.3 (7%)	39.8 (18%)	30.3 (24%)
dk15	67.3 ns	50.7 (25%)	41.9 (17%)	36.0 (14%)
keyb	96.3 ns	73.5 (24%)	61.2 (17%)	51.0 (17%)
primary1	165.0 ns	131.1 (21%)	110.6 (16%)	82.4 (25%)
dk14	63.5 ns	51.2 (19%)	43.5 (15%)	36.6 (16%)
ex6	60.0 ns	47.7 (20%)	41.7 (13%)	36.6 (12%)
opus	36.7 ns	39.0 (-6%)	34.6 (11%)	29.3 (15%)
dk16	69.9 ns	49.6 (29%)	44.8 (10%)	38.3 (15%)
ex2	41.0 ns	30.4 (26%)	27.8 (9%)	23.9 (14%)
bbara	53.8 ns	42.4 (21%)	39.0 (8%)	35.2 (10%)
bbsse	45.5 ns	43.2 (5%)	40.3 (7%)	36.5 (9%)
beecount	43.8 ns	36.7 (16%)	34.3 (7%)	26.5 (23%)
sse	42.0 ns	42.6 (-1%)	40.1 (6%)	35.6 (11%)
cse	74.9 ns	63.0 (16%)	59.0 (6%)	47.9 (19%)
mark1	41.5 ns	32.8 (21%)	31.2 (5%)	27.8 (11%)
ex4	32.3 ns	29.8 (8%)	28.9 (3%)	24.3 (16%)
Average		15.0%	14.2%	16.2%

Table 2. Effect of router improvements on clock period

Not surprisingly, table 2 shows that delay-sensitive routing with a single upper limit for all connections gives substantial improvement in clock period (15% shorter, on average) vs. routing to minimize resource usage alone. There are cases in which delay-sensitive routing produces a longer clock period. This is understandable, because with a single limit, non-critical connections will compete for and sometimes win fast rout-

† Each CLB contains two flip-flops, two function generators capable of generating arbitrary functions of four Boolean variables, and other special-purpose logic.

ing resources, which can hurt the performance on critical paths.

Table 2 also shows that path-sensitive routing, guided by limits U(c) determined by LBA, yields *further* improvements to clock period for every design: 14% on average, and more than 20% in six cases. These results demonstrate that path analysis is especially valuable in improving system performance in FPGAs.

The percentages associated with the T_{low} column of table 2 tell us that not much more can be achieved solely through improvements to the router. We cannot expect every connection to achieve its minimum possible delay in the same layout. An average of at most 16% further performance improvement is possible, and in most cases even less. To improve performance further, we can apply LBA to earlier stages of layout.

5. Conclusions

(1) We have introduced LBA, a generalization of a known procedure to produce upper limits on connection delay, and demonstrated its utility to implement new approaches to slack allocation for performance-driven layout. (2) We showed how the process of generating upper limits can make use of realistic lower bounds and realizable upper bounds on connection delays. (3) We showed that integrating these techniques in an FPGA router yields major performance improvements.

Future work will study the effectiveness of LBA for other path types and for systems with multiple clocks. It will also investigate the use of LBA for performance improvement during FPGA technology mapping and placement.

References

[1] H. Youssef and E. Shragowitz, "Timing constraints for correct performance," *Proc. of ICCAD '90*, pp. 24-27, 1990.

[2] A.E. Dunlop, V.D. Agrawal, D.N. Deutsch, "Chip layout optimization using critical path weighting," *Proc. of the 21st Design Automation Conference*, pp. 278-281, 1984.

[3] M. Burstein and M.N. Youssef, "Timing influenced layout design," *Proc. of the 22nd Design Automation Conference*, pp. 124-130, 1985.

[4] S. Teig, R.L. Smith, and J. Seaton, "Timing-driven layout of cell-based ICs," *VLSI Systems Design*, pp. 63-73, May 1986.

[5] M. Marek-Sadowska and S.P. Lin, "Timing driven placement," *Proc. of ICCAD '89*, pp. 94-97, 1989.

[6] J. Garbers, B. Korte, H.J. Promel, E. Scheietzke, A. Steger, "VLSI-placement based on routing and timing information," *Proc. of the European Design Automation Conference*, pp. 317-321, 1990.

[7] 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," *Proc. of the 27th Design Automation Conference*, pp. 84-89, 1990.

[8] M.A.B. Jackson and E.S. Kuh, "Performance-driven placement of cell based IC's," *Proc. of the 26nd Design Automation Conference*, pp. 370-375, 1989.

[9] M.A.B. Jackson, A. Srinivasan, and E.S. Kuh, "A fast algorithm for performance-driven placement," *Proc. of ICCAD '90*, pp. 328-331, 1990.

[10] A. Srinivasan "An algorithm for performance-driven initial placement of small-cell ICs," *Proc. of the 28th Design Automation Conference*, pp. 636-639, 1991.

[11] R.-S. Tsay and J. Koehl, "An analytic net weighting approach for performance optimization in circuit placement," *Proc. of the 28th Design Automation Conference*, pp. 620-625, 1991.

[12] R.B. Hitchcock, Sr., G.L. Smith, and D.D. Cheng, "Timing analysis of computer hardware," *IBM J. Res. Devlop.*, Vol. 26, No. 1, pp. 100-108, 1982.

[13] P.S. Hauge, R. Nair, E.J. Yoffa, "Circuit placement for predictable performance," *Proc. of ICCAD '87*, pp. 88-91, 1987.

[14] W.K. Luk, "A fast physical constraint generator for timing driven layout," *Proc. of the 28th Design Automation Conference*, pp. 626-631, 1991.

[15] H. Youssef, "Timing analysis of cell based VLSI designs," Computer and Information Sciences, University of Minnesota, Ph.D. Thesis, January 1990.

[16] Xilinx, *The XC4000 data book*, Xilinx, Inc., 1991.