Throughput-driven floorplanning with wire pipelining.

Throughput-Driven Floorplanning With Wire Pipelining

Mario R. Casu, *Member, IEEE,* and Luca Macchiarulo

***Abstract*—The size of future high-performance SoC is such that the time-of-flight of wires connecting distant pins in the layout can be much higher than the clock period. In order to keep the frequency as high as possible, the wires may be pipelined. However, the insertion of flip-flops may alter the throughput of the system due to the presence of loops in the logic netlist. In this paper, we address the problem of floorplanning a large design where long interconnects are pipelined by inserting the throughput in the cost function of a tool based on simulated annealing. The results obtained on a series of benchmarks are then validated using a simple router that breaks long interconnects by suitably placing flip-flops along the wires.**

***Index Terms*—Floorplanning, systems-on-chip (SoC), throughput, wire pipelining.**

I. Introduction

CURRENT design of large integrated circuits already constitutes a formidable problem due to the complexity and interrelations involved; technology scaling, while representing the source of new possibilities of innovation and integration, at the same time forces designers to come to terms with novel issues. A particularly difficult problem which will soon become intolerable if not addressed with clear methodologies and design tools is the increasing relative timing cost of on-chip communications. It is projected that in a few years the time of flight of signals will make it possible to reach only a small fraction of the entire die in a clock cycle, thus possibly slowing the race toward higher clock frequencies that is indicated in the International Technology Roadmap for Semiconductors [1]. Furthermore, the very complexity that calls for such a dense arrangement of computational power on a single chip forces designers to embrace design paradigms that decrease the cost of redesign by reusing predesigned blocks (intellectual properties) whose input/output functionality is guaranteed, and whose modifications are kept to a minimum. These paradigms allow much faster time-to-market at the cost of limited flexibility. In such cases, incorporating latency issues in conventional design flow (for example, by performing full-chip retiming) might be impossible, due to either the sheer complexity of the flattened netlist or its actual unavailability. However, reducing IC frequency is not an option, especially for high-speed designs. This has brought many researchers to try a middle ground between classical synchronous design and more or less elaborated asynchronous paradigms that can guarantee functionality by abandoning the all chip synchronous hypothesis. The introduction of logic that controls the communication between blocks in a synchronous fashion is a solution that might represent a good compromise between a theoretically appealing but practically expensive asynchronous design and a more traditional approach. In addition, this comes with the advantage of being perfectly transparent to the designer and guaranteeing both timing constraints and correct functional behavior with little overhead. Such design styles allow the adoption of any possible clock frequency compatible with the specifications of the interconnecting blocks by intelligently pipelining the interconnects. However, the existence of computational loops forces a reduction in throughput that, if not properly controlled, could jeopardize all the advantage gained by the cycle time decrease. For this reason, and given that the loop constraints are derived by the physical placement of interacting blocks, a strategy that addresses the problem at the physical design stage is necessary.

In this paper, based on [17], we present a floorplanning and routing strategy with interconnect pipelining that takes into account the throughput problem. Furthermore, we highlight the extent to which these issues influence the full exploitation of a latency-aware methodology. The paper is organized as follows. After an analysis of previous work on the topic, in Section III we establish a few general results on the throughput of interconnected systems, and give a description of an algorithm that enables us to compute the maximum throughput of an interconnection of blocks. A heuristic function that approximates the throughput cost is described in Section IV. Section V describes the theory and practice of a flip-flop routing algorithm that is suited as a back end for the physical design of such systems. Various experimental evaluations are presented in Section VI, which is followed by the conclusion.

II. Related Work

The problem of how to insert pipeline stages in long interconnects has been recently addressed in its various aspects. At the system level, Carloni and Sangiovanni-Vincentelli report a methodology they call "latency insensitive" [2], [4], [5] that allows the preservation of the functionality of a system that is assumed working under zero wire delay constraint when an arbitrary amount of wire latency is added. The modified

Manuscript received June 28, 2004; revised September 19, 2004. This work was supported by the Center of Excellence for the Multimedia Radio Communications (CERCOM), Torino, Italy. This paper was recommended by Guest Editor P. Groeneveld.

M. R. Casu is with the Politecnico di Torino and CERCOM, I-10129 Torino, Italy (e-mail: mario.casu@polito.it).

L. Macchiarulo was with the Politecnico di Torino and CERCOM, I-10129 Torino, Italy. He is now with the Department of Electrical Engineering, University of Hawaii, Honolulu, HI 96822 USA (e-mail: lucam@hawaii.edu).

Digital Object Identifier 10.1109/TCAD.2005.846371

system works by means of a latency insensitive protocol that requires the addition of a couple of bits, valid and stop, to each of the communication channels between the various blocks of the system. While the overhead of the additional logic for the protocol management is negligible, the routing overhead may be relevant. We recently demonstrated in [6] another approach to latency insensitive design that preserves the performance of the previous approach but does not require the routing of the protocol signals. We assume these papers as a reference to show how systems can be made working with wire pipelining without RTL modifications of the netlist blocks in a way that preserves functionality. However, the number of valid operations per cycle, that is, the system's throughput, may be affected, as explained in the following section.

Concerning the physical synthesis aspects, Cong and Lim [7], and more recently Lu and Koh [8] and Chu *et al.* [9], formulate the problem of wire pipelining as one of retiming, therefore assuming that latches or flip-flops can be moved from logic blocks to interconnects. In a recent paper, Tong and Young [10] proposed a method for register placement along global wires, given a retiming solution. However, as already noticed in the introduction, while optimal, retiming can only be applied if logic blocks are described at least at RT level and either the logic description or the post-synthesis netlist is available. That is of course not possible for hard-IPs. As for soft-IPs, while theoretically possible, the manual rework to account for pipelined global wires at RT level might severely impact the design costs. We address here the case of intellectual properties that designers are likely to use in a plug-and-play fashion. The combination of retiming and interconnect latency insensitive design may lead to further improvements [11].

Other recent papers show how routing and buffer insertion techniques, all derived in the end from the classic paper by Van Ginneken [12], can be adapted to routing path construction and simultaneous flip-flop and repeater insertion [13]–[16]. In [13] and [14], only pin-to-pin connections are considered, while this paper also considers the case of multipin connections, like in [15]. None of these papers, however, addresses floorplanning issues or how netlist connections translate into final performance achievable from the buffer-flip-flop routing after a given placement. As far as we know, these issues were addressed for the first time in our preliminary work [17], of which this paper is an extension with new improvements. More specifically, with respect to the original paper, we give here a proof of the properties used for deriving the exact and approximate throughput estimation algorithms, a new flip-flop routing strategy with better throughput savings, comparisons with other possible cost functions, and detailed data rate considerations for frequency variations.

III. Throughput Evaluation

In order to be able to assess the performance of a design in the presence of added latency, it is necessary to consider a structural constraint. This kind of constraint has emerged in various disguises in the history of high throughput design (for example, see the *iteration bound* of Messerschmitt *et al.* [18]), but it consists in a relatively simple hard limit: Cycles introduce functional dependency in which overall latency is of the essence.

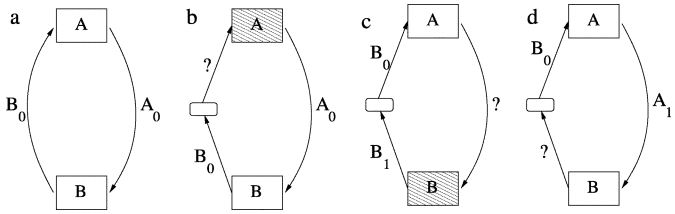


Fig. 1. Throughput limitation example.

We can follow the problem in the example of Fig. 1. In Fig. 1(a), two blocks are shown to be connected by a bidirectional link in such a way that, *in every clock cycle* the output of A is needed by B and vice versa. This is as tight a communication constraint as can exist but, as long as both connections are supposed to take no time, no performance bottleneck arises. If, on the other hand, we suppose that, for any reason one of the two connections is physically so long that it needs a pipeline in order to guarantee the maximum frequency constraint, it is clear that the system made up of the two blocks (and by extension any larger system incorporating it) will no longer be able to work at the maximum throughput. For, if at moment t_0 both A and B issue valid data A_0 and B_0, at the next clock tick t_1, while block B is able to produce the next valid datum for A, having processed A_0, block A will not operate on any valid datum, given that its next legal input B_0 was delayed by one clock cycle by the presence of the pipelining element p. Strictly speaking, this implies that the system as it is will not work: At time t_1 [Fig. 1(c)], block A will have performed a transition based on nonvalid data [labelled "?" in Fig. 1(b)] and will produce invalid data that will propagate through the system. This small example shows that simply adding a pipelining element modifies the functionality of the system (mostly unpredictably). To preserve it, we are left with two alternatives: to move existing memory elements from logic to wires or to control the synchronicity by selectively controlling the elements' clocks. The first alternative borders with classical retiming approaches, and it might be seen as the optimal solution: When in need of a memory element to segment a long connection, we reclaim one or more flip-flops from one or both of the connected blocks, thus changing at the same time the connectivity and the block in order to guarantee the same functionality.

If, for the reasons described in the introduction, we rule out such an alternative, the remaining solution is that of controlling the single block in such a way that it reacts only to valid signals. This requires an overall clock scheduling that will activate the various units in a coordinated way. A discussion of how this scheduling is enforced and whether a global control (as proposed by [22]) or a distributed approach [5], [6] is preferred is outside the scope of this paper; here, it is sufficient to say that such a control is possible. With this in mind, let us follow how such a strategy could be employed to "legalize" the situation of Fig. 1. At time 1, block A should be prevented from operating to avoid logical errors: let us suppose then its clock is gated. Block B will be able to operate normally, and therefore will produce new output B_1 (dependent on its state and on input A_0 which it can read). At time 2, A will finally be able to compute its next output A_1 (based on its valid input B_0); at the same time, B has to be stopped as the datum it needs from A (that is, A_1) is

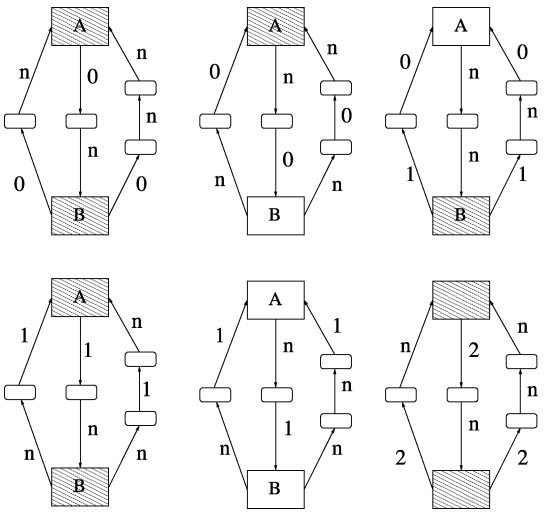


Fig. 2. Combined loop.

just about to be computed, and therefore not ready yet. From this evolution it is clear that, even though the whole system proceeds in a synchronous fashion and evolves on the basis of the clock ticks, any single signal is *stalled* from time to time. In this example, both A and B's output are stalled once every three clock cycles, though not at the same time. It is possible to prove not only that this result is possible (i.e., there exists a schedule that will allow a throughput of 2/3) but also that this is the best possible result for that cycle: The presence of a new pipelining element in the cycle has the consequence of adding one clock delay to any computation that starts at a block and ends at the same block. A computation that would have taken two clock cycles (the number of elements in the loop) to complete will now take $2 + 1$ clock cycles, thus degrading the average performance to 2/3.

Interaction between loops is such that the most critical one dictates the overall performance. An example is shown in Fig. 2: Even though the left loop might proceed at an average throughput of 1/2 as a self-standing subsystem, the presence of a common edge with the smaller maximal throughput of 2/5 forces it to slow down accordingly. It can be shown that this is always the case, i.e., the loop with the smallest ratio $m/(m+n)$ always controls the upper bound of the throughput, in each strongly connected component of the graph.

Before drawing a general conclusion, we need to define schedules formally and see how a block's schedule can influence neighboring blocks or pipelining registers. Let us define:

Definition: A **schedule function** for a block b is a function $s_b(i)$ mapping natural numbers into $\{0, 1\}$ such that $s_b(i) = 1$ implies that the block is active at clock tick i, and vice versa.

Definition: The **time function** of a block b is a function $t_b(i)$ mapping natural numbers into natural numbers such that $t_b(i) = j$ implies that the ith valid emission of the block is scheduled at clock tick j.

Schedule and time functions are alternative ways of specifying the schedule of a block (time function being a sort of inversion of the cumulative schedule). The time function for a block can be easily derived from its schedule function, as shown for

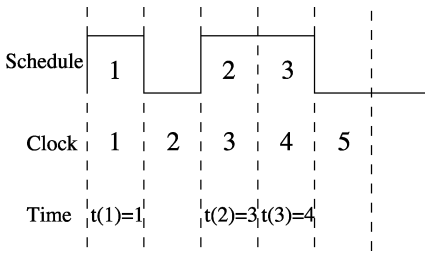


Fig. 3. Example of schedule and time functions.

the case of Fig. 3: We simply have to report as $t_b(i)$ what is reported as clock tick count under the ith occurrence of a 1 in the schedule.

Schedule function is a simpler way to describe what the single block is doing from the point of view of clock gating (and is therefore the easiest representation for implementation purposes). However, the time function representation can be used to derive interesting properties of schedules between connected blocks.

Lemma: If an input of a pipelining element p (be it another pipelining element or a block) has a time function $t_{\text{inp}}(i)$, the time function $t_p(i) = t_{\text{inp}}(i) + 1$ is a valid schedule for the element p.

Proof: In order for the register to be able to emit the ith valid signal, it needs to memorize it at least in the previous clock tick, so it must receive it 1 clock cycle before. □

Lemma: If an input of a block b has a time function $t_{\text{inp}}(i)$, the time function $t_b(i+1) = t_{\text{inp}}(i) + 1$ is a valid schedule for the element b.

Proof: In this case, the block, after operating on the ith signal, will be able to emit its $i + 1$th output in the following clock cycle. Note that the lemma makes the assumption that the sequential element behaves as a Moore machine. The assumption will be held true for all the following. □

Theorem 1: A strongly connected component of a block netlist always admits a schedule which allows an average throughput of $m/(m + n)$, where m and n are, respectively, the number of blocks and the number of flip-flops on the loop that makes the expression minimal, and no schedule exists that allows a better average throughput.[1]

(Partial) Proof: The proof is articulated in two main parts: we first show that simple cycles satisfy the theorem, and then, that any fully connected topology can support a schedule that satisfies it.

Simple Loops: Given a loop with m blocks and n wire pipelining elements [as exemplified by Fig. 4(a)], it is possible to use any schedule that contains m active slots out of $m + n$ slots in a period. To show this, let us consider any such schedule $t_b(i)$ and arbitrarily assign it to a block. Then, using properties expressed by lemmas we can derive acceptable schedules for all the following elements, until we find another valid schedule for b, which can be expressed (due to the associativity and commutativity of the transformations described in the lemmas) as $t_{\text{loop}_b}(i) = t_b(i - m) + m + n$. Now, the schedule t_b has m active slots out of $m+n$ so that $t_b(i-m)$ must be $m+n$ smaller

[1]The complete proof of the following theorem is not reported because it is outside the scope of the paper.

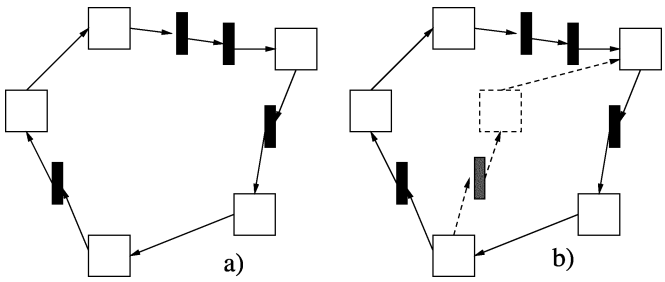


Fig. 4. Basis of induction and inductive step.

than $t_b(i)$, thus confirming the validity of the schedule. Any schedule with a smaller n/m ratio will necessarily generate a $t_b(i-m)$, not satisfying the constraint: Therefore, no better schedule exists.

General Topology (sketch of proof): Take any loop of the topology with the most critical (lowest) m/n ratio. The procedure outlined above will give a valid schedule for all the nodes and edges inside such loop [the solid loop in Fig. 4(b)]. Use the schedule that will maximally distribute the number of zeros in the schedule function (this is necessary to guarantee the success of the extension, but in certain cases it can be avoided). Then, extend the schedule to all other nodes by following the same basic ideas. Whenever a node has more than one input, only the one that leads to the latest arriving signal has to be considered for schedule calculation. Each calculated node needs to be propagated throughout its transitive fanout, until no schedule adjustment is possible. The schedules of the critical loops should not be touched. It can be shown that in no block will the extension cause a new input to be scheduled before it has to be actually processed, and that the process terminates (a detailed proof requires techniques derived from cost-to-time ratio problem theory, as they are applied in [6]). □

It might be asked whether only loops can contribute to throughput degradation. Reconvergent fanouts outside loops with branches holding a different number of delays need resynchronization, and therefore the fastest branch has to slow down its source, in order not to lose this data. However, this case substantially differs from that of loops in that throughput can be increased by **adding** flip-flops, while loops can better their performance only by **deleting** them. The constructive proof of the previous theorem implies such a conclusion, wherein the basic minimal buffering assumption is equivalent to the possibility of adding the correct amount of pipelining elements for each input that would need such an addition to equalize paths. Although it is relatively easy to add elements (for example, at the blocks' terminals), removing them might be impossible (removing a block will cause a timing violation, and the system will not be able to work at the given frequency). It can be shown that resynchronization can always be achieved by appropriate scheduling and/or communication protocols (details omitted, but see [6]), such that the system can operate at its theoretically maximum throughput.

In conclusion, a block netlist's best possible average throughput is the minimum value of $m/(m+n)$, where m and n are the number of blocks and delay elements, respectively, of a loop.

Even if the preceding proposition gives a complete **theoretical** answer to the question of throughput evaluation, it does not address the **practical** issue of how to evaluate real cases. In fact, the very structure of the cost formula, in which some nodes in the loop (blocks) increase its value, while others (delays) decrease it, makes it impossible to tell up front if the minimum will be found in the longest loop, or in the shortest, or somewhere in between. It would seem that the only solution is to traverse all possible loops, a hopeless task for all but the simplest cases, given the exponential nature of loop enumeration.

However, computation can be simplified. To see how, let us define a couple of terms.

Definition: The **length** of a path is the number of its blocks.

Definition: The **partial cost** of a path is $n/(m+n)$, where m and n are, respectively, the blocks and delays on the path.

Then, the key observation is expressed by the following

Theorem 2: Given two paths with common start and end nodes, if they have the same length, and if the partial cost of the first is greater than the partial cost of the second, it can never be the case that the second is part of a loop whose cost is higher than the first.

Proof: The two paths have the same length, meaning that m_1, the number of blocks of the first path, and m_2, the number of blocks of the second path, are equal. The partial cost is $n_1/(m_1+n_1)$ for the first and $n_2/(m_2+n_2)$ for the second. The hypothesis on the partial costs is equivalent to $(m_1/n_1) < (m_2/n_2)$. Therefore, $(1/n_1) < (1/n_2)$, due to the equality of the ms. In conclusion, $n_1 > n_2$. Now, the cost of any loop is a monotonous function of the ratio n/m, as can be seen by rearranging the fractions. For any possible path closing the loop between start and end, we have costs that are monotonous with $(n_1+n_c)/(m_1+m_c)$ and $(n_2+n_c)/(m_2+m_c)$, respectively. Therefore, as $m_1 = m_2$, the two fractions have the same denominator, and the first one is always greater than the second one. □

In practical terms, this means that, while traversing the netlist in search of the maximum cost $n/(m+n)$ (which is to say, one minus the minimum throughput $1-(m/(m+n))$), it is not necessary to record each path through which we visit a single node, but at most as many as the possible path lengths (limited by the overall number of nodes). The pseudocode that follows illustrates the basic structure of the algorithm.

```
Function computeExactCost ({V_in, E_in})
Input: GRAPH = {V_in, E_in}
Output: COST
exactCost({V_in, E_in});
forEach v ∈ V do
  listPartial(v) = {(0,0)};
  forEach vElse ∈ V − v do
    listPartial(vElse) = 0;
  end
  Next = {v};
  while ∃v_c ∈ Next do
    forEach v_o ∈ Fanout(v_c) do
      forEach (m_v, n_v) ∈ listPartial(v_c) do
        tempCost = (m_v + 1, n_v + intDist(v_c, v_o));
```

```
        if !∃(m_v + 1, n_old) ∈ listPartial(v_o) : n_old >
        n_v + intDist(v_c, v_o) then
          if v_o == v_c then
            updateWorstCost(tempCost);
          end
          listPartial(v_o) = listPartial(v_o)∪
            tempCost;
          Next = Next ∪ {v_o};
        end
      end
    end
    Eliminate v_c from Next;
  end
end
return WorstCost
```

listPartial is a list of partial costs that accumulates on each node visited in the breadth-first search, and is limited by the total number of nodes due to the previous theorem. **intDist** computes the number of register elements needed to connect the two nodes in order to evaluate the relevant parameter n of the cost function. Finally, **updateWorstCost** updates the value of current worst loop found.

The search is repeated for each starting node, but all nodes already used are marked (not shown in the pseudocode) so that no other path through them is considered in subsequent searches (all loops through them are already counted). Therefore, the maximum loop that passes through a given node can be assessed in a time bounded by the number of edges E (each edge is traversed only once for each possible partial length) multiplied by the number of nodes V (maximum number of partial costs recorded), and the overall problem has a complexity of $O(EV^2)$.

The algorithm was tested on classic MCNC (ami33, ami49, apte, hp, xerox) and some GSRC benchmarks (n10, n30, n50, n100) [29]. Average and maximum CPU times were 0.2 and 1.1 s, respectively. This is sufficient for performance evaluation, but too slow to be included in a floorplan optimization loop. For this reason, in the following we will describe the heuristic alternatives we evaluated and the solution we pursued.

IV. Floorplanning for Throughput

The high computational cost of the exact throughput was not the only reason that brought us to look for a different cost function to integrate in a floorplanning environment. There are also intrinsic reasons to look for a different way of expressing the cost. To discuss the question, we focus on a floorplanning implementation. In particular, we decided that the best choice of a floorplanning strategy was that of a simulated annealing approach. The availability of efficient and compact representations for floorplans (slicing and not) containing all feasible solutions with a minimal redundancy (corner block list [23], O-tree [24], Sequence Pair [25], to name just a few) has made simulated annealing the preferred method to implement optimizations on more diverse cost functions (such as array modules [26], routability and buffer planning [27], congestion control). Now, three conditions will decide whether or not a simulated annealing optimization will succeed.

1) The representation should be compact, exhaustive, and efficient.
2) The cost function must be easy to compute.
3) The cost function must be devoid of strong discontinuities.

While the first condition is satisfied by practically any of the aforementioned alternatives, both the second *and* the third make a strong case **against** the exact evaluator previously described. Simple modifications of the evaluator were tried, as detailed in the results section, but did not give satisfactory performance. An annealing process is able to settle on an acceptable solution only if its temperature schedule is sufficiently slow, and this already makes the evaluator impractical. Also, especially when we get close to the end of the annealing process (precisely when small variations on the cost will mean much for the final solution), the exact function is not smooth at all, due to the high sensitivity to small local changes: A minimal increase in a net's length could introduce a new delay in a loop, thus abruptly increasing the throughput cost. So, even if throughput itself is our goal, it might be a good idea to avoid including it explicitly in the cost function.

We needed a function that follows closely the real throughput trend (possibly monotonical with it) while being smoother and faster to compute. In order to make it as smooth as possible, we looked for a function of the whole floorplanned circuit rather than a maximum value. It had to be computed on the entire netlist and to depend strongly on the possible presence of delay on critical loops. A suitable function can be computed as follows.

1) Before entering the annealing iteration, we statically evaluate a weight for each pin to pin net, the inverse of the shortest loop the net belongs to.
2) At each iteration, we consider each pin to pin connection and evaluate their Manhattan distance.
3) The distance is divided by the maximum length admissible between clocked elements, and the integer part of the result is taken.
4) This last number is multiplied by the weight computed in the first point.
5) All such values are summed.

A pseudocode `computeHeuristicCost` for the preceding heuristic follows (**shLoop** represents the length of the shortest loop containing the (b, b_0) edge).

```
Function computeHeuristicCost ({V_in, E_in})
Input GRAPH = {V_in, E_in}
Output: COST
heuristicCost ({V_in, E_in});
Cost = 0;
foreach b ∈ Block do
  foreach b_o ∈ fanOut(b) do
    Cost+ = (intDist(b, b_o))/(shLoop(b, b_o));
  end
end
return Cost
```

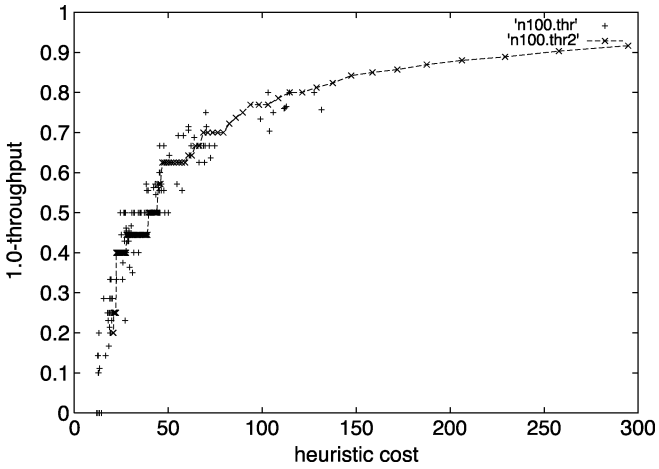


Fig. 5. Correlation between exact and heuristic throughput evaluation.

The rationale behind this function is simple. By dividing the Manhattan distance between two pins by the maximum length, we get an estimate of the minimum number of delays on that edge (all multipin nets are implicitly decomposed into pin to pin nets); each of these delays will count more or less according to the loop of maximum cost the net belongs to.

As the exact evaluation of this cost is very computationally expensive (it amounts to the algorithm of Section III), we suppose that small loops will have a great impact on the cost, and weigh every single delay so found according to the inverse length of the loop. Summing over the entire set of nets smooths the function sufficiently to make it suitable for an annealing process. It is true that the cost function herein described will rather optimize an average throughput than the best case, but a similar objection can be made for timing-driven cost functions. The rationale is that, on average, by reducing the mean throughput cost of cycles, the worst case will be affected in the right direction. The requirement for a smooth function in the annealing process makes a strong point in favor of this choice, and experimental results will show that the results are actually consistent with the expected optimizations. We ran some experiments in which the exact cost function was used (on an example—n10—in which this was possible, considering the computational cost). We observed that the introduction of an exact but abrupt evaluation did not allow any optimization whatsoever. We believe this is due to the difficulty that even a technique as robust as simulated annealing has in directing optimization with extremely peaked cost functions.

The loop computation is performed through a simple breadth first search starting from the destination of the edge at stake. Besides, being outside the loop, it represents a small fraction of CPU time.

Our experiments showed a pretty high correlation between this cost and the exact throughput. Fig. 5 reports the 1-complement of the exact throughput versus the heuristic cost for a floorplan of GSRC benchmark n100 [29] with many different values of the maximal distance. Similar behavior has been observed in the other benchmarks, thus confirming the good quality of the heuristic herein described. The connected line shows the case of a specific floorplan with varying length constraints, while the other points show results from different optimized floorplans. It is apparent that the two costs are closely related. Even though the measure cannot be used for an absolute evaluation of a system's throughput, it can certainly be used inside an optimization engine because heuristic and exact cost are monotonously correlated. It is interesting to observe that the heuristic function presents a smoother variation than the exact one, as we wanted in order to include it in the evaluation loop of the annealing process.

The curve appearance could raise the question of whether a different function could give rise to a more direct relation between heuristic and exact throughput, possibly benefiting the optimization process. For this reason, we tried to use a function of the single loops' behavior that could give rise to a more linear trend, but all experiments resulted in a similar or worse optimization results (not shown for space constraints). However, we do not exclude that a more elaborate function could result in a better matching between the heuristic and real throughput and at the same time guarantee a better optimization.

On the other hand, wirelength and throughput are very loosely correlated, if at all (see Section VI for examples).

We conclude this section with two observations.

1) Even though the cost function is based on the nets' lengths, it is not too strictly correlated with the overall wirelength, due to the integer division step. We observed experimentally that its main effect is that of taking into account only critical nets which are normally a small fraction of the total. Also, even critical nets that are not included in short loops (for example, nets from and to the terminals) do not contribute to the cost.
2) The computational cost of the evaluation is very small, because the most expensive work is done statically outside the loop.

V. Flip-Flop Routing

The number of flip-flops evaluated within the floorplan framework is a simple function of the Manhattan distance between two pins. Every point-to-multipoint (P2MP) net is decomposed into point-to-point (P2P) connections. As a result, the estimated number of flip-flops in a path connecting two pins of a P2MP net can be different from what is obtained after a routing aiming at minimizing wirelength, and so the number of flip-flops, *for each net*. In general, the throughput is maximized when the *overall* number of flip-flops in the paths forming the loop is minimum, while a router normally deals with one net at a time. Reducing the wirelength in each net can be a contrasting objective and may lead to further throughput degradation. In order to validate the results of the throughput based floorplanning, we built a simple router that places flip-flops along the interconnects such that given timing specifications are respected. We did not take into account many important issues like congestion and the need to use an accurate wire delay model even though we are aware of the inaccuracies of this approach. Blockages are also not considered. This assumption is justified by the already claimed need to exploit the porosity in large IP's layout for buffer insertion [20] that might also

be used for flip-flop insertion. The underlying assumption for wiring blockages is that two layers of metal are reserved for global routing that is allowed to run over the cells. However, for the purpose of verifying our approach, the router described in the following is sufficient.

The router inputs are the positions of nodes to be connected, internally represented as a graph $G = \{V, E\}$. The minimum spanning tree (MST) is first built using the rectilinear distance. Then, it is used by an algorithm that approximates the minimum rectilinear Steiner tree (MRST) using a heuristic that adds nodes to the graph, aiming at reducing wirelength by sharing common segments (and buffers and flip-flops if needed) among the sinks of the net. The field of efficient Steiner tree building is still being explored (see [21]). We alert the reader that the following algorithm is far from optimal in terms of the complexity and quality of the achieved heuristical MRST, which was outside the scope of our work. Its use is only propaedeutical to the flip-flop planning algorithm.

The algorithm works as follows. At each step of the algorithm, a node $u \in V$ that has not yet been connected is treated. All the unconnected nodes $v \in V$ surrounding u are first organized in four sets of sinks $\{N(u), E(u), S(u), W(u)\}$ based on their position (north, east, south, or west) with respect to node u. A single node v_i may belong to more than one set (e.g., both north and west, south and east). The direction for the next routing is $D \in \Delta = \{N, E, S, W\}$. For each set, the minimum segments that may be shared among the corresponding nodes are computed $\Lambda = \{\lambda_N, \lambda_E, \lambda_S, \lambda_W\}$. For instance, if (x_u, y_u) and (x_{v_i}, y_{v_i}) are the positions of nodes u and v_i respectively, and $v_i \in N(u)$ (i.e., $y_{v_i} > y_u$, $\forall i \in [1, |N(u)|]$), then $\lambda_N = \min(y_{v_i} - y_u)$, $v_i \in N(u)$. The direction $D \in \Delta$ is then chosen according to $\lambda_D \cdot |D(u)| = \max(\{\lambda_N \cdot |N(u)|, \lambda_E \cdot |E(u)|, \lambda_S \cdot |S(u)|, \lambda_W \cdot |W(u)|\}$ which maximizes segment sharing. Another node n is then added. If the direction was north, then $(x_n, y_n) = (x_u, y_u + \lambda_N)$. Node n inherits from u all nodes belonging to the set $S \in \{N(u), E(u), S(u), W(u)\}$ corresponding to the chosen direction, while u loses them. Then the algorithm proceeds in a breadth-first fashion visiting nodes inherited by n. After the exploration of the subtree of n, the algorithm goes back to u, whose remaining directions can be explored by reapplying the heuristic on the nodes that still have to be visited. With this approach, the added nodes stay on the Hanan's grid, built according to the position of the original nodes. The algorithm is graphically exemplified in Fig. 6.

After routing, a delay model is applied to the tree in order to evaluate if the timing constraints set at its leaves are respected. We suppose that optimal buffering is performed and then we compute the timing slacks. If some of the constraints are not met, the tree has to be legalized by adding some flip-flops. Every branch, i.e., every segment connecting two nodes, is then marked "legal" or "illegal." The legality is determined by computing the required time r at each node u starting from the leaves of the routing tree where a required time is set. In [17], we used the following approach for evaluating the legality. If u is not a branching node with only one child v, and if the directed edge $(u, v) \in E$ has delay $d(u, v)$, the required time $r(u)$ is given by $r(v) - d(u, v)$. If $r(u) < 0$ the edge $(u, \text{child}(u))$ is

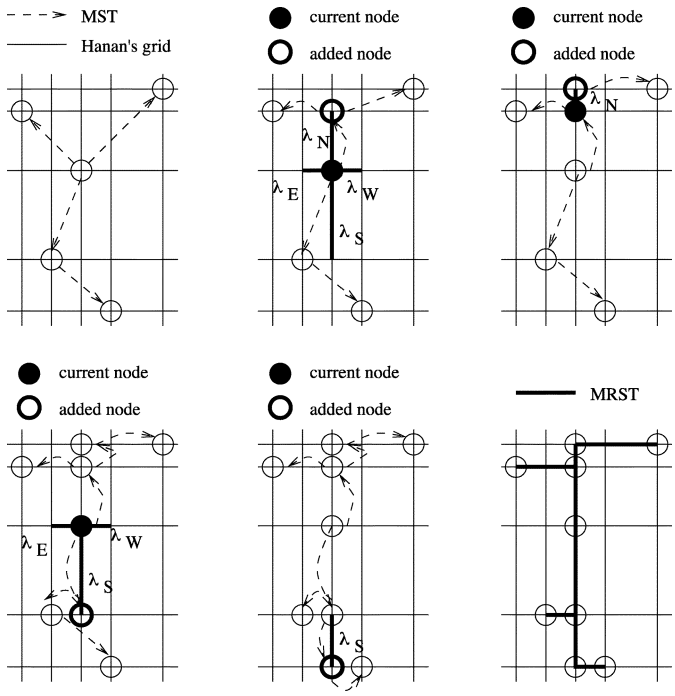


Fig. 6. Example of heuristical MRST construction.

marked "illegal;" otherwise, it is legal. If u is a branching node with N children, all the corresponding "candidate" required times $r_i(u), i = 1 \ldots N$, are computed and used as necessary to mark as illegal the related edges. Since one node can only have one required time, if at least one of the candidates is positive, $r(u)$ is set at the minimum of such positive values. This strategy leaves unaffected the already "legal" paths passing through the node. This function is basically a depth-first search, and its complexity amounts to $O(V + E)$. An illegal branch followed by a legal branch is then chosen as a feasible region for flip-flop placement.

The described approach has a drawback. We stated before that we annotate a branching node u with the worst (i.e., minimum) required time among the different required times coming from its edges: $r(u) = \min(r(u, v))$, $\forall v \in \text{children}(u)$. Suppose that u has two children, v and w, and that $r(u, v)$ is worse than $r(u, w)$ such that $r(u) = r(u, v)$. Suppose that the upstream branch from node z that ends in u is marked illegal because $r(z) = r(u) - d(z, u) < 0$. This means that the arrival time $a(w) > r(w)$. Then a flip-flop will be placed in (z, u) to legalize it. The latency at nodes v and w increases by $+1$, which is certainly correct for v. However, it might be the case that the path to w would have been already legal without the flip-flop addition, i.e., $a(w) \leq r(w)$. These circumstances are exemplified in Fig. 7(a) and (b), where a timing constraint violation at node v $(a = 21 > r = 20)$ traces back to the root z whose $r(z) = -1 < 0$. As a consequence, a flip-flop is added at (z, u), therefore increasing the latency at both nodes v and w. However, w was already legal being $a(w) = 16 < r(w) = 20$. This leads to a conservative solution that sometimes might worsen the system throughput if the added flip-flop is within a worst loop. Nonetheless, this can be easily avoided with an improvement we implemented in a new

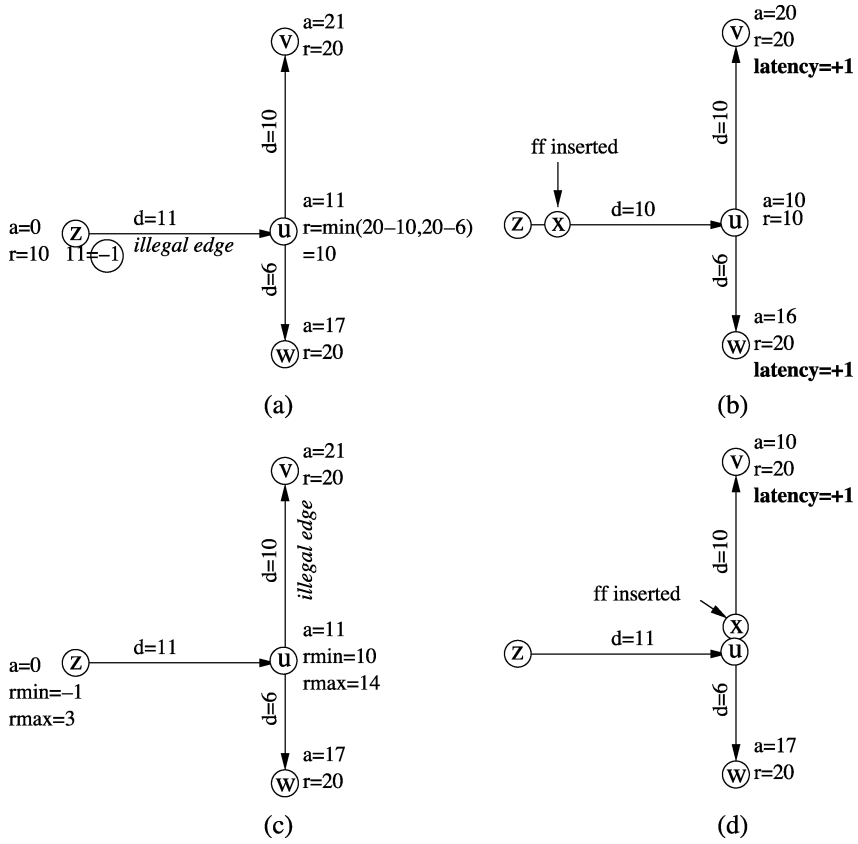


Fig. 7. Illegal edge (z, u) is detected according to the old algorithm [17] because of the timing constraint violation $a(v) > r(v)$ that traces back to the root: $r(z) < 0$ (a). Consequently a flip-flop x is placed along (z, v) (b). According to the new algorithm (z, u) is no more illegal because $r_{\max}(z) > 0$ (c). Thus only (u, v) is marked illegal and x is placed along (u, v) (d).

algorithm with respect to [17] described by the pseudocode `evalBranchLegality`.

```
Function evalBranchLegality (G{V,E})
Input: MRST = G{V,E}, node u
evalBranchLegality(G{V,E},u);
foreach v ∈ Children(u) do
  evalBranchLegality(G{V,E},v);
  d(u,v) = a(v) − a(u) //delay from u to v;
  r_min(u,v) = r_min(v) − d(u,v);
  r_max(u,v) = r_max(v) − d(u,v);
  if r_max(u,v) < 0 then
    mark illegal (u,v);
  else
    mark legal (u,v);
  end
end
r_min(u) = min(r_min(u,v));
r_max(u) = max(r_max(u,v));
```

Instead of just recording the worst required time, both worst $r_{\min}$ and best $r_{\max}$ required times may be annotated at each node of the tree. Therefore an edge (u, v) is judged illegal only if both values $r_{\min}(u)$ and $r_{\max}(u)$ are negative. The latter condition states that no path $p(w)$ exists that ends in leaf-node w and passes through node u such that $a(p) < r(p)$. We formalize it in the following theorem.

Theorem: Given a tree T, a node $u \in T$, and a timing constraint r at the leaves, if $r_{\min}(u) < 0$ and $r_{\max}(u) < 0$, then $!\exists w \in \text{leaves}(T) : u \in p(w)$ and $a(w) < r$.

Proof: It can be demonstrated *ab absurdo*. Suppose that $p(w)$ exists from $\text{root}(T)$ to leaf-node w such that $u \in p(w)$ and $a(w) < r$. Among the various leaves of T that have node u as predecessor there will certainly be leaves $\min$ and $\max$ with corresponding minimum and maximum arrival times $a_{\min}$ and $a_{\max}$ (possibly coincident). The length of the path from u to $\min$ is $|p(u, \min)| = a(\min) - a(u)$ while from u to $\max$ is $|p(u, \max)| = a(\max) - a(u)$. When we trace back from the leaves toward the root we compute minimum and maximum required times at each node by subtracting the length of the path from that node to its children. As a consequence, when we reach the node u, the minimum required time $r_{\min}(u)$ equals the required timing constraint r at node $\max$ minus the length of the path to $\max$ in

$$r_{\min}(u) = r - |p(u, \max)| = r - a(\max) + a(u). \quad (1)$$

Similarly, the maximum required time is

$$r_{\max}(u) = r - |p(u, \min)| = r - a(\min) + a(u). \quad (2)$$

Now, leaf w's arrival time $a(w) \geq a(\min) = r - r_{\max}(u) + a(u)$, where we used (2). If *ab absurdo* $r > a(w) \geq r - r_{\max}(u) + a(u)$ it follows that, by rearranging the inequality, $r_{\max}(u) \geq a(u) \geq 0$, which is contrary to the hypothesis. □

When we trace back to the root node T, if at least one of $r_{\min}(T)$ and $r_{\max}(T)$ is positive, then there exists *at least* one legal path. That is, when node u is the root, the inverse of the previous theorem holds. This is detailed in the following corollary.

Corollary: Given a tree T, and a timing constraint r at the leaves of T, if and only if $r_{\min}(T) < 0$ and $r_{\max}(T) < 0$, then $\nexists\, w \in leaves(T) : a(w) < r$.

Proof: The "if" part is the same of the previous proof with $u = T$. As for the "only if" part suppose that one between $r_{\min}(T)$ and $r_{\max}(T)$ is ≥ 0. Let us take $r_{\max}(T)$. According to (2) there is at least one minimum length path with the arrival time $a_{\min}$ such that $r_{\max}(T) = r - |p(T, \min)| = r - a(\min) + a(T) = r - a(\min)$, because the arrival time $a(T) = 0$ (beginning of the tree). From $r_{\max}(T) \geq 0$ it follows that $a(\min) \leq r$. Therefore, there is at least one path which is totally legal. □

To summarize the consequences of the theorem and its corollary, when both $r_{\min}(T)$ and $r_{\max}(T)$ are <0, none of the paths are already legal and we can proceed with the legalization algorithm described in the following. If one of them is ≥ 0, at least one legal path exists that does not need flip-flops for legalization. In the latter case, to complete the legality check, the tree has to be visited once again in search of those nodes whose arrival time does not respect the maximum required time. Therefore, if there still remains some edge (u, v) with $a(v) > r_{\max}(v)$, then (u, v) is marked illegal. This is exemplified in Fig. 7(c), where we compare the new legality check method with the previous method in [17] depicted in Fig. 7(a). Min and Max required times at node z are (−1,3). With the method in Fig. 7(a), the edge (z, u) is marked illegal while in Fig. 7(c) it is not, because path to leaf-node w is already legal. Therefore the tree is visited and the edge (u, v) is marked illegal because $a(v) = 21 > r = 20$. Consequently, the flip-flop x is added in (u, v) in Fig. 7(d), while in Fig. 7(b) it was placed in (z, u). The latency at node w is not increased. As we will see in Section VI, this leads to an improvement in throughput with respect to the results shown in [17].

After the legality check, the legalization consists in the following steps repeated from the leaves toward the root until the whole tree is legal: 1) place flip-flop in a suitable location within an illegal branch and 2) re-evaluate the legality from the source to the remaining sinks and to the added flip-flop.

Once the feasible region is found, the correct position has to be determined. Let (u, v) be the suitable branch. The flip-flop is placed by evaluating the point in (u, v) whose arrival time equals the latch required time. The new point is added to the MRST. For our experiments we used a geometric delay model for $d(u, v)$ based on a linear function (then invertible) of the simple wirelength $l(u, v)$. Although this might be inaccurate, we judge it suitable for the purpose of this work [7], [19].

Once the flip-flop is inserted, delay and legality from the root to the leaves or to the added flip-flops are recomputed, again assuming that the best buffer assignment is performed. The subtrees following the inserted flip-flops are already legal by construction. The algorithm for legalization is described by function `legalize`. The complexity of the function, which is basically a depth-first search on the net graph, is $O(V + E)$ in the best case, if everything is already legal, and $O(V^2 + VE)$ in the worst case, if all branches are illegal.

```
Function legalize (G{V, E})
Input: MRST = G{V_in, E_in}, node u
Output: MRST = G{V_out, E_out}
legalize(G{V, E}, u);
V = V_in, E = E_in;
foreach v ∈ Children(u) do
    legalize(G{V, E}, v);
  if (u, v) is illegal then
    find location n in (u, v) such that a(n)−
    r(n) = 0;
    V = V∪{n}, E = E∪(u, n)∪(n, v), E = E\(u, v);
    reapply the delay model to the modified
    tree;
    evalBranchLegalityNew(G{V, E}, root);
    legalize(G{V, E}, root);
  end
end
return G{V, E}
```

VI. Experimental Results

A. Experimental Setting

We integrated the algorithms described in the previous sections into an existing publicly available simulated annealing floorplanner based on the sequence pair representation PARQUET (see [30]). On the floorplanner side, our modifications consisted of empowering the existing framework by making it possible to deal with pin directions (which is fundamental for our method, but immaterial for normal floorplan problems); compute short loops, exact and heuristic throughput; and add one optimization mode which, in analogy with the corresponding wirelength optimization, uses a weighted sum of area and throughput cost. Even if this tool is far from a fully engineered implementation of the method, we believe that the results reported below help understanding the original features of the problem we introduced.

B. Benchmarking

A somewhat sensitive issue is that of which benchmarks to choose in order to validate our approach. Classical [Microelectronics Center of North Carolina (MCNC)] and even more recent benchmark suites (such as the Gigascale Silicon Research Center (GSRC) series [29]) lack a fundamental feature for our purposes: clear direction information. Even when this information is actually present (as in the *.nets* format of GSRC series) all pins are marked as "bidirectional," which is practically useless for our purposes. To be more precise, even if a net is actually a bidirectional bus, it is never the case that the channel of communication between the two or more blocks it connects is accessible during the same clock period; considering it to be so, besides substantially restricting the achievable optimization (all bidirectional connections, treated this way, amount to the shortest possible loop of two blocks, thus automatically fixing at 0.5 the maximum achievable throughput when 1 delay is present), misrepresents the functional situation. For this reason,

TABLE I
FLOORPLANNING RESULTS

Bench.Perc	Opt Area			Opt WL			Opt Thr			Opt Thr+WL		
	Area	WL(/1000)	Thr	Area(%)	WL(/1000)	Thr	Area	WL(/1000)	Thr	Area	WL(/1000)	Thr
ami33.30	4.53/2.82	106/92	0.78/0.75	11.1/8.31	45.4/41.8	0.64/0.57	12.3/10.0	83.9/74.66	0.56/0.5	13.4/9.0	73.0/65.7	0.58/0.5
ami33.50	4.88/3.63	108/88	0.67/0.6	12.9/8.38	44.7/40.7	0.49/0.37	13.1/9.57	81.2/77.0	0.425/0.2	12.5/8.3	77/71	0.44/0.33
ami33.70	4.62/3.72	106/84	0.54/0.5	13.2/10.2	44.1/41.0	0.296/0	13.6/7.59	83.6/78.0	0.253/0	11.4/7.2	76.0/64.3	0.28/0.2
ami33.100	4.50/3.01	102/85	0.47/0.33	12.1/8.94	43.1/41.2	0.091/0	11.7/7.98	81.8/74.9	0.08/0	12.6/10.0	74.1/62.3	0.09/0
ami49.30	4.31/3.88	1,908/1,738	0.812/0.789	11.8/8.76	718/643	0.685/0.642	9.98/7.50	1,211/1,071	0.689/0.65	11.9/8.39	1051/973	0.72/0.69
ami49.50	4.39/3.37	1,846/1,679	0.707/0.666	12.5/8.67	718/632	0.561/0.5	10.3/9.67	1,203/1,100	0.553/0.5	11.0/8.78	1057/947	0.57/0.5
ami49.70	4.43/3.76	1,885/1,613	0.646/0.6	12.1/10.1	689/629	0.446/0.333	11.3/6.07	1,170/954	0.466/0.411	11.1/9.13	1053/893	0.45/0.4
ami49.100	4.24/3.53	1,850/1,700	0.5/0.5	11.7/8.56	719/686	0.305/0.25	11.9/9.52	1,162/992	0.304/0.2	11.7/9.74	1062/964	0.27/0.2
apte.30	3.89/1.09	780/646	0.821/0.785	5.14/3.11	526/507	0.782/0.769	7.88/3.80	708/673	0.752/0.727	7.32/3.42	691/644	0.75/0.7
apte.50	3.14/1.09	801/691	0.725/0.666	7.12/3.11	539/512	0.685/0.666	7.02/3.42	689/657	0.613/0.5	6.84/2.51	692/638	0.598/0.5
apte.70	5.50/1.09	772/693	0.641/0.545	7.60/3.19	539/507	0.57/0.5	6.82/2.10	690/624	0.52/0.5	7.63/6.46	696/652	0.52/0.5
apte.100	1.60/1.09	808/761	0.51/0.5	8.53/3.83	534/507	0.441/0.333	5.47/1.98	726/642	0.391/0.25	7.87/3.91	703/658	0.35/0.25
hp.30	18.4/3.73	274/230	0.799/0.666	16.0/7.88	170/155	0.743/0.666	14.8/3.73	269/221	0.699/0.625	15.7/4.10	237/202	0.73/0.6
hp.50	16.4/3.73	274/214	0.725/0.6	22.7/3.43	167/149	0.638/0.6	19.7/6.60	247/222	0.553/0.4	21.4/7.3	232/201	0.55/0.5
hp.70	13.2/3.09	263/213	0.608/0.5	18.1/8.19	173/166	0.52/0.4	15.9/4.09	274/221	0.44/0.333	16.9/4.0	240/209	0.47/0.33
hp.100	19.0/3.73	269/216	0.493/0.333	20.0/3.43	167/154	0.373/0	21.0/5.72	246/197	0.283/0	20.0/13.8	252/212	0.22/0
xerox.30	5.92/2.42	738/680	0.841/0.818	10.6/5.88	437/336	0.807/0.772	9.44/5.21	625/569	0.801/0.781	7.67/5.18	645/586	0.8/0.77
xerox.50	5.39/3.75	760/658	0.751/0.714	10.7/7.48	384/333	0.677/0.642	9.26/5.88	642/534	0.703/0.666	9.84/7.65	657/562	0.7/0.66
xerox.70	6.65/3.81	743/608	0.655/0.6	10.0/5.25	425/333	0.592/0.5	8.00/5.04	627/494	0.566/0.5	10.2/6.44	635/503	0.56/0.5
xerox.100	5.11/3.14	774/713	0.5/0.5	10.7/5.33	392/323	0.49/0.4	8.38/2.63	634/539	0.5/0.5	9.29/5.89	689/629	0.5/0.5
n10.30	5.73/4.14	63/59	0.756/0.692	8.78/5.34	54/51	0.750/0.7	11.2/6.16	59/54	0.695/0.636	12.0/6.1	57.7/54.2	0.68/0.64
n10.50	5.97/4.11	62/59	0.570/0.5	8.15/5.41	54/52	0.573/0.5	8.48/3.97	59/53	0.54/0.5	12.5/8.3	59.5/54	0.52/0.4
n10.70	6.16/4.07	64/59	0.520/0.454	8.93/6.21	54/52	0.477/0.333	10.7/7.16	611/56	0.402/0.25	11.6/9.47	56.7/55.7	0.44/0.29
n10.100	5.92/4.70	63/59	0.411/0.2	9.96/6.46	54/51	0.445/0.2	11.7/8.89	59/54	0.235/0.166	10.6/7.12	57.9/52.3	0.23/0
n30.30	4.84/3.11	186/177	0.775/0.75	8.45/7.18	166/162	0.771/0.75	10.4/8.49	178/174	0.680/0.636	9.95/8.14	183/174	0.74/0.71
n30.50	4.91/3.60	185/181	0.634/0.571	8.66/6.41	163/160	0.618/0.54	9.87/6.75	179/172	0.543/0.5	9.8/7.77	180/169	0.57/0.5
n30.70	5.05/3.92	185/177	0.532/0.5	8.96/6.16	164/159	0.502/0.5	10.2/8.28	178/168	0.426/0.375	10.4/6.67	177/167	0.45/0.4
n30.100	4.78/3.94	184/177	0.442/0.333	8.20/5.87	163/158	0.408/0.25	9.72/6.94	177/163	0.278/0.2	9.33/6.53	177/173	0.33/0.25
n50.30	4.52/3.87	235/227	0.786/0.733	8.65/6.49	187/180	0.709/0.666	9.48/7.23	220/210	0.671/0.636	10.1/8.28	217/201	0.71/0.66
n50.50	4.56/3.92	235/227	0.668/0.636	7.70/6.44	191/185	0.549/0.5	9.83/6.72	221/214	0.50/0.473	9.97/8.86	225/216	0.56/0.5
n50.70	4.48/3.63	232/221	0.527/0.5	7.94/6.90	188/182	0.49/0.4	9.59/6.92	221/214	0.419/0.375	9.6/5.55	219/207	0.45/0.36
n50.100	4.57/3.69	236/228	0.458/0.384	7.92/6.30	191/185	0.331/0.166	9.57/7.15	222/216	0.217/0.133	10.1/7.75	221/213	0.31/0.22
n100.30	5.50/4.38	397/388	0.763/0.703	9.84/7.34	320/308	0.693/0.642	11.4/9.73	382/366	0.573/0.5	12.5/10.4	303/284	0.65/0.58
n100.50	5.69/5.08	398/392	0.645/0.571	10.0/7.48	319/304	0.509/0.444	10.3/8.63	383/376	0.439/0.375	12.9/9.82	304/293	0.47/0.4
n100.70	5.38/4.85	395/387	0.534/0.5	9.67/7.51	321/313	0.420/0.230	10.7/9.08	378/372	0.248/0.2	13.2/10.0	303/287	0.34/0.25
n100.100	5.20/4.61	396/386	0.465/0.35	9.64/7.73	315/306	0.246/0.142	11.2/9.08	379/368	0.079/0	12.8/10.3	300/283	0.21/0.13

we decided to assign directions to all pins. We first adopted the strategy suggested in [28] of making the last pin of a net the net source. This strategy worked perfectly for the four GSRC benchmarks we analyzed (n10, n30, n50, and n100), but it was not viable for the classical MCNC benchmarks (ami33, ami49, apte, hp, and xerox), because this would have led to loopless networks. For this reason, we assigned randomly one pin per net as its source. This solution, of course, could generate unnaturally easy or hard problems, and to test it, we introduced one more benchmark that was built *ad hoc* from the IBM placement suite where, on the contrary, pin directions of gates are indicated. However, the mixed standard-cell block structure of the benchmarks, made it impossible to employ them in a methodology such as the one herein proposed, that presupposes large blocks with fairly constrained mutual communication. Therefore, we took one of the simplest benchmarks of the suite (ibm02), partitioned it with hMetis [31] into a netlist of 30 large blocks with their own input and output terminals, and randomly picked 200 nets, preserving their direction information. With the obtained block level netlist, we proceeded as for the other benchmarks. We also experimented with the use of synthetic benchmarks like GNL [32], proceeding in a similar way, partitioning flat gate level netlists into larger blocks. The results obtained (i.e., achieved throughput, area, wirelength) were comparable with all other results obtained with the classic floorplanning benchmarks.

C. Results for Floorplanning

In this section, we present our results in terms of throughput-driven floorplanning. The main aim of our work was not to compare with other floorplanning strategies, but rather to give some indications of how the throughput problem could be partially solved with the use of a cost function like the one described in Section IV. For this reason, we launched our floorplanner with four different cost functions for each case under consideration: Area, Area + Half Perimeter Wire Length, Area + Heuristic Throughput cost, and Area + Wire Length + Heuristic Throughput. For the area and wirelength, and area and throughput optimizations, we used a balanced weighting of the two costs (half each), while in the three-objective case we used proportions of 0.4, 0.3, and 0.3, respectively. We ran each case ten times, with a time limit of 60 s on a Linux machine with a Pentium III processor at 1.4 GHz, with 700 MB of RAM and 512 kB of cache. The usage of a time limit means that the number of iterations varies for the three optimizations. However, the wirelength and throughput optimizations have a similar execution time per iteration. Incidentally, this confirms the efficiency of the cost function and its easy integrability with existing frameworks. For each case, we gathered three average and minimum results for whitespace, wirelength, and degradation of throughput (computed with the exact algorithm). A result of 0.391/0.25, for example, means that the benchmark has an average throughput of 1–0.391 and a maximum throughput of 1–0.25. Please take note that we express throughput in this way throughout the paper and, therefore, a *smaller* value implies *better* performance. Also, for each benchmark, we considered four different critical lengths: 0.3, 0.5, 0.7, and 1.0 times the minimum die edge computed as the square root of the total block area. Results are reported in Table I. Line ami33.50, for example, reports the results for

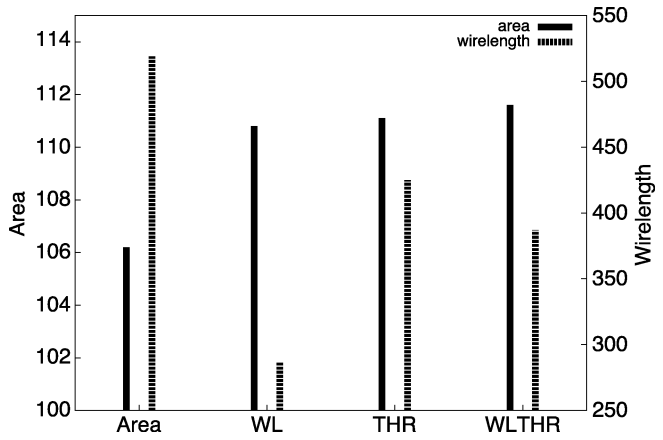


Fig. 8. Average area and wirelength for the various cost functions.

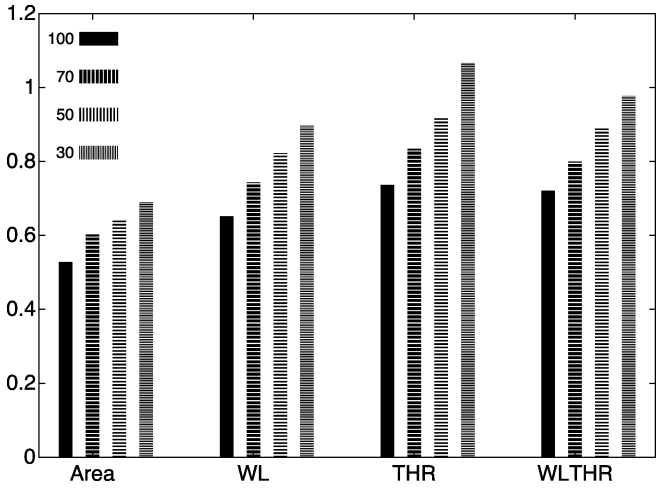


Fig. 9. Data rates for the various cost functions.

benchmark ami33 where critical length is 50% of the minimum floorplan edge. As the dimensions of the blocks are of the essence for this method (i.e., a floorplanning problem with critical length of 30% of the die is bound to be different from an analogous problem with a 100% critical length), the units of the length and areas are different for each example, but are not reported for simplicity.

Fig. 8 shows average area and wirelength under different optimization objectives [area, wirelength (WL), throughput (THR), combined throughput and wirelength (WLTHR)]. The left y axis' units are percentages of the total area of the blocks: An area of 106 represents an average of 6% whitespace. The right y axis' units is 1000 of length units. Fig. 9 shows how data rates (frequency by throughput) are influenced by the various cost functions, depending on the critical length. Arbitrary units are used to fit the figure size.

It is apparent from Fig. 8 that, provided some cost related to interconnect length and/or throughput is taken into consideration, area penalties are similar. On the other hand, throughput comes at the cost of wirelength, for which it can be conveniently traded, as shown by the WLTHR results. Fig. 9 shows an interesting increasing trend for data rates with deeper pipelining (no matter what optimization is used), which will be investigated in Section VI-D.

The results allow us to draw a series of conclusions.

1) THR optimization really achieves better throughput results. Average gain with respect to area and wirelength minimization are 25% and 11%, respectively. If we consider only the gain in the case of the longest critical length, the two gains are 64% and 24%, respectively, thus suggesting the existence of a threshold critical length over which the gain becomes substantial.
2) Wirelength and throughput minimization are goals that are not as closely correlated as it might be thought.
3) Different benchmarks behave differently as long as throughput is concerned, and their difficulty is not a simple function of other features, let alone of their complexity (number of blocks, number of nets).
4) The task is in and by itself inherently difficult. There is no chance of getting an acceptable throughput with area optimization, while wirelength minimization also can lead to highly suboptimal results (see, for example, the case of n100).
5) The last three columns (three-objective optimization) show that sometimes wirelength can be recovered by trading in a small amount of throughput, even though the effect of the combined optimization is benchmark-dependent. In general, we experimented with different weights to the various optimization goals, finding results within the bounds of the corner cases of only wirelength and only throughput minimization. This can help in managing a tradeoff in highly congested designs.

D. Frequency Versus Throughput Considerations

An abstract consideration might be used against any throughput optimizing strategy that aims at increasing frequency through a latency-independent method. It follows these lines: Given a working implementation for which each connection is just shorter than the critical length, at a frequency f_0, another implementation at a double frequency $2f_0$ is possible that introduces one pipelining element on each edge. Then, if the system is not purely feedforward, the theory admits a degradation of throughput of 1/2, thus giving a data rate of $(1/2)2f_0$, neither better nor worse than the original case, with the added disadvantage of requiring a more critical clock distribution network, with all the related issues. So, it is important to see if the doubling of the frequency always calls for a halving of the throughput, or if the optimization process might actually end up with substantially better results.

For these reasons, we took the various benchmarks and made a sweep of the critical length parameter, then graphing the data rate. We repeated the experiment with three different cost functions: The heuristic throughput cost described in Section IV, a quadratic length, and a simplified version of the exact cost, wherein only loops of maximum length 4 are counted. The reason for adding the other two cost functions was also to compare the proposed heuristic with other possible approaches. In the quadratic case, we used as a cost the sum of the squares of the euclidean lengths of the pin-to-pin connections belonging to loops, the rationale being that a quadratic cost will tend to favor shorter

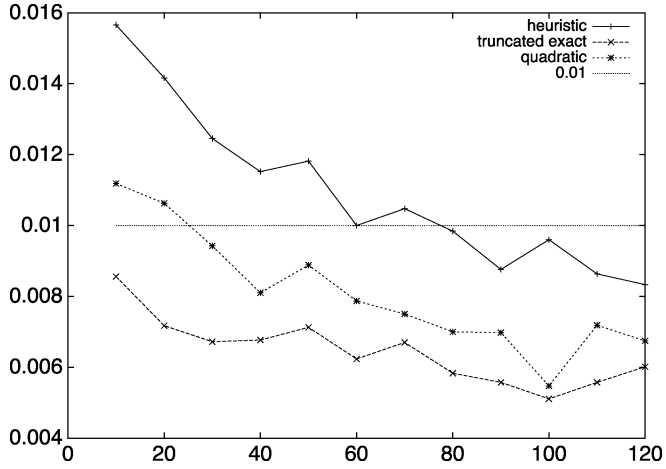


Fig. 10. Overall performance of n100.

connections with respect to a Manhattan distance. The truncated exact cost, on the other hand, tames the complexity of the exact computation by limiting the depth of the search to loops of length 4. The results, for benchmark n100, are shown in Fig. 10, where the x axis represents the critical length (in percentage of the minimum die edge). The values represent the average over five random seed placements using the different values for the critical length. The topmost line is the curve obtained using the heuristic cost that behaves much better than the more "accurate" cost function that characterizes the "reduced exact" curve. Finally, even if the quadratic cost behaves better than the pseudo exact cost, it is still outperformed by the heuristic, which proved to be the best in all cases, for all examples we ran.

It appears that, *ceteris paribus*, reducing the critical length is always beneficial in increasing the overall performance (data rate). This apparently counterintuitive conclusion has to be weighted against the simplifying assumptions herein made: simple inverse proportionality between critical length and frequency, abstraction from detailed circuit implementations, intrinsic logic limitations (it is not possible to achieve arbitrarily high frequencies due to combinational logic delays), clock tree synthesis, and skew control. Nonetheless, the graph shows there is abundant design space to proceed toward faster systems.

E. Results for Tree Flip-Flop Routing

The columns in Table II indexed by *Rout* compare the results obtained after floorplanning for one of the ten trials with those after flip-flop routing. Results are reported for both old [17] and new algorithms. Degradation with the old routing strategy is monotonous, with an average loss of 16%. On the other hand, the new routing algorithm shows a clear improvement, giving a moderate loss of only 6% on average.

The other columns in Table II report the number of pipelining registers after routing, and compare it with two measures: A half perimeter measure of the necessary registers (min HPWL), and the number of FF required by routing each pin-to-pin connection separately (P2P). The first value (min HPWL) tries to capture the result of a good routing, in the sense that, for each multipin net, the half perimeter length is evaluated, and then divided by the critical length. The rationale behind this is that an HPWL is known to be always a correct measure of wirelength for connections with up to three pins, and generally an underestimate for nets with more pins. We point out that, should the source of the net be in the middle of the net's bounding box, a clever routing could give a smaller number of flip-flops (if some of the P2P distances are smaller than the critical length, for example, they will not introduce any pipelining elements). This explains why our routing strategies, in some cases, can produce a smaller number of elements than HPWL. On the other hand, the P2P values are to be an upper bound on the number of registers. Averages in the table are normalized with respect to the ideal throughput and HPWL respectively.

TABLE II
ROUTING RESULTS

Bench.Perc	Rout			# FF		
	min	old	new	HPWL	P2P	new
ami33.30	0.5	0.5	0.5	142	144	144
ami33.50	0.333	0.333	0.333	79	81	81
ami33.70	0.2	0.2	0.2	36	37	36
ami33.100	0	0.2	0	21	22	22
ami49.30	0.7	0.77	0.77	312	348	309
ami49.50	0.57	0.66	0.66	121	121	118
ami49.70	0.44	0.545	0.5	64	70	64
ami49.100	0.25	0.5	0.333	11	19	15
apte.30	0.7	0.785	0.785	189	232	191
apte.50	0.5	0.666	0.625	91	120	96
apte.70	0.4	0.666	0.6	57	69	58
apte.100	0.33	0.625	0.333	23	26	24
hp.30	0.57	0.6	0.6	196	303	200
hp.50	0.5	0.5	0.5	96	137	92
hp.70	0.25	0.333	0.25	61	86	58
hp.100	0.25	0.333	0.25	40	48	35
xerox.30	0.833	0.833	0.833	325	400	351
xerox.50	0.75	0.75	0.75	141	159	159
xerox.70	0.666	0.666	0.666	72	74	71
xerox.100	0.5	0.6	0.5	17	26	21
n10.30	0.75	0.75	0.75	184	193	185
n10.50	0.5	0.5	0.5	73	71	69
n10.70	0.5	0.5	0.5	56	53	53
n10.100	0.25	0.333	0.25	9	8	8
n30.30	0.68	0.68	0.68	685	690	680
n30.50	0.545	0.5625	0.545	343	344	339
n30.70	0.416	0.444	0.444	183	175	175
n30.100	0.222	0.222	0.222	94	90	90
n50.30	0.64	0.65	0.65	940	1009	935
n50.50	0.5	0.5	0.5	477	504	464
n50.70	0.35	0.44	0.36	265	273	252
n50.100	0.18	0.33	0.18	129	129	120
n100.30	0.64	0.64	0.55	2002	2057	1824
n100.50	0.428	0.5	0.5	1011	1018	948
n100.70	0.285	0.4	0.333	623	616	560
n100.100	0.125	0.33	0.2	306	292	276
Average	1	1.16	1.06	1	1.06	0.96

The number of elements is always close to or better than the HPWL values, thus indicating a good overall routing quality; at the same time, especially in the case of short critical length, the savings with respect to the P2P strategy can be substantial (see, for example, n100).

VII. CONCLUSION AND FUTURE WORK

The problem of the insertion of flip-flops in interconnects for wire pipelining in deep-submicron large size integrated circuits

has been tackled in this paper from the perspective of the potential throughput degradation this technique may imply. In order to limit this detrimental countereffect, we inserted a modified cost function in a simulated annealing based floorplanner. Such a new cost metric models the latency of the interconnects as a simple function of the Manhattan distance between two points in a layout and then evaluates the throughput reduction. To the best of our knowledge, this work is the first to systematically address this problem in general and to propose a viable yet effective solution.

The experimental results demonstrate the advantages of the new approach compared to standard cost functions like area or wirelength. The final throughput achieved by the new floorplanner has been tested using a flip-flop routing algorithm suitably developed and used as a back-end tool. The results consistently agreed on all benchmarks.

Future work will address many effects not considered here. On the optimization side, the floorplanner framework will be better explored to fine tune its performance and allow treatment of soft blocks. As for the router, congestion and various blockages shall be considered for a more realistic evaluation. Another key point is the adoption of a better delay model. Finally, an effort shall be made to gather more appropriate benchmark date.

REFERENCES

[1] The International Technology Roadmap for Semiconductors, SIA, 2003.

[2] L. P. Carloni *et al.*, "A methodology for "correct-by-construction" latency insensitive design," in *Proc. ICCAD*, 1999, pp. 309–315.

[3] T. Chelcea and S. M. Nowick, "Robust interfaces for mixed-timing systems with application to latency-insensitive protocols," in *Proc. DAC*, 2001, pp. 21–26.

[4] L. P. Carloni and A. L. Sangiovanni-Vincentelli, "Performance analysis and optimization of latency insensitive protocols," in *Proc. DAC*, 2000, pp. 361–367.

[5] L. P. Carloni *et al.*, "Theory of latency-insensitive design," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 20, no. 9, pp. 1059–1076, Sep. 2001.

[6] M. R. Casu and L. Macchiarulo, "A new approach to latency insensitive design," presented at the DAC, San Diego, CA, Jun. 2004, pp. 576–581.

[7] J. Cong and S. K. Lim, "Physical planning with retiming," in *Proc. ICCAD*, 2000, pp. 2–7.

[8] R. Lu and C.-K. Koh, "Interconnect planning with local area constrained retiming," in *Proc. DATE*, 2003, pp. 442–447.

[9] C. Chu *et al.*, "Retiming with interconnect and gate delay," in *Proc. ICCAD*, 2003, pp. 221–226.

[10] D. K. Tong and E. F. Y. Young, "Performance-driven register insertion in placement," in *Proc. ISPD*, 2004, pp. 53–60.

[11] L. P. Carloni and A. L. Sangiovanni-Vincentelli, "Combining retiming and recycling to optimize the performance of synchronous circuits," in *Proc. SBCCI*, 2003, pp. 47–52.

[12] L. P. P. P. Van Ginneken, "Buffer placement in distributed RC-tree networks for minimal Elmore delay," in *Proc. ISCC*, 1990, pp. 865–868.

[13] R. Lu *et al.*, "Flip-flop and repeater insertion for early interconnect planning," in *Proc. DATE*, 2002, pp. 690–695.

[14] S. Hassoun *et al.*, "Optimal buffered routing path constructions for single and multiple clock domain systems," in *Proc. ICCAD*, 2002, pp. 247–253.

[15] P. Cocchini, "Concurrent flip-flop and repeater insertion for high performance integrated circuits," in *Proc. ICCAD*, 2002, pp. 268–273.

[16] ——, "A methodology for optimal repeater insertion in pipelined interconnects," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 32, no. 12, pp. 1613–1624, Dec. 2003.

[17] M. R. Casu and L. Macchiarulo, "Floorplanning for throughput," in *Proc. ISPD*, 2004, pp. 62–69.

[18] H. D. Lin and D. G. Messerschmitt, "Improving the iteration bound of finite state machines," in *Proc. ISCAS*, vol. 3, 1989, pp. 1923–1928.

[19] J. Cong and D. Z. Pan, "Interconnect delay estimation models for synthesis and design planning," in *Proc. ASP-DAC*, 1999, pp. 97–100.

[20] C. J. Alpert *et al.*, "Porosity aware buffered steiner tree construction," in *Proc. ISPD*, 2003, pp. 158–165.

[21] H. Zhou, "Efficient steiner tree construction based on spanning graphs," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 23, no. 5, pp. 704–710, May 2004.

[22] F. R. Boyer *et al.*, "Optimal design of synchronous circuits using software pipelining techniques," *ACM TODAES*, vol. 6, no. 4, pp. 516–532, 2001.

[23] X. Hong *et al.*, "Corner block list: an effective and efficient topological representation of nonslicing floorplan," in *Proc. ICCAD*, Nov. 5–9, 2000, pp. 8–12.

[24] P.-N. Guo *et al.*, "An O-tree representation of nonslicing floorplan and its applications," in *Proc. DAC*, Jun. 21–25, 1999, pp. 268–273.

[25] H. Murata *et al.*, "VLSI module placement based on rectangle-packing by the sequence-pair," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 15, no. 12, pp. 1518–1524, Dec.

[26] M. Moe and H. Schmit, "Floorplanning of pipelined array modules using sequence pairs," in *Proc. ISPD*, 2003, pp. 143–150.

[27] C. W. Sham and E. F. Y. Young, "Routability driven floorplanner with buffer block planning," in *Proc. ISPD*, 2002, pp. 50–55.

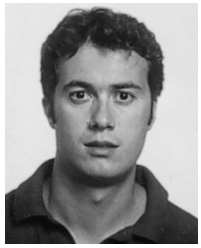
[28] Y. Ma *et al.*, "An integrated floorplanning with an efficient buffer planning algorithm," in *Proc. ISPD'03*, pp. 136–142.

[29] (2003) GSRC T2 Bookshelf at UC Santa Cruz. [Online] Available: www.cse.ucsc.edu/research/surf/GSRC/progress.html

[30] Paraquet: Fixed-Outline Floorplanner. [Online] Available: http://vlsicad.eecs.umich.edu/BK/parquet/

[31] hMETIS: Serial Hypergraph & Circuit Partitioning. [Online] Available: http://www-users.cs.umn.edu/~karypis/metis/hmetis/

[32] D. Stroobandt *et al.*, "Generating synthetic benchmark circuits for evaluating CAD tools," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 19, no. 9, pp. 1011–1022, Sep. 2000.



Mario R. Casu (M'05) received the Electronics Engineer degree (*summa cum laude*) and the Ph.D. degree from the Politecnico di Torino, Torino, Italy, in 1998 and 2002, respectively.

He is currently an Assistant Professor in the Department of Electronics, Politecnico di Torino. In 2001, he was with ST Microelectronics Central Research and Development, where he worked on SRAM development and CMOS library characterization on a partially depleted 0.13-μm SOI technology. He has coauthored several papers published in international conferences proceedings and journals. His research interests are in the field of CMOS circuits modeling and design and interconnect related circuit, architecture, and computer-aided design-level issues in advanced deep submicron technologies.

Luca Macchiarulo received the M.S. and Ph.D. degrees from the Politecnico di Torino, Torino, Italy, in 1995 and 1999, respectively.

He was a Postdoctoral Researcher at the University of California, Santa Barbara, and the Politecnico di Torino. He is currently an Assistant Professor in the Department of Electrical Engineering, University of Hawaii, Honolulu. His research interests are in high-speed interconnect design and analysis, architectural and layout optimization of throughput, field programmable gate array and regular logic synthesis and optimization, and physical design issues of ultra-deep-submicron scaling.