Buffering Interconnect for Multicore Processor Designs

Yifang Liu, Jiang Hu, Senior Member, IEEE, and Weiping Shi, Senior Member, IEEE

Abstract—Recently, the microprocessor industry is headed in the direction of multicore designs in order to continue the chip performance growth. We investigate buffer insertion, which is a critical timing optimization technique, in the context of an industrial multicore processor design methodology. Different from the conventional formulation, buffer insertion in this case requires a single solution to accommodate different scenarios, since each core has its own parameters. If conventional buffer insertion is performed for each scenario separately, there may be a different solution corresponding to each of these scenarios. A straightforward approach is to judiciously select a solution from one scenario and apply it to all the scenarios. However, a good solution for one scenario may be a poor one for another. We propose several algorithmic techniques for solving these multiscenario buffer insertion problems. Compared with a straightforward extension of the conventional buffer insertion, our algorithm can improve slack by 20–280 ps for max-slack solutions. For min-cost solutions, our algorithm causes no timing violation, while the extended conventional buffering results in 35% timing violations. Moreover, the computation speed of our algorithm is faster.

Index Terms—Buffer insertion, interconnect, multicore processor designs, optimization.

I. INTRODUCTION

FOR SEVERAL decades, the microprocessor industry had pushed for increasingly high chip frequency, which is driven by the market need for performance growth and is enabled by VLSI technology scaling. When VLSI feature size shrinks to the nanometer regime, chip power density approaches its fundamental limit. Shackled by tight power constraint, the performance gain from the frequency increase is diminishing. This fact has forced microprocessor companies to make a strategic move—pursuing multicore design. Nowadays, multicore designs have become common in almost all kinds of processor applications: servers, desktops, and laptops.

Since multicore is an architectural approach, most of related research works are naturally focused on the architecture level. In this paper, we will show that multicore designs also imply circuit-level issues and discuss such a problem. Specifically, we will investigate how to perform buffer insertion in the context of multicore processor designs. Buffer insertion is a powerful technique for optimizing interconnect, which is a well-known bottleneck of chip performance [6]. In traditional microprocessor and ASIC designs, buffer insertion solutions are often found by using variations of van Ginneken’s [1] or Lillis’ [2] algorithm. Given an interconnect tree and candidate buffer locations on it, van Ginneken’s algorithm [1] propagates a set of partial solutions, each of which consists of buffer assignments for nodes in the subtree rooted at a given node, from the leaf nodes toward the source and eventually finds the best timing solution at the source. Based on van Ginneken’s algorithm, Lillis made an important implementation with efficient data structures, which can deliver a set of solutions with timing-cost tradeoffs in pseudopolynomial runtime. In both algorithms, inferior partial solutions are pruned during the propagation so that the computation runtime is reasonable.

In an industrial multicore design methodology, the design of the cores and chip integration are performed at about the same time. An interconnect net typically has multiple instances for different cores. In Fig. 1(a), the net has one instance for core \( A \) and another instance for core \( B \). For each net, the interconnect inside the cores must be identical for each instance since these cores have to be the same. Even a small difference may lead to large changes through ripple effect. However, the interconnect outside any core may vary as it is difficult for chip integration to enforce core-based regularity to a large extent outside the cores. For the example in Fig. 1(a), the required arrival time (RAT) \( q \) at sink \( u \) is 70 ps for core \( A \) and 50 ps for core \( B \). Since these cores have to be identical, the buffer insertion solutions inside the cores should be identical for all these instances. Therefore, we need to find a single buffer insertion solution that accommodates different scenarios outside the cores. This is a key difference from the conventional buffer insertion problem.

Note that multicore processor design is different from the case of IP-core design where the knowledge on the prospective applications is limited. In contrast, in a microprocessor company, designers of the cores work side-by-side with chip integration team and therefore know the out-core environment, at least approximately. The often-aggressive performance goal in microprocessor designs requires that such knowledge is utilized for performance improvement rather than being neglected. Certain interfaces, such as buffers at the boundary of cores, can decouple the designs of in-core and out-core portions and therefore make the problem much easier to solve. However, such interface or boundary buffers may result in large area/power overhead if they are deployed without scrutiny.

For the aforementioned problem of buffering for multicore designs, a straightforward approach is to run Ginneken–Lillis
algorithms separately on each instance and then pick one to be
shared by all instances. However, the algorithm that runs on
one instance may prune a partial solution which is preferred
in another instance. It is also possible that the solution sets at
the sources of these instances have no overlap. If we pick the
optimal solution for one instance and apply it to other instances,
timing violations may occur on the other instances. For the
example in Fig. 1(a), the minimum cost solution that satisfies
timing constraint for core A is to insert a buffer between
Steiner node and sink u. However, this solution causes negative
slack of −15 ps at sink u for core B. A single solution that
satisfies the timing constraints for both core A and core B is
to insert buffers at both branches as in Fig. 1(b). This solution
is pruned out when Ginneken–Lillis algorithm is performed for
either core A or core B separately.

In this paper, we make significant extensions to Ginneken–
Lillis algorithms such that a single buffering solution can be
found to accommodate different scenarios for different cores.
We consider the following differences among instances in
nets with identical topology: 1) RAT for sinks outside cores;
2) sink capacitance for sinks outside cores; 3) arrival time (AT)
for drivers outside cores; and 4) driver resistance for drivers
outside cores. Aside from the characteristic of the source and
sinks, the topology of nets can be different among instances.
We consider topological difference in three parts of a net:
1) upper part of a net; 2) lower part of a net; and 3) middle
part of a net. Note that the AT at the driver is not important
for conventional buffer insertion problems as changing AT does
not affect the relative timing criticality among all candidate
solutions. For multicore designs, different AT implies different
timing criticalities among the instances even when they share
the same delay and the same sink RATs. In this paper, we deal
with the overall net solutions, which are composed of instance
solutions, instead of handling the instance solutions separately.
In van Ginneken's algorithm [1], each solution is characterized
in a 2-D space of timing and downstream load. In Lillis’
algorithm [2], each solution is characterized in a 3-D space.
In general, the computation complexity grows rapidly with the
number of dimensions. If there are h cores (instances), a direct
extension of Ginneken–Lillis algorithms results in solutions in
a 2h + 1 dimension space, which is too high for direct solving.
We propose several techniques to reduce the dimensionality
of the solution space with limited degradation to solution quality.

To the best of our knowledge, this is the first work on the
multicore buffering problem. Compared with a straightforward
extension of conventional buffering, our algorithm can improve
the slack by 20–280 ps for max-slack solutions. For min-cost
solutions, our algorithm causes no timing violation while the
extended conventional buffering results in 35% timing viola-
tions. Moreover, our algorithm runs faster by a factor linear to
the number of cores.

II. TRADITIONAL BUFFERING

The traditional buffer insertion problem is solved with
Ginneken–Lillis style algorithms. Given the layout of a Steiner
tree with candidate buffering locations, Ginneken–Lillis algo-
rithms propagate candidate solutions from the sinks toward the
source.

Each node vi is associated with a solution set S(vi) which
includes candidate solutions propagated there. Each candidate
solution is characterized by a triple (c(vi), q(vi), w(vi)), where
the value of c(vi) denotes the downstream load capacitance,
q(vi) represents the RAT, and w(vi) is the cost for the solution.
At each node, a candidate solution is formed with a combi-
nation of child solutions (solutions of immediate downstream
nodes) and the buffer choice at node vi. The cost w(vi) is the
summation of child node costs and the cost of the buffer at
node vi. The RAT at node vi is given by decreasing the minimum
child solution RAT by wire and the buffer delay at node vi, i.e.,

\[ q(v_i) = \min_{v_j \in \text{children}(v_i)} q(v_j) - \text{delay(wire}_{ij}\text{)} - \text{delay(buffer}_{i}) \]

(1)

where children(vi) is the set of child nodes of node vi and
the buffer delay is zero when no buffer is inserted at node vi, i.e.,
delay(\emptyset) = 0.
Solutions that lead to worse source RAT and cost than other solutions are inferior solutions. In order to reduce the size of solution sets, inferior solutions are kept from entering new solution sets and pruned from existing solution sets.

A basic inferior solution detection rule [2] is as follows.

Property 1: Given two solutions \( s = (c, q, w) \) and \( s' = (c', q', w') \) in a node's solution set, \( s \) is inferior to \( s' \) if the following condition holds:

\[
w \geq w', \quad c \geq c', \quad \text{and} \quad q \leq q'.
\]

By implementing this rule with an efficient data structure, Lillis' algorithm [2] runs in pseudopolynomial time.

III. MULTICORE BUFFERING PROBLEM

In buffering for multicore designs, a net has multiple instances, each of which corresponds to one core. For example, in Fig. 1, the net has two instances, one for core A and the other for core B. In the general case, some parts of the net are inside cores and the other parts are outside cores. In Fig. 1, sink \( u \) and \( v \) are outside cores, and the source \( s \) is inside cores. It is also likely that the source is outside cores. Since the cores are usually identical, the in-core part of the net is the same in all instances. However, the out-core part may vary from one instance to another, depending on the design of outside cores. In this paper, we begin with investigating the cases with difference in the source and sinks of the net:

1) RAT of sinks outside cores;
2) capacitance of sinks outside cores;
3) AT of source outside cores;
4) driver resistance of source outside cores.

We next examine the cases with topology difference in the out-core parts. In this case, not only the timing and RC characteristics of out-core source and sinks vary among instances but also the topology and wire characteristics are different. There are three common scenarios of topology differences.

1) Nets with upper cores: The source and some of its continuous downstream nodes are inside the cores, while downstream nodes starting from some points are outside the cores.

2) Nets with lower cores: The sinks and some of their continuous upstream nodes are inside the cores, while upstream nodes starting from some points are outside the cores.

3) Nets with sandwich cores: Some continuous downstream nodes of the source and upstream nodes of the sinks are inside the cores, while the nodes in between are outside the cores. A special case of sandwich cores is nets with identical source and lower cores, which have upper cores consisting of the source only.

Fig. 2 shows upper and lower cores with three toy nets. There are two instances of each net in which the cores are indicated by shaded areas. Topology outside the cores may be different. For all three nets, node \( v_5' \) is connected to different nodes in instance (1) and (2). Net (a) with upper cores also has different length of wires connected to sinks. Net (b) with lower cores has different length of wire connected to the source. Net (c) is an example of sandwich cores with topology difference only in the middle. For brevity, we omit the discussion of another type of net with identical cores in the middle and different topology on parts both above and below the cores. This kind of net can be handled by a technique very similar to the one used for sandwich cores, which combines the procedures used on upper-core nets and lower-core nets.

Because the cores are identical, buffering solutions for all instances have to be the same for the in-core part. Such a net with identical cores is called a multicore net. Traditional buffering algorithms such as Ginneken–Lillis algorithms [1], [2] are applicable to only individual instances. If they are carried out on each instance separately, it is difficult to ensure that all instances share the same buffering solution. Sometimes, the solution sets at the sources of different instances have no overlap, and consequently, it is impossible to find a common solution among these sets. A straightforward method is to perform the Ginneken–Lillis algorithm on one instance and apply the solution of this instance to all the other instances. However, a good solution for one instance may be poor for other instances due to the difference on out-core parts.

In multicore designs, the buffering problem is how to find a single solution that can accommodate all instances with...
differences. We consider two common problem formulations stated as follows.

1) **Max-slack problem:** Find a single buffering solution for all instances of the in-core part of a net such that the minimum slack among all instances is maximized.

2) **Min-cost subject to timing constraint problem:** Find a single buffering solution for all instances of the in-core part of a net such that the total buffering cost is minimized while the timing constraints of all instances are satisfied.

We define **critical slack** as the minimum slack among all instances of a net. Therefore, the max-slack problem is to maximize the critical slack of a net.

### IV. ALGORITHM OVERVIEW

Our algorithm propagates candidate solutions from sinks toward sources like the dynamic programming in Ginneken–Lillis algorithms. A key idea for solving the multicore buffering problem is to propagate the same in-core buffering solutions among all instances. In other words, we propagate net solutions inside the cores. If there are $h$ cores, an in-core net solution consists of $h$ identical buffering solutions, one for each instance. The solutions in conventional buffering can be treated as instance solutions. Therefore, we can also say that an in-core net solution is composed by multiple identical instance solutions. For example, Fig. 1(a) shows one net solution composed of two instance solutions.

Although the instance solutions of an in-core net solution are identical, they may have different RATs and/or different load capacitances due to the differences on the out-core parts. Therefore, a net solution is characterized in a $2h+1$ dimensional space for an $h$-core design: $h$ dimensions for load capacitances, another $h$ dimensions for RATs, and the other dimension for the buffering cost. Therefore, a solution at node $v_i$ is characterized by

$$c(v_i, \phi_1), q(v_i, \phi_1), \ldots, c(v_i, \phi_h), q(v_i, \phi_h), w(v_i)$$

where $c(v_i, \phi_j)$ and $q(v_i, \phi_j)$ are the load capacitance and RAT of node $v_i$ in instance $\phi_j$, $j \in \{1, 2, \ldots, h\}$, respectively.

The framework of our algorithm is similar to Ginneken–Lillis algorithms except that we propagate net solutions instead of instance solutions inside the cores. It is very difficult to perform pruning for the net solutions since their dimension is significantly higher than that of conventional buffering. Consequently, the algorithm on a $2h+1$ dimensional solution space can be very slow. A main focus and contribution of this paper is to represent the $2h+1$ dimensional problem by a 3-D problem, which well preserves the quality of the solution yet has a computational complexity similar to conventional buffering. Such a transformation is achieved through the concept of critical components, which is introduced next.

**Definition 1:** The critical component of a net solution at node $v_i$ in multicore buffering problem is a triple

$$s(v_i) = (\hat{c}(v_i), \hat{q}(v_i), \hat{w}(v_i))$$

where

$$\hat{w}(v_i) = w(v_i) = \sum_{k=1}^{h} w(v_i, \phi_k)$$

is the summation of the cost over all instances and $\hat{q}(v_i)$ is the minimum RAT over all instances, i.e.,

$$\hat{q}(v_i) = \min_{k=1}^{h} q(v_i, \phi_k).$$

The first element $\hat{c}(v_i)$ is a capacitance value extracted in different ways under different conditions.

Section V handles cases with identical topology of the net, where net solutions are propagated throughout the net. In the cases where net topology is different outside the cores, our method propagates net solutions inside the cores and instance solutions outside the cores. Section VI addresses topology differences in three common cases.

### V. NETS OF IDENTICAL TOPOLOGY AND DIFFERENT SOURCE AND SINKS

In Section V-A, we introduce the algorithm for a relatively simple case where only the sink RATs are different among instances. The algorithm for this case is still optimal. In Section V-B, we discuss more general and more difficult cases where each sink has different load capacitances and different RATs in different instances. The differences on source AT and driver resistance are addressed in Section V-C.

#### A. Cases With Only Sink RAT Differences

In this section, we introduce an algorithm for a multicore buffering problem where only sink RAT may be different among instances. Then, without loss of generality, we can assume that the ATs at the sources of all instances are zero (the difference of AT at the sources will be discussed in Section V-C). In this case, the slack of an instance $\phi_k$ is equal to the RAT $q(v_0, \phi_k)$ at the source node $v_0$. In addition, the critical slack is equal to the RAT of the critical component at the source node $v_0$. Then, the two multicore buffering problems formulated in Section III can be restated as max-slack problem

$$\text{maximize : } \min_{k=1}^{h} q(v_0, \phi_k) = \hat{q}(v_0) \quad (6)$$

and min-cost subject to timing constraint problem

$$\text{minimize : } w(v_0) = \hat{w}(v_0)$$

subject to : $\min_{k=1}^{h} q(v_0, \phi_k) = \hat{q}(v_0) \geq 0. \quad (7)$

Next, we introduce the notion of a complementary solution to assist the presentation of properties and algorithms. A complementary solution at node $v_i$, denoted by $u(v_i)$, is a solution that contains buffer choices at all nodes in the net other than those in the subtree rooted at node $v_i$. In the example shown in Fig. 3, a partial solution at node $v_5$, $s(v_5)$ is composed of all the buffer
choices at nodes in the subtree rooted at $v_5$ (outside the shaded area). A complementary solution at node $v_5$, $u(v_5)$ consists of all the buffer choices at nodes outside the subtree rooted at $v_5$ (in the shaded area). Obviously, with a pair of partial and complementary solution, $s(v_i)$ and $u(v_i)$ at node $v_i$, $u(v_i) \cup s(v_i)$ forms an overall solution, which contains buffer choices for all nodes in the net. Denote by $U(v_i)$ the set of all possible complementary solutions at node $v_i$. Complementary solutions build up a bridge between partial and overall solutions, which enables some general rules to identify inferior solutions.

**Property 2:** Given two partial solutions $s(v_i)$ and $s'(v_i)$ at some node $v_i$, $s(v_i)$ is inferior to $s'(v_i)$ if the following condition holds:

$$\forall u(v_i) \in U(v_i),$$
$$u(v_i) \cup s(v_i) \text{ is an overall solution inferior to } u(v_i) \cup s'(v_i).$$

Based on the aforementioned general properties of inferior solutions, we develop more specific properties for the multicore net. The case of equal sink capacitance is investigated in the rest of this section.

**Definition 2:** An iso-cap net is a multicore net, such that each of its sinks has equal capacitance across all instances, i.e.,

$$\forall v_i \in \{v|v \text{ is a sink}\}, \forall k \in \{1, \ldots, h\}, C(v_i, \phi_k) = C(v_i)$$

where $\{1, \ldots, h\}$ is the set of instance indices and $C(v_i)$ is a constant for sink $v_i$.

In addition, we define iso-cap solution at node $v_i$ to be a solution with equal downstream capacitances across all instances, i.e., $\forall k \in \{1, \ldots, h\}, c(v_i, \phi_k) = c(v_i)$. It is obvious that every solution at any node in an iso-cap net is an iso-cap solution.

For solutions in iso-cap nets, we set the first element of their critical component to be the unique downstream capacitance across all instances, i.e., for a solution at node $v_i$

$$\hat{c}(v_i) = c(v_i, \phi_k), \text{ for any } k \in \{1, \ldots, h\}. \tag{8}$$

The following property of inferior solutions in iso-cap nets is based on this critical component assignment.

**Property 3:** Given two partial solutions

$$s(v_i) = (\hat{c}(v_i), \hat{q}(v_i), \hat{w}(v_i))$$
$$s'(v_i) = (\hat{c}'(v_i), \hat{q}'(v_i), \hat{w}'(v_i))$$

at node $v_i$, $q$ in an iso-cap net, $s(v_i)$ is inferior to $s'(v_i)$ if the following condition holds:

$$\hat{w}(v_i) \geq \hat{w}'(v_i), \hat{c}(v_i) \geq \hat{c}'(v_i), \text{ and } \hat{q}(v_i) \leq \hat{q}'(v_i). \tag{9}$$

**Proof:** See the Appendix.

Based on Property 3, a minor modification to Ginneken–Lillis algorithms is needed to accommodate buffer insertion in iso-cap nets. At the beginning, the critical components of all sinks are extracted. Then, the solutions are propagated from sinks to the source with critical component $(\hat{c}, \hat{q}, \hat{w})$ being used as $(c, q, w)$ of each solution in conventional buffering algorithm. This way, the algorithm reduces the computational complexity to a single core problem by performing combination and pruning in 3-D solution space, while the algorithm guarantees the optimal overall net solution of iso-cap nets.

An example of two merging solution sets in a net with two cores is shown in Fig. 4. Each bracket embraces the set of partial solutions at the point. Each solution is expressed in the form of $(c(\phi_1), q(\phi_1), c(\phi_2), q(\phi_2), w)$, where critical component of the solution is underlined. For instance, the critical component of the solution $(0.9, 3.1, 0.9, 2.5, 4.4)$ at node $v_3$ is composed of its identical load capacitance 0.9, minimum RAT 2.5 (among 3.1 and 2.5 for the two instances, respectively), and the total cost 4.4. Although values other than the critical components do not play any role in the algorithm, we keep them in the solution form for a clear presentation of how critical component works.

In this small example, we demonstrate the solution propagation procedure at each node by showing how solutions at $v_1$ and $v_2$ are merged and buffered to form solutions at $v_3$. Brackets next to edges $(v_1, v_3)$ and $(v_2, v_3)$ contain solutions at $v_1$ and $v_2$, respectively, which include buffer and wire delays above them. In this example, there are two solutions at $v_1$. The solution with critical component $(0.5, 2.1, 3.1)$ is inferior to the other solution with critical component $(0.3, 2.5, 2.5)$, due to its larger capacitance 0.5, smaller RAT 2.1, and larger cost 3.1. Thus, the inferior solution is pruned out. Solutions at $v_1$ and $v_2$ are merged as in Lillis’ algorithm to create the
solution \((0.9, 3.1, 0.9, 2.5, 4.4)\) in front of \(v_3\), which is then expanded by two buffering options at \(v_2\) to form the two solutions embraced in the bracket next to the edge above \(v_3\). Solutions are propagated through every node in the net as in the aforementioned procedure. The overall solutions of the net are obtained at the source in the end.

**B. Cases With Different Sink Cap and RAT**

In this section, we first introduce the notion of iso-cap frontline and then use it to categorize nodes in the net and process them with different techniques.

The frontline at any moment during solution propagation is composed of processed nodes whose parent nodes have not been processed yet. Fig. 5 shows an example in a very small net with two cores. Each solution set is shown with a box next to its corresponding node. A node without a box next to it indicates that it has never been updated (not processed yet). Each solution set is shown with a box next to its corresponding node. A node without a box next to it indicates that it has never been updated (not processed yet).

The example gives a snapshot of solution sets in the middle of solution propagation. Nodes \(v_5\) and \(v_6\) have been processed, while their parent nodes \(v_7\) and \(v_8\) have not yet undergone it. Therefore, nodes \(v_5\) and \(v_6\) form the frontline at the moment.

In nets with different sink capacitances across different instances, Property 3 does not apply directly. However, this does not mean that the dimension of solutions should rise to \(2h + 1\). We have two techniques to reduce solution dimension in this case. Recall that a solution at node \(v_1\) with buffer inserted at it must be an iso-cap solution, although this is not the only way for a solution to be iso-cap. If all solutions at the frontline nodes are iso-cap solutions, then the part of net above the propagation frontline can be treated as an iso-cap net. We solve this part with help similar to Property 3 as in iso-cap nets. For the part of net below the front-line, we use a heuristic to extract critical components of solutions, thus reducing the solution space to three dimensions while well preserving the quality of solution.

We call a solution set an iso-cap solution set if all solutions in it are iso-cap. If all frontline solution sets are iso-cap, then the frontline is iso-cap. The part of the net composed of the nodes on the iso-cap frontline and all their upstream nodes is an iso-cap subnet. An iso-cap subnet has similar property as an iso-cap net. All solutions in iso-cap subnet are iso-cap. Again, for these solutions, we set the first element of their critical components to their unique downstream capacitance, i.e.,

\[
\hat{c} = c(\phi_k), \quad \text{with any } k \in \{1, \ldots, h\}. \tag{10}
\]

**Property 4:** Given two partial solutions

\[
s(v_i) = (\hat{c}(v_i), \hat{q}(v_i), \hat{w}(v_i)),
\]

\[
s'(v_i) = (\hat{c}'(v_i), \hat{q}'(v_i), \hat{w}'(v_i))
\]

at node \(v_i\) in an iso-cap subnet, \(s(v_i)\) is inferior to \(s'(v_i)\) if the following condition holds:

\[
\hat{w}(v_i) \geq \hat{w}'(v_i), \quad \hat{c}(v_i) \geq \hat{c}'(v_i), \quad \text{and } \hat{q}(v_i) \leq \hat{q}'(v_i). \tag{11}
\]

**Proof:** The proof is similar to that of Property 3. \(\square\)

During the solution propagation procedure, once the frontline is detected to be iso-cap, we use Property 4 to solve the iso-cap subnet as in an iso-cap net. For the downstream nodes below the iso-cap frontline, solutions are represented with critical components extracted by another dimension reduction method.

For nodes below the iso-cap frontline, it is very difficult to determine if one solution is strictly inferior to another only based on its min-RAT \((\hat{q})\), cost \((\hat{w})\), and load capacitances. Since the solutions are not iso-cap, a solution with a small RAT can have a low load capacitance; furthermore, solutions at other nodes also have various load capacitances across different instances. Therefore, when two solutions are propagated upstream and are combined with solutions from other nodes, their relative order in min-RAT may change along the way. One way to compensate this variance is to incorporate the prediction of future capacitance difference into critical component extraction, but it is computationally expensive and ineffective. Therefore, we choose a simple yet effective method. Considering the worst case, in each solution, the maximum load capacitance over all instances is used as an estimation of the load for critical component, i.e., the first element of critical component is set as

\[
\hat{c} = \max_{k=1}^h c(\phi_k). \tag{12}
\]

For example, node \(v_1\) in Fig. 5 has its critical component as \((0.3, 2.9, 2.0)\). Extracting this critical component and using it as \((c, q, w)\) of a solution in conventional buffering, our algorithm propagates solutions from the sinks to the iso-cap frontline in a slightly different style from Ginneken–Lillis algorithm, which is introduced next.

**Algorithm 1** Procedure: `update_isocap(v_i)`

\[
\text{Procedure: } \text{update_isocap}(v_i)
\]

\[
\text{if } \text{status}[v_i] \text{ has never been updated then}
\]

\[
\text{if } \forall v_j \in \text{children}(v_i), \text{status}[v_j] = \text{isocap} \text{ then}
\]

\[
\text{status}[v_i] \leftarrow \text{isocap}
\]

\[
\text{update_isocap(parent}(v_i))
\]

\[
\text{end if}
\]

\[
\text{end if}
\]
There is one more issue to resolve: an efficient method to detect iso-cap frontline during the propagation procedure. The iso-cap frontline is better to be detected as early as possible, and we do not want to run the detection over and over during solution propagation. Our method runs in linear time and ensures the detection of iso-cap frontline at the earliest moment. The basic idea is as follows. When a new solution set is found to be iso-cap for the first time, it updates the status of the corresponding node and its upstream nodes (if applicable) to iso-cap. If a newly created solution set is not iso-cap, the statuses of the corresponding node and all nodes in the subtree rooted at its parent node are set to noniso-cap (if they have not been processed yet). The current frontline is detected to be iso-cap if the root’s status becomes iso-cap at any moment. Algorithms 1 and 2 give the outlines of status updating procedures called each time a new solution set is created. Initially, the statuses of all nodes in the net are set to noniso-cap. Their statuses are updated during the propagation procedure. In order to ensure the earliest detection of iso-cap frontline, only those noniso-cap parents of frontline nodes can be the next node to be processed. This is a key difference of iso-cap frontline detection from a traditional propagation procedure.

Algorithm 2 Procedure: update_noniso-cap(v_i)

if status[v_i] has never been updated by update_noniso-cap
then
    status[v_i] ← noniso-cap
    for all v_j ∈ children(v_i) do
        update_noniso-cap(v_j)
    end for
end if

C. Handling Source Difference

In the multicore buffering problem, the signal AT at the source may be different for different instances. Consequently, one buffering solution may result in different slacks at different instances, even if these instances share the same sink cap, the same sink RAT, and the same driver resistance. It is difficult to directly consider the effect of source AT in a bottom-up dynamic programming. Our approach is to shift the source AT and all sink RATs of each instance such that the source AT of every instance is aligned to zero. For each instance, if we shift its source AT and all of its RATs by the same amount, the slack, which is the difference between AT and RAT, is not affected. Therefore, the problem after the AT alignment is equivalent to the original problem. We construct an equivalent dual problem as follows.

Given a net of h instances, the source ATs (g(v_0, φ_k)) and sink RATs of each instance φ_k are shifted as follows:

\[ g(v_0, φ_k) = 0 \]

∀v_i ∈ {v | v is a sink node}, g(v_i, φ_k) = g(v_i, φ_k) − g(v_0, φ_k).

(13)

Similarly, the driver resistance may be different in different instances. As a result, a partial net solution inferior to another in the cores may become superior to the later net solution when propagated to the source, if the source is outside the cores. Solution inferiority reversion like this does not happen with small driver resistance difference. If the difference between driver resistances in different instances is large, we insert a buffer at the source to make the resistance at the driver identical, which only induces a minor decrease of slack.
VI. NETS OF DIFFERENT TOPOLOGY

One obvious difference caused by out-core topology difference is that the buffering solutions across instances are no longer required to be the same (it is also impossible in many cases). Thus, the instances have different buffer solutions outside the cores. The basic idea to handle topology difference in a part of the net is to propagate net solutions inside the cores and instance solutions outside the cores. Those out-core nodes that have their parent nodes (immediate upstream nodes) inside the cores are called boundary nodes. On the boundary of the cores, we use various convergence or divergence techniques to allow transform between net and instance solutions. Sections VI-A–VI-C handle three common cases, namely, upper cores, lower cores, and sandwich cores, respectively.

A. Nets With Upper Cores

For nets with upper cores, we propagate instance solutions in out-core nodes until the core boundary is reached, perform convergence of instance solutions to form net solutions at boundary nodes, and propagate net solutions among in-core nodes from the boundary to the source. The propagation of instance solutions in each instance is the same as in Lillis’ algorithm, and the propagation of net solutions is the same as presented in the previous section. However, the convergence of instance solutions can make the net solution set size at the core boundary grow exponentially. For each node on the core boundary, every instance solution has equal opportunity to be considered in net solutions. Thus, every combination composed of $h$ instance solutions respectively from the $h$ instances is considered in solution convergence at node $i$. Our approach to manage the converged net solution set size is to prune them by critical components as in nets with unique topology. Critical components of net solutions are extracted as given by (4), (5), and (12). Therefore, the net solution set size at boundary nodes is approximately the size of traditional solution set at a node with $h$ children.

Refer to Algorithm 4 for a procedure outline of processing one node in this case. Notice that the procedure $node\_solution(v_i, \phi_{h})$ is basically Lillis’ algorithm applied on node $v_i$ in instance $\phi_{h}$.

B. Nets With Lower Cores

Solution propagation in nets with lower cores proceeds in a different order. First, net solutions in the cores are propagated from the sinks to the core boundary. Then, net solutions at the boundary are split into instance solutions, which are propagated to the source respectively in their individual instances. The issue caused by the splitting is that overall (final) instance solutions in different instances may contain different partial net solutions at boundary nodes. As a result, when all instances pick their solutions at the source, the overall solutions from different instances may imply different buffering solutions in the cores, which is obviously contradictory to the problem constraint. We use an approximation to resolve this problem—simply allowing each boundary solution set to have only one net solution, which is an optimal solution according to the problem objective. This way, the net solution is guaranteed to be consistent across all the instances.

In order to pick the solution at a boundary node for min-cost subject to timing constraint problem, we have to predict the source RAT of each solution on the boundary. Given a net solution $s = \{\hat{c}, \hat{q}, \hat{w}\}$ at boundary node $v_i$, its source RAT is estimated by

$$h \min_{k=1}^{h} (\hat{q} - D(v_0, v_i, \phi_k))$$

(14)

where $D(v_0, v_i, \phi_k)$ is the delay between nodes $v_0$ and $v_i$ in instance $k$. We borrow an estimation of $D(v_0, v_i, \phi_k)$ from the study in [9]

$$D(v_0, v_i, \phi_k) = L(R_b C + RC_b + \sqrt{2R_b C_b RC})$$

(15)

where $L$ is the length of the path between $v_0$ and $v_i$, $R$ and $C$ are the unit resistance and capacitance of wires, and $R_b$ and $C_b$ are the resistance and capacitance of the buffer.

An outline of the procedure processing one node in nets with lower cores is given in Algorithm 5.

Algorithm 4 Procedure: $node\_solution\_netuppercore(v_i)$

if $v_i$ is an out-core node then
for all $k \in \{1, \ldots, h\}$ do
node_solution($v_i, \phi_k$)
end for

if parent($v_i$) is an in-core node then
for all $s \in S(v_i)$ as a combination of instance solutions $(c(v_i, \phi_1), q(v_i, \phi_1), \ldots, c(v_i, \phi_h), q(v_i, \phi_h))$ do
extract($\hat{c}(v_i), \hat{q}(v_i)$) of $s$ by eqs. (5) and (12)
end for
prune $S(v_i)$ by critical components of solutions
end if
else
node_solution($v_i$)
end if

C. Nets With Sandwich Cores

Since nets with sandwich cores can be considered as nets containing both lower cores and upper cores, a straightforward way of buffering is to propagate solutions as in a net with lower cores from the bottom up, and then, solutions at the boundary of the upper core are converged and propagated as in a net with upper cores. A simple sketch of the procedure is presented in Algorithm 6.

A special case of sandwich cores is the net with identical source and lower core. In this case, the upper core consists of only the identical source. The same algorithm used to nets with sandwich cores is applied to nets with identical source and upper cores.
Algorithm 5 Procedure: node_solution_netlowercore(v_i)
  if v_i is an in-core node then
    node_solution(v_i)
  end if
  if parent(v_i) is an out-core node then
    reduce S(v_i) to solution set with only one solution \{s_{opt}\}, s.t. s_{opt} is an optimal solution for the problem objective
    expand the net solution to a combination of h instance solutions (c, q_1, ..., c, q_h)
  end if
  else
    for all k ∈ {1, ..., h} do
      node_solution(v_i, \phi_k)
    end for
  end if

Algorithm 6 Procedure: node_solution_netuplowcore(v_i)
  if v_i is a lower-core node then
    node_solution_netlowcore(v_i)
  else
    node_solution_netupcore(v_i)
  end if

VII. EXPERIMENTAL RESULTS

A. Experiment Setup

The multicore buffering algorithms are implemented with C++ code and tested on 200 nets based on industrial designs. The number of sinks varies from 2 to 36. The number of candidate buffer locations for each instance of each net is up to 300. Table I shows the distribution of number of sinks.

We consider four-core designs so that there are four instances for each net. Sections VII-B and VII-C present the experiment of nets with identical topology across all instances, which involve different sink and source characteristics. For each sink, its capacitances at different instances vary by at least ±10% around a center value. The sink RATs may vary by up to ±10% of max source–sink delay. The variations of driver resistance are about ±5% of a center value. The buffer library contains five buffers. The driving resistance of buffers varies between 45 and 120 Ω, and the buffer input capacitance is from 6.27 to 12.15 fF.

Sections VII-D–VII-F show experiments in four cases of topology difference: nets with upper cores, nets with lower cores, nets with sandwich cores, and nets with identical source and lower cores. The variations in out-core source and sinks are the same as the experiment of identical topology. The topology difference across instances of each net is reflected in the difference in wire capacitance and resistance, which is induced by fluctuating wire length between out-core nodes. The length of each wire among different instances is up to ±15%.

All interconnect delays at buffers and wires are calculated according to Elmore delay model. Our algorithms can be extended for accurate delay models like in [4]. The total cost of a net is measured based on the total input capacitance of all inserted buffers.

To the best of our knowledge, there is no previous work on the multicore buffering problem. We compared our algorithm with instance-based buffering (IBB). As mentioned previously, IBB basically solves a traditional buffering problem on each individual instance. It selects the instance whose best solution (for whichever objective) is critical among all the instances, and applies this best solution to all the other instances.

B. Max-Slack Solution for Identical Topology

In the max-slack problem formulation, the objective is to maximize the critical slack, which is the minimum slack among all instances of a net. For each net, we compute the critical slack improvement from our algorithm over IBB, which is the critical slack from our algorithm minus the critical slack from IBB. Fig. 6 shows the histogram of the critical slack improvement of all nets. For all the nets, the slack improvement is at least 20 ps. That is, our algorithm produces larger critical
net slack than IBB for all nets. The average slack improvement is around 100 ps.

In order to obtain more insight, we compare the slack of individual instances. Note that the algorithm performance is measured in critical slack, by which our algorithm is always better than IBB. Here, by instance slack, we try to look further into the background mechanics. Fig. 7 shows the histogram of the instance slack improvement from our algorithm over IBB. Most of the time, our algorithm results in remarkable improvement over IBB. Occasionally, the improvement is a small negative value. This is because our algorithm attempts to provide good solutions for all instances of a net while IBB is usually focused only on one instance for a net. Although IBB may occasionally yield a good solution to a specific instance, the solution is often poor for the other instances. For those instances with poor IBB solutions, the improvement from our algorithm can be very large. This is why the histogram of instance slack improvement in Fig. 7 is spread out more than the critical slack improvement in Fig. 6.

Table II summarizes the average slack improvement, total cost, and computation time of the two algorithms. Our algorithm has slightly higher total cost (0.6% larger) than IBB, but it achieves significantly better slack at a much faster speed.

C. Min-Cost Solution for Identical Topology

Min-cost solutions are from the formulation that minimizes the total buffer cost subject to timing constraints. In other words, a min-cost solution should have nonnegative slack. Otherwise, it has timing violation.

Fig. 8 shows the distribution of critical slack, which is the minimum slack among all instances for a net, from both our algorithm and IBB. It is clear that our algorithm can ensure nonnegative slacks, while IBB causes many timing violations. IBB picks the solution from an instance and applies it to the other instances. Although such a solution is feasible for one instance, there is no guarantee that it is feasible for the other instances. The distributions of slacks for individual instances are shown in Fig. 9. Again, all instance slacks from our algorithm are nonnegative, while IBB causes many timing violations.

The number of timing violations, total buffer area, and CPU time for min-cost solutions are listed in Table III. The total buffer area from our algorithm is slightly larger than that from IBB, but the difference is negligible.

D. Solution for Nets With Upper Cores

The nets with upper cores in this experiment are the same nets from identical topology experiments, except that the wire lengths at the bottom two layers of the nets are perturbed among net instances. Specifically, the two lowest level nodes are found after performing breadth-first-search top–down through a net, and then, the lengths of wires that are connected to nodes in the lowest two levels are perturbed.

Max-slack and min-cost solutions of each net are obtained much faster using our algorithm compared with IBB. The distribution of critical slack improvement over IBB is shown in Fig. 10. The improvement is positive in every net, with negligible difference in total buffer area. The critical component calculation at the boundary of the cores works effectively, particularly for the case where our algorithm improves the slack by 160–180 ps. Sometimes, the worst-scenario critical component approximation is very conservative; thus, the slack improvement by our algorithm is not very significant for those nets with 20–40-ps slack improvement. The average critical slack improvement per net by max-slack solutions is 64.3 ps as shown in Table IV. The results for min-cost solutions are shown in Table V. The total buffer area for min-cost solutions from the two algorithms is almost identical, while our algorithm

<table>
<thead>
<tr>
<th>TABLE II</th>
<th>MAX-SLACK SOLUTION RESULTS FOR 200 NETS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Instance Based Buffering</td>
</tr>
<tr>
<td>Avg slack improvement/net (ps)</td>
<td>0 (baseline)</td>
</tr>
<tr>
<td>Avg slack improvement/instance (ps)</td>
<td>0 (baseline)</td>
</tr>
<tr>
<td>Total buffer area (fF)</td>
<td>2503.66</td>
</tr>
<tr>
<td>Total CPU time (s)</td>
<td>6408</td>
</tr>
</tbody>
</table>

Fig. 8. Histograms of critical slacks for min-cost problem.

Fig. 9. Histograms of instance slacks for min-cost problem.
TABLE III
MIN-COST SOLUTION RESULTS FOR 200 NETS

<table>
<thead>
<tr>
<th></th>
<th>Instance Based Buffering</th>
<th>Our Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total # of critical timing violations (ps)</td>
<td>155 (77.50% of 200 nets)</td>
<td>0</td>
</tr>
<tr>
<td>Total # of instance timing violations (ps)</td>
<td>282 (35.26% of 800 instances)</td>
<td>0</td>
</tr>
<tr>
<td>Total buffer area ((fP))</td>
<td>2312.12</td>
<td>2314.77</td>
</tr>
<tr>
<td>Total CPU time (s)</td>
<td>6412</td>
<td>851</td>
</tr>
</tbody>
</table>

For the test in a special case of sandwich cores—identical source and lower cores, we use nets with wire length perturbed in layers 2 and 3. The results are summarized in Table X. The slack improvement is a little lower than the general sandwich core case, due to the small experiment size. The slack improvement distribution over individual nets is shown in Fig. 13.

VIII. CONCLUSION

This paper proposes algorithms for multicore buffer insertion. Three techniques are developed. One of them aims to detect a condition for solution space reduction. The other two target at extracting critical components from candidate solutions. The first retains the algorithm optimality, while the second has minor approximation. These techniques combined with traditional buffering method are applied to nets with topology difference. The experimental results show that our algorithm significantly outperforms an extension to conventional buffering in terms of both slack quality and computation speed. Future work seeks to incorporate various traditional buffering speedup techniques, such as those in [5], into our algorithm framework.

APPENDIX

Property 3:

Proof: According to Property 2, in order to prove that \(s(v_i)\) is inferior to \(s'(v_i)\), we just need to show that with any complementary solution \(u(v_j)\), overall solution \(s(v_i) \cup u(v_i)\) is inferior to \(s'(v_i) \cup u(v_i)\).

We prove this by induction on the steps of solution propagation from current node to the source.

Basic step: At the current node \(v_i\), the following inferiority relation holds on \(s(v_i)\) and \(s'(v_i)\):

\[
\hat{w}(v_i) \geq \hat{w}'(v_i), \quad \hat{c}(v_i) \geq \hat{c}'(v_i), \quad \text{and} \quad \hat{q}(v_i) \leq \hat{q}'(v_i).
\]

Induction step: For any two consecutive nodes \(v_j\) and its parent \(v_k\) on the path from \(v_i\) to the source \(v_0\), if the inferior relation holds on two solutions at \(v_j\), i.e.,

\[
\hat{w}(v_j) \geq \hat{w}'(v_j), \quad \hat{c}(v_j) \geq \hat{c}'(v_j), \quad \text{and} \quad \hat{q}(v_j) \leq \hat{q}'(v_j)
\]

then the inferior solution relation is also held on the solutions, \(s(v_k)\) and \(s'(v_k)\) at \(v_k\), which are derived from \(s(v_j)\) and \(s'(v_j)\), respectively, i.e.,

\[
\hat{w}(v_k) \geq \hat{w}'(v_k), \quad \hat{c}(v_k) \geq \hat{c}'(v_k), \quad \text{and} \quad \hat{q}(v_k) \leq \hat{q}'(v_k).
\]

Apparently, if the aforementioned induction stands, we have \(\hat{w}(v_0) \geq \hat{w}'(v_0), \quad \hat{c}(v_0) \geq \hat{c}'(v_0), \quad \text{and} \quad \hat{q}(v_0) \leq \hat{q}'(v_0)\) at the source \(v_0\), i.e., \(s(v_i) \cup u(v_i)\) is inferior to \(s'(v_i) \cup u(v_i)\).
TABLE IV
MAX-SLACK SOLUTION RESULTS FOR TEN NETS WITH UPPER CORES

<table>
<thead>
<tr>
<th></th>
<th>Instance Based Buffering</th>
<th>Our Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg slack improvement/net (ps)</td>
<td>0 (baseline)</td>
<td>64.3</td>
</tr>
<tr>
<td>Total buffer area (IF)</td>
<td>529.6</td>
<td>530.8</td>
</tr>
<tr>
<td>Total CPU time (s)</td>
<td>1758</td>
<td>720</td>
</tr>
</tbody>
</table>

TABLE V
MIN-COST SOLUTION RESULTS FOR TEN NETS WITH UPPER CORES

<table>
<thead>
<tr>
<th></th>
<th>Instance Based Buffering</th>
<th>Our Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total # of critical timing violations (ps)</td>
<td>10 (100% of 10 nets)</td>
<td>0</td>
</tr>
<tr>
<td>Total # of instance timing violations (ps)</td>
<td>22 (55% of 40 instances)</td>
<td>0</td>
</tr>
<tr>
<td>Total buffer area (IF)</td>
<td>529.6</td>
<td>530.8</td>
</tr>
<tr>
<td>Total CPU time (s)</td>
<td>1758</td>
<td>720</td>
</tr>
</tbody>
</table>

TABLE VI
MAX-SLACK SOLUTION RESULTS FOR TEN NETS WITH LOWER CORES

<table>
<thead>
<tr>
<th></th>
<th>Instance Based Buffering</th>
<th>Our Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg slack improvement/net (ps)</td>
<td>0 (baseline)</td>
<td>155.2</td>
</tr>
<tr>
<td>Total buffer area (IF)</td>
<td>567.2</td>
<td>562.4</td>
</tr>
<tr>
<td>Total CPU time (s)</td>
<td>841</td>
<td>62</td>
</tr>
</tbody>
</table>

TABLE VII
MIN-COST SOLUTION RESULTS FOR TEN NETS WITH LOWER CORES

<table>
<thead>
<tr>
<th></th>
<th>Instance Based Buffering</th>
<th>Our Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total # of critical timing violations (ps)</td>
<td>10 (100% of 10 nets)</td>
<td>0</td>
</tr>
<tr>
<td>Total # of instance timing violations (ps)</td>
<td>19 (48% of 40 instances)</td>
<td>0</td>
</tr>
<tr>
<td>Total buffer area (IF)</td>
<td>532.0</td>
<td>530.4</td>
</tr>
<tr>
<td>Total CPU time (s)</td>
<td>841</td>
<td>62</td>
</tr>
</tbody>
</table>

Fig. 11. Histograms of critical slack improvement for nets with lower cores.

The basic step holds immediately from the problem. Now, we prove the induction step. Each solution propagation step from node $v_j$ to its parent $v_k$ is composed of two operations: merging of solutions from the children of $v_k$, and adding buffer and wire delay on the fanin of $v_k$.

First, we prove that when $s(v_j)$ and $s'(v_j)$ are merged with a solution $s(v_l)$ from its sibling $v_l$, the new merged solutions $s(v_k)$ and $s'(v_k)$ formed at their parent $v_k$ still have the relation, $\hat{w}(v_k) \geq \hat{w}'(v_k)$, $\hat{c}(v_k) \geq \hat{c}'(v_k)$, and $\hat{q}(v_k) \leq \hat{q}'(v_k)$. By the definition of iso-cap nets, $\hat{w}(v_k) \geq \hat{w}'(v_k)$ and $\hat{c}(v_k) \geq \hat{c}'(v_k)$ are trivially true. Here, $\hat{q}(v_k) \leq \hat{q}'(v_k)$ is proved by contradictory.

Assume $\hat{q}(v_k) > \hat{q}'(v_k)$ instead. Since merging operation changes each $q$ value in a solution in a nonincreasing way, in order for $\hat{q}(v_k)$ to become larger than $\hat{q}'(v_k)$, $\hat{q}'(v_k) < \hat{q}'(v_j)$ must hold.

Without loss of generality, let $\hat{q}'(v_k) = \hat{q}'(v_k, \phi_1)$. (16)

Fig. 12. Histograms of critical slack improvement for nets with sandwich cores.
The fact $\hat{q}'(v_j) \leq q'(v_j, \phi_1)$ and the aforementioned relation $\hat{q}'(v_k) < \hat{q}'(v_j)$ imply $q'(v_k, \phi_1) < q'(v_j, \phi_1)$, which leads to $q(v_k, \phi_1) < q'(v_k, \phi_1)$. Then, we have

$$q(v_k, \phi_1) \leq q(v_l, \phi_1) < q'(v_k, \phi_1).$$  \hspace{1cm} (17)

Since $\hat{q}(v_k) = \min_r q(v_k, \phi_r)$, we have

$$\hat{q}(v_k) \leq q(v_k, \phi_1).$$  \hspace{1cm} (18)

From (16), (17), and (18), we get $\hat{q}(v_k) \leq \hat{q}'(v_k)$, which contradicts with the assumption $\hat{q}'(v_k) > \hat{q}(v_k)$.

Therefore, we have proved that $\hat{q}(v_k) \leq \hat{q}'(v_k)$ holds after merging operation.

Next, we show that the relation also holds after adding buffer and wire delay at the fanin of $v_k$. Since every solution has identical capacitance for all instances, all instances are shifted by the same amount of delay in a solution. Because $\hat{c}(v_k) \geq \hat{c}'(v_k)$, the buffer and wire delay $d$ for $s(v_k)$ is larger than $d'$ for $s'(v_k)$. As a result, $\hat{q}(v_k) - d \leq \hat{q}'(v_k) - d'$ still holds, since $\hat{q}(v_k) \leq \hat{q}'(v_k)$. Thus, after adding buffer and wire delay, $\hat{w}(v_k) \geq \hat{w}'(v_k)$, $\hat{c}(v_k) \geq \hat{c}'(v_k)$, and $\hat{q}(v_k) \leq \hat{q}'(v_k)$ still hold.

Therefore, the induction step is proved by showing that merging solutions and adding buffer and wire delay preserve the inferiority relation in each induction step.

**ACKNOWLEDGMENT**

The authors would like to thank J. Zuo, Y.-Y. Mo, and S. Chowdhury of SUN Microsystems for their insights to the problem; C. Sze and Z. Li for their help on experimental test cases; and D. Beece for his help on the writing. The authors would also like to thank the reviewers for their valuable comments. Their suggestions make this paper a lot better.

**REFERENCES**


Yifang Liu received the B.S. degree in computer science from the University of Electronic Science and Technology of China, Chengdu, China, in 1999 and the M.S. degree in computer science from the University of Maryland, Baltimore County, in 2005. He is currently working toward the Ph.D. degree in the Department of Electrical and Computer Engineering, Texas A&M University, College Station. His current research interests include computer-aided design for VLSI circuits, particularly physical design, interconnect analysis, and optimization.

Jiang Hu (M’01–SM’07) received the B.S. degree in optical engineering from Zhejiang University, Hangzhou, China, in 1990, and the M.S. degree in physics and the Ph.D. degree in electrical engineering from the University of Minnesota, Minneapolis, in 1997 and 2001, respectively. He was with the IBM Microelectronics from January 2001 to June 2002. Currently, he is an Associate Professor with the Department of Electrical and Computer Engineering, Texas A&M University, College Station. His current research interests include computer-aided design for VLSI circuits, particularly on interconnect optimization, clock network synthesis, variation tolerance technology, and design for manufacturability.

Dr. Hu has served as technical program committee member for DAC, ICCAD, ISPD, ISQED, ICCD, DATE, and ISCAS. Currently, he is an Associate Editor of IEEE Transactions on CAD. He received a best paper award at the ACM/IEEE Design Automation Conference in 2001 and an IBM Invention Achievement Award in 2003.

Weiping Shi (S’91–M’92–SM’02) received the B.S. and M.S. degrees in computer science from Xi’an Jiaotong University, Xi’an, China, in 1982 and 1984, respectively, and the Ph.D. degree in computer science from the University of Illinois, Urbana–Champaign, in 1992. He is currently a Professor with the Department of Electrical and Computer Engineering, Texas A&M University, College Station. His current research interests include layout synthesis, parasitic extraction, testing, and design and analysis of algorithms.