Floorplanning for Throughput

Mario R. Casu
Politecnico di Torino/CERCOMM
Dipartimento di Elettronica
C.so Duca degli Abruzzi, 24
1-10129, Torino, ITALY
mario.casu@polito.it

Luca Macchiarulo
Politecnico di Torino
Dipartimento di Elettronica
C.so Duca degli Abruzzi, 24
1-10129, Torino, ITALY
luca.macchiarulo@polito.it

ABSTRACT

Large Systems-on-Chip (SoC) in advanced technologies run at such high frequencies that the time-of-flight of signals connecting two distant pins in the layout can be higher than the clock period. In order to avoid performance penalties wires are pipelined using latches. However the throughput of the system may be altered due to the presence of loops in the logic netlist. In this paper we address the problem of floorplanning a large design with interconnect pipelining and inserting throughput in the cost function of the floorplanning algorithm. The throughput results obtained on a series of benchmarks are then validated using a simple router that places flipflops along the nets built with a heuristical minimum rectilinear steiner tree.

Categories and Subject Descriptors
B.7.2 [Integrated Circuits]: Design Aids—Placement and Routing

General Terms
Algorithms

Keywords
Systems-on-chip, Floorplanning, Wire pipelining, Throughput

1. INTRODUCTION

The design of large Systems-on-Chip (SoC) poses a number of challenges the designers have to face throughout the entire design flow. On the one hand there is a strong need to reuse existing designs in the form of intellectual properties (IP) in order to speed up design time and decrease the impact of non-recurring engineering costs. Available IP’s range from soft blocks, where in general a RT level netlist is available, to hard blocks where everything is fixed up to mask design. On the other hand, one has to make the SoCs run at a given target frequency and at a given “number of operations” per cycle (i.e. throughput), while not worsening power, signal integrity, reliability, testability and so on. The latter objectives are often in contrast with the previous cost-driven requirements. Fulfilling performance specifications has become an extremely hard task when dealing with the huge complexity of today’s large design in ultra deep submicron technologies. It is clear that traditional approaches looking first at RTL, architectures, gate level designs do not work anymore.

Large designs in advanced technologies may work at such high clock frequencies that many cycles may be needed for a signal to travel across two points within the die, even considering optimal sizing and buffering of the wires. Since slowing down the frequency is in contrast with performance-driven requirements, the more natural approach to deal with the problem is trying to hide the interconnect latency with wire pipelining, i.e. segmenting the line and inserting latches or flip-flops. Since the behavior of a sequential system may be affected by this insertion, some researchers try to formulate this problem as a retiming problem where sequential elements are moved away from logic blocks and relocated within wires. This approach is already formidable if one has the control of what is the block. If the designers sees only a “black-box” IP, plus timing and electrical specifications, the retiming approach may be impossible altogether.

Other researchers have showed that it is possible to make a system latency insensitive so that wires can be pipelined, provided that a minimum of protocol, with low overhead, is established in order to implement the communication.

In this paper we suppose to deal with SoC blocks like IP that cannot be explored nor modified and that constitute a system whose functionality is represented by a logic connection of them (netlist of blocks). After a review of the related work in the field, we briefly show how the throughput behavior of a system is affected when such connections are pipelined, provided that functionality is guaranteed some way. Then we propose a throughput driven floorplanning where throughput enters in the cost function of a simulated annealing based tool. The results of the floorplanning are also validated using a simple router that connects blocks while inserting pipeline stages. Floorplanning benchmarks have been used to accomplish the work of verifying the validity of our approach. We conclude by suggesting improvements and summarizing the work done so far.

2. RELATED WORK

The problem of how to insert pipeline stages in long in-
terconnects has been recently addressed in its various aspects. At the system level, L. Carloni and A. Sangiovanni-Vincentelli report a methodology ([1][2][3]) that allows the preservation of functionality of a system which is assumed working under zero wire delay constraint and it is made working with a certain amount of added wire latency. We assume these papers as a reference to show how systems can be made working with wire pipelining without RTL modifications. Other methods can be certainly conceived, but for our purposes what is important is that flip-flop or latches addition in wires can be done without modifying the system. While functionality is preserved, the number of valid operations per cycle, alias its throughput, may be affected, as explained in the following section.

For what concerns the physical synthesis aspects, J. Cong and S.K. Lim [4], and more recently R. Lu and C-K. Koh [5], formulate the insertion of pipeline stages in wires as a retiming problem, therefore assuming that latches or flip-flops can be moved from logic blocks to interconnects. However, as already noticed in the introduction, while optimal, retiming can only be applied if logic blocks are described at least at RT level. We address the case of IP’s that designers are likely to use in a plug-and-play fashion.

Other recent papers on the physical design side of the problem show how the modern techniques developed for the routing and buffer insertion, all derived in the end from the classic paper by L. Van Ginneken [6] can be adapted to routing path construction and simultaneous flipflop and repeater insertion [7][8][9]. In [7] and [8] only pin-to-pin connections are considered while this paper also consider the case of multiple connections, as in [9]. All these papers, however do not address floorplanning issues and how netlist connections translate on final performance achievable from the buffer-flipflop routing after a given placement. As far as we know, these issues are addressed for the first time in this work.

3. THROUGHPUT EVALUATION

In order to be able to assess the performance of a design in presence of added latency, it is necessary to consider a structural constraint. This kind of constraint emerged successively in various disguises in the history of high throughput design (for example see the iteration bound of Messerschmitt et al.[12]), but it consists in a relatively simple hard limit: cycles introduce functional dependency in which over time the connectivity and the block in order to guarantee the same functionality. This small example shows that simply adding a pipelining element where needed to solve long path problems is impossible as it modifies the functionality of the system (mostly unpredictably). To preserve it, we are left with two alternatives: play with existing memory elements or 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 the blocks connected, 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 it reacts only to valid signals. This requires an overall clock scheduling that will activate the various unit in a coordinated way. It is not the scope of this paper to discuss how this scheduling is going to be enforced, neither whether a global control (as proposed by [13]) or a distributed approach (as in the Latency Insensitive Protocols by Carloni and Sangiovanni-Vincentelli [3]) is to be preferred: it is sufficient to say that such a control is possible. With this in mind, let’s follow how such a strategy could be employed to “legalize” the situation of figure 1. At time 1, block A should be prevented from operating to avoid logical errors: let’s suppose then that the protocol gates its clock. 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 $A_2$ (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 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, t.i. is kept from changing according to its logical evolution, from time to time. In this example, both A’s and B’s output (though not at the same time) are stalled once every 3 clock cycles. It is possible to prove that not only this result is possible (t.i., there exist a schedule that will allow a throughput of
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, without the added element would have taken 2 clock cycles (the number of elements in the loop) to complete will now take $2+1$ clock cycles, thus degrading the average performance to $\frac{3}{2}$. A similar line of thought brings to the following conclusion, which we don’t prove formally for the sake of brevity:

[Image of diagrams showing loop structures]

**Theorem:** A strongly connected component of a block netlist always admits a schedule which allows an average throughput of $\frac{m}{m+n}$ 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.

Looking at the case of figure 2, where invalid data are marked as $n$, even though the right loop might proceed at an average throughput of $\frac{3}{2}$, forces it to slow down accordingly.

It might be asked whether only loops can contribute to throughput degradation: Reconvergent fanouts 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. And, given the physical problem to solve which they are introduced, while 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 won’t 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) s.t. the system can operate at such theoretically maximum throughput.

In conclusion, a block netlist’ best possible average throughput is the minimum value of $\frac{m}{m+n}$ 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 of a block netlist, it leaves unsatisfied the practical issue on 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 upfront 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 were to traverse all possible loops, a hopeless task for all but the simplest cases given the exponential nature of loop enumeration.

Fortunately, the situation can be handled due to a lucky situation. Let’s 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 $\frac{1}{m+n}$ where $m$ and $n$ are the blocks and the delays on the path respectively.

Then the key observation is expressed by the following

**Theorem:** Given two paths with common start and end node, 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 omitted)

In practical terms this means that, while traversing the netlist in search of the maximum cost $\frac{1}{m+n}$ (which is to say one minus the minimum throughput) it is not necessary to record each path through which we got to visit a single node, but at most as many as the possible path lengths (limited by the overall number of nodes). Therefore, the maximum loop that passes through a given node can be assessed in a time which is bounded by the number of edges multiplied by the number of nodes, and the overall problem has a complexity of $O(VN^2)$ ($V$ edges and $N$ nodes). The same observation allows us to write an algorithm implementing the throughput evaluation of which we will omit all details. We simply report the performance of our algorithm on a series of benchmarks in table 1.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>#blocks</th>
<th>#nets</th>
<th>CPU Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ami43</td>
<td>33</td>
<td>123</td>
<td>4.01</td>
</tr>
<tr>
<td>ami49</td>
<td>49</td>
<td>397</td>
<td>11.00</td>
</tr>
<tr>
<td>apte</td>
<td>9</td>
<td>94</td>
<td>2.25</td>
</tr>
<tr>
<td>hp</td>
<td>11</td>
<td>65</td>
<td>1.01</td>
</tr>
<tr>
<td>xerox</td>
<td>10</td>
<td>181</td>
<td>13.5</td>
</tr>
<tr>
<td>n10</td>
<td>10</td>
<td>118</td>
<td>0.58</td>
</tr>
<tr>
<td>n30</td>
<td>30</td>
<td>349</td>
<td>65.2</td>
</tr>
<tr>
<td>n50</td>
<td>50</td>
<td>485</td>
<td>17.2</td>
</tr>
<tr>
<td>n100</td>
<td>100</td>
<td>885</td>
<td>557</td>
</tr>
</tbody>
</table>

Table 1: Throughput Evaluation CPU times.

The above algorithm is sufficiently efficient to allow a performance evaluation, but still way too slow to be included in a loop of an iterative method of optimization as one needed in floorplanning. For this reason in the following section we will describe the alternative we pursued.

4. FLOORPLANNING FOR THROUGHPUT

The high computational cost of the exact evaluation of a netlist of block’s 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 reason is that simulated annealing allows easy integration in existing frameworks of more or less exotic cost functions, and multi-objective optimization is taken care of in a relatively smooth way. The availability of efficient and compact representations for floorplans (slicing and
not) containing all feasible solutions with a minimal redundancy (corner block list [14], O-tree [15], Sequence Pair [16], to name just a few) has made simulated annealing the election method to implement optimizations on more diverse cost functions (such as array modules [17], routability and buffer planning [18], congestion control). Now, there are three conditions which 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. An annealing process will be 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 (and exactly 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 increasing abruptly 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 could at the same time follow closely the real throughput trend (possibly monotonical with it) and be 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 a function computed on the entire netlist. It had to depend strongly on the possible presence of delay on critical loops. A function that satisfies all previous conditions 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, based on the current position of the blocks and the relative positions of the pins we evaluate the Manhattan distance between the pins.
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.

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 (it implicitly decomposes all multipin nets 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 3) 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 for this choice, and experimental results will show that the results are actually consistent with the expected optimizations.

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

![Figure 3: Correlation Between exact and heuristic throughput evaluation.](image)

Our experiments showed a pretty high correlation between this cost and the exact throughput. In figure 3 the abscissas report the values of the heuristic cost, while the ordinates those of the exact throughput on a floorplan of an example (GSRC benchmark [20] n100) with many different values of the maximal distance. 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 match. Even though the measure cannot be used for an absolute evaluation of a system’s throughput (as heuristic cost depends on the net numbers and their detailed topology structure), the figure shows that relative correlations are well mapped, so that the cost can be used inside an optimization engine.

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

We conclude this section with two observations:

- 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: in fact only the nets that are “meaningful” for throughput optimization are taken into account, and they are normally, especially towards the end of the annealing schedule, a small fraction of the total. Also, even long nets that are not included in short loops (for example, nets from and to the terminals) don’t contribute to the cost. The difference w.r.t. a typical minimal wirelength optimization will show clearly in the experimental results.
5. FLIP-FLOP ROUTING

In order to validate the results of the throughput based floor planning, we have 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 use of an accurate timing model even though we are perfectly aware of the inaccuracies of this approach. Blockages are also not considered. This assumption is justified by the already claimed need for exploiting the porosity in large IP’s layout for buffer insertion [11] 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 net nodes to be connected that are internally represented as a graph $G = \{V,E\}$. The first step is the construction of the minimum spanning tree (MST) using the rectilinear (a.k.a. Manhattan) distance. The MST is the input to an algorithm that approximates the minimum rectilinear Steiner tree (MRST) using a heuristic that adds nodes to the graph in order to reduce the wirelength by sharing common segments (and then also buffers and flip-flops if needed) among the sinks of the net. At each step of the algorithm a node $u \in V$ that has not been connected yet is treated. The direction for next routing $D \in \Delta = \{N,E,S,W\}$ is chosen based on the four set of sinks $\{N(u),E(u),S(u),W(u)\}$ each one containing nodes $v \in V$ that still have to be connected and whose position is respectively north, east, south and west with respect to $u$. 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 $(x_a,y_a)$ and $(x_i,y_i)$ are the position of nodes $u$ and $v_i$ respectively, and $v_i \in N(u)$ (i.e. $y_i > y_a, \forall i \in [1,|N(u)|]$), then $\lambda_N = \min(y_i - y_a), v_i \in N(u)$. The direction $D \in \Delta$ is then chosen according to $\lambda_D \cdot |D(u)| = \max\{\lambda_N \cdot |N(u)|,\lambda_E \cdot |E(u)|,\lambda_S \cdot |S(u)|,\lambda_W \cdot |W(u)|\}$ that maximizes the segment sharing. Another node $n$ is added. If direction was north, then $(x_n,n) = (x_a,y_a + \lambda_N)$. 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. The algorithm is graphically exemplified in figure 4 and formally described with the pseudo-code buildSteinerTree.

With this approach, the added nodes stay on the Hanan’s grid built based on the position of the original nodes.

Once the routing has been defined, a delay model is applied to the tree in order to evaluate if timing constraints set at its leaves are respected. We suppose that an optimal buffer placement 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 where needed. 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 order to meet timing constraints. 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 $r(v) - d(u, v)$.

Function buildSteinerTree($u,\{V_in,E_in\}$)

Input: $u, \text{MST} = \{V_{in},E_{in}\}$

Output: $\text{MST} = \{V_{out},E_{out}\}$

buildSteinerTree($u,\{V_in,E_in\}$);
$V = V_{in}$;
mark $u$ as visited;
while $\exists v \in V : (u,v) \in E$ do
build sets $N(u),E(u),S(u),W(u)$;
find minimum segments $\lambda_N,\lambda_E,\lambda_S,\lambda_W$;
select direction $D \in \{N,E,S,W\}$ such that $\lambda_D \cdot |D(u)|$ is maximum;
add node $n$, distance $\lambda_D$, direction $D: V = V \cup \{n\}$;
foreach $v \in D(u)$ do
add edge from $n$ to $v: E = E \cup \{(n,v)\}$;
remove edge from $u$ to $v: E = E \setminus \{(u,v)\}$;
end
buildSteinerTree($n,V,E$);
end
return $V_{out} = V, E_{out} = E$.

Figure 4: Application of the heuristical Minimum Rectilinear Steiner Tree (MRST) construction.
An illegal branch followed by a legal branch is a feasible region for flipflop placement. The evaluation of the legality is implemented with a depth-first search of the MRST as described in the code 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(u, v) = r(v) − d(u, v);
if r(u, v) < 0 then
   mark illegal (u, v);
else
   mark legal (u, v);
end
end
if ∃v ∈ Children(u) : r(u, v) ≥ 0 then
   r(u) = min(r(u, v) ≥ 0);
else
   r(u) = min(r(u, v));
end

The legalization consists in the following steps repeated from the leaves toward the root until all the tree is legal:
- place flipflop in a suitable location within an illegal branch;
- re-evaluate the legality from the source to the remaining sinks and to the added flipflop.

Once the feasible region is found, the correct position has to be determined. Let (u, v) be the suitable branch and path p(v) = p(u) ∪ (u, v) the set of contiguous edges from the root to v. Suppose that node w exists that shares part of its p(w) with p(u), i.e. ∃z ∈ V such that p(z) = p(v) ∩ p(w) or in other words z ∈ predecessors(v) ∩ predecessors(w). Suppose p(u) is legal. The question is now if placing the flipflop in branch (u, v) will make path p(w) illegal. If the flipflop input capacitance is less than the downstream equivalent capacitance seen in the insertion point p_n in edge (u, v) before insertion, the arrival time after insertion in p_n ∈ (u, v) will be certainly smaller. This is certainly the case for long interconnects as the ones we are investigating. Therefore the flipflop placement will not make path p(w) illegal.

The correct position of the flipflop is determined by evaluating the point in (u, v) whose arrival time equals the latch required time. Since adding the new element actually alters the arrival time, a few iterations with the delay model may be required, or if analytical and invertible functions are used they can be employed in order to find the location in a closed form. 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). Albeit this might be inaccurate, we judge it suitable for the purpose of this work $[4][10]^1$.

This model does not consider the fanout load, but the buffers that decouple from the load reduces the inaccuracy.

Once the flipflop is inserted, the delay and legality of the tree from the root to the leaves are recomputed, again assuming that the best buffer assignment is performed. The algorithm for legalization described by function legalize:

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 legal 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);
   re-apply the delay model to the modified tree;
   evalBranchLegality(G(V, E), root);
   legalize(G(V, E), root);
end
return G(V, E)

6. EXPERIMENTAL RESULTS

6.1 Experimental Setting

In order to test our optimization technique and gather some statistics on the features of this problem, we implemented the algorithms described in previous sections integrating it in an existing publicly available simulated annealing floorplanner based on the sequence pair representation: PARQUET (see [21]). On the floorplanner side our modifications consisted in empowering the existing framework by making it possible to deal with pin directions (which is fundamental for our method, but normally immaterial for normal benchmark problems), compute short loops, exact throughput, heuristic throughput, and add one optimization mode which, in analogy with the corresponding wirelength optimization, uses a weighted sum of area and throughput cost. The main purpose of this evaluation was to compare area, wirelength and throughput optimization in a homogeneous way, rather than propose a fully engineered implementation of the method. In this respect we left over many possible improvements like an exhaustive search on annealing schedules or a fine tuning of parameters. However, we believe that the results below reported represent a faithful image of the original features of the problem we introduced. We leave other improvements to future work.

6.2 Benchmarking

A somewhat sensible issue is that of the benchmarks to choose in order to validate our approach. Classical (MCNC) and even more recent benchmark suites (such as GSRC series [20]), which are well suited for all normal benchmarking, lack a fundamental feature for our purposes: a 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 at the same time (that is, during the same clock period) and considering it, besides making it impossible to achieve any optimization goal (all bidirectional connection, treated this way, amount to the shortest possible loop of two blocks, thus automatically fixing at 0.5 the maximum achievable throughput when delays are present), misrepresents the functional situation. For this reason, we decided to assign direction to the pins. We first adopted the strategy suggested in [19], of making the last pin of a net the net source. This strategy worked perfectly for the 4 GSRC benchmarks we analyzed (n10, n30, n50 and n100), but it was unviable for the classical MCNC benchmarks (ami33, ami49, apte, hp and xerox). The reason is that (at least in the distribution we used) the blocks connected by each net were listed always with the same ordering, fact that automatically made the assumption of last block as a source derive a loopless netlist, for which the cost is of course always 0. For this reason we assigned randomly one pin per net as its source.

6.3 Results for Floorplanning

We present in this section our results in terms of throughput-driven floorplanning. The main aim of our work was not that of comparing with other floorplanning strategies, but rather to give some indications on how the throughput problem could be partially relieved with the use of a cost function like the one described in section 4. For this reason we launched our floorplanner with three different cost functions for each case under consideration: Area, Area + Half Perimeter Wire Length and Area + Heuristic Throughput cost. For the area and wirelength, and area and throughput optimizations, we used a balanced weighting of the two costs (half each). We ran each case 10 times, with a time limit of 60 seconds on a Linux machine with a Pentium III processor at 1.4 GHz, with 700 Mbytes of RAM memory and 512 KBytes of cache. The usage of a time limit means that the number of iterations varies for the three optimization. However, the wirelength optimization and the throughput optimization have a similar execution time per iteration. We gathered for each case three results: average and smallest whitespace, average and smallest wirelength, average and smallest 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 maximum throughput of 1-0.25. 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 2. The line ami33.50, for example reports the floorplanning results for benchmark ami33 where critical length is taken to be 50% of the minimum floorplan edge. As the dimension of the blocks are of the essence for the method (t.i., 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, and not reported for simplicity. The 100% case, however, corresponds to the benchmark descriptions in the literature.

The results allow us to draw a series of conclusions:

- Throughput optimization really achieves better results in terms of throughput. The gain w.r.t. wirelength minimization is in average 11%, while the gain w.r.t. area is 25% again in average over all the experiments. If we consider only the gain in the case of the longest critical length, though the two gains are 24% and 64% respectively, thus suggesting the existence of a threshold length about which the gain becomes substantial.
- Wirelength and throughput minimization are goals which are not so correlated as it might be thought.
- Different benchmarks behave differently as long as throughput is concerned, and their difficulty is not a simple measure of other features, let alone their complexity (number of blocks, number of nets).
- The task is in and by itself inherently difficult: there is no chance of getting an acceptable throughput with area optimization, while also wirelength minimization can lead to highly suboptimal results (such as the case of n100, for example).
- Even if it depends on the benchmark considered, there is a value for critical dimension such that below it, no matter what kind of cost function, the throughput cannot be optimized, while above that threshold the three methods differ.

6.4 Results for Tree Routing

The last column in table 2 compares the throughput results obtained after floorplanning for one of the ten trials with results after routing and flip-flop placement. Since the routing algorithm described in section 5 tries to minimize wirelength, and so the number of flipflops, this objective may lead to a further throughput degradation because the number of flipflops evaluated within the floorplan optimization framework is a simple function of the minimum rectilinear distance. Degradation is monotonous with maximum and average loss of 34% and 15% respectively. We underline the fact that such a degradation monotonously affects all floorplanning results, in the sense that a wirelength optimized floorplan will have its throughput degraded as well as its throughput-related counterpart (data not shown).

7. CONCLUSIONS AND FUTURE WORK

In this work we considered the effect of the insertion of flip-flops in interconnects for wire pipelining under the point of view of possible throughput degradation. We inserted a modified cost function in a simulated annealing based floorplanner in order to take into account the modified throughput by considering the latency of interconnect as a simple function of the Manhattan distance between two points in a layout. Results confirm the capability of the cost function to increase performance even considering the effects of routing.

Our future works will take different directions. On the optimization side we will explore in depth the floorplanner framework to fine tune its performance and allow treatment of soft blocks. As for the router, we will take into account congestion and various blockages. Another key point is the insertion of a better delay model. While a linear model is sufficiently accurate for optimal repeated interconnects, it fails to model unbuffered wires or suboptimal buffer assignments.

Finally, an effort should be made in the direction of appropriate benchmark definition.
### 8. REFERENCES


