Taming the Complexity of Coordinated Place and Route

Jin Hu, Myung-Chul Kim and Igor L. Markov
{jinhu, mckima}@us.ibm.com, imarkov@eecs.umich.edu

ABSTRACT

IC performance, power dissipation, size, and signal integrity are now dominated by interconnects. However, with ever-shrinking standard cells, blind minimization of interconnect during placement causes routing failures. Hence, we develop Coordinated Place-and-Route (CoPR) with (i) a Lightweight Incremental Routing Estimation (LIRE) frequently invoked during placement, (ii) placement techniques that address three types of routing congestion, and (iii) an interface to congestion estimation that supports new types of incrementality. LIRE comprehends routing obstacles and non-uniform routing capacities, and relies on a cache-friendly, fully-incremental routing algorithm. Our implementation extends and improves our winning entry at the ICCAD 2012 Contest.

1. INTRODUCTION

The nature of global routing has changed since the 1980s as interconnect stacks grew from three metal layers to 9-12 layers with non-uniform pitches [19, 20]. Router runtimes have increased, and so has the impact of routing on design quality. Modern *global routing* cannot be viewed as a standalone optimization because of signal integrity concerns and the impact of coupling capacitance on interconnect delays. *Placement* is also no longer standalone, as it interacts with numerous other optimization steps to control interconnect lengths and delays. In the last 15 years, global placement has often been guided by routability estimation [16] in commercial EDA tools and academic contests [18–20]. The development of such *integrative optimizations* requires understanding the strengths and weaknesses of *dedicated optimizations*, as well as invoking the right primitive at the right time. Indeed, *complexity* — both the number of steps executed at runtime and the number of lines of code — is the main gating factor for what can be achieved by EDA tools in the foreseeable future. Moreover, new optimization primitives must be justified by their context and intended use.

In this work, we develop a streamlined system for Coordinated Place-and-Route (CoPR) that (i) uses cache-friendly routing primitives to quickly and accurately estimate routing congestion (LIRE), (ii) leverages incrementality in routing and congestion updates in new ways, and (iii) offers a new categorization of congestion and new congestion-relief techniques during placement. CoPR achieves unprecedented trade-offs between speed and placement quality on large industry netlists, as we illustrate using ICCAD 2012 contest benchmarks from IBM Research [20].

The remainder of this paper is structured as follows. Section 2 presents our fast and accurate routing estimation technique. Section 3 introduces our placement techniques that proactively alleviate routing congestion. Section 4 describes the interactions between the placer and the routing estimator. Section 5 compares our techniques to currently-known approaches. Section 6 empirically validates the scalability of our techniques. Section 7 concludes our discussion. Supplemental material is provided in the Appendices.

DAC'13, May 29 - June 07 2013, Austin, TX, USA.

2. LIRE: ROUTING ESTIMATION

We develop a Lightweight Incremental Routing Estimator (LIRE) that quickly produces congestion maps as accurate as those by a global router (Figure 5). Empirically, we target 75K nets per second,[1] but also facilitate a tradeoff between quality and runtime. In contrast, modern routers [4, 12] complete 6K nets per second.[1]

Notation. We consider an $X \times Y$ routing grid $G(V, E)$ with (i) a set V of *GCells* (nodes) where each GCell $v \in V$ has integer coordinates (x_v, y_v), and (ii) and a set E of directed edges $e = (v_1, v_2)$, where the weight w_e of edge e encapsulates routing congestion and history costs (Lagrangian multipliers). Each node $v \in V$ is adjacent to its four cardinal neighbors: `NORTH` $(x_v, y_v + 1)$, `SOUTH` $(x_v, y_v - 1)$, `EAST` $(x_v + 1, y_v)$ and `WEST` $(x_v - 1, y_v)$. Consider a point-to-point connection π between two distinct GCells $S, T \in V$. When $x_S \leq x_T$ and $y_S \leq y_T$, a *forward* edge for π is an edge (v_1, v_2) such that $x_{v_1} < x_{v_2}$ or $y_{v_1} < y_{v_2}$, i.e., `EAST` or `NORTH`, and a *backward* edge for π is an edge (v_1, v_2) such that $x_{v_1} > x_{v_2}$ or $y_{v_1} > y_{v_2}$, i.e., `WEST` or `SOUTH`. Definitions for the three other orientations of π are symmetrical.

Key definitions. A *route segment* is a directed path in the routing grid. A *flat* route segment is a set of directed edges that are all `NORTH`, `SOUTH`, `EAST` or `WEST`. A *monotonic* segment is a connected set of flat segments such that each flat segment is either: (i) `NORTH` or `EAST`, (ii) `NORTH` or `WEST`, (iii) `SOUTH` or `EAST`, or (iv) `SOUTH` or `WEST`. Each monotonic segment is classified as `NORTH-EAST`, `NORTH-WEST`, `SOUTH-EAST`, or `SOUTH-WEST`. A *route* r_π is a collection of routing segments linking S and T.

2.1 Faster Routing

Global routing spends a large fraction of runtime finding weighted shortest paths in highly-congested regions [20]. Such shortest-path (maze) routing is necessary in congested regions for both *global routing* and *accurate congestion estimation* because, unlike pattern routing, it adequately captures detours. Detours are shaped by edge weights, which include congestion and history costs [4]. These weights must be maintained with sufficient accuracy and can be neither binned nor rounded without adverse impact on resulting routes. Therefore, shortest-path routing in congested regions is performed by A*-search. However, (i) the priority queue in A*-search is responsible for an extra $O(\log V)$ term in the overall complexity of the algorithm, (ii) priority queues, even when implemented using Fibonacci heaps, are too slow [10] and their pointer-based algorithms can experience costly cache misses, (iii) typical A* admissible functions based on straight-line distance become ineffective when history-based costs become large, and (iv) A*-search cannot leverage incrementality, i.e., given a candidate path, it cannot check optimality or perform an incremental improvement.

[1] Median single-thread router performance on placements by the top three ICCAD 2012 contestants (Intel Xeon 3.4GHz CPU).

Algorithm 1 Bellman-Ford Algorithm with Non-negative Weights

Input: Point-to-point connection π, Search Space (V', E')
Output: route r_π

1: **for** i from $1 \rightarrow |V'|$ **do**
2: $cost[v_i] = \infty$;
3: **end for**
4: $cost[S] = 0$;
5: **for** i from $1 \rightarrow |V'|$ **do**
6: **for** $j = 1$ from $1 \rightarrow |E'|$ **do**
7: $e_j = (v_1, v_2)$;
8: **if** $cost[v_2] < cost[v_1] +$ COST(e_j) **then** ▷ relaxation
9: $cost[v_2] = cost[v_1] +$ COST(e_j);
10: $parent[v_2] = v_1$;
11: **end if**
12: **end for**
13: **end for**
14: r_π = TRACE_PATH(π);

Linear-time cache-friendly routing. Given that A*-search is derived from Dijkstra's algorithm [1, Section 24.3], we hope to avoid these priority-queue-based approaches. Of the classic weighted shortest-path algorithms, the *Bellman-Ford (BF) algorithm* [1, Section 24.1] is array-based and moreover preserves memory locality. However, it may require V linear-time passes, taking $O(EV)$ time.

Notably, the *worst-case complexity* of Bellman-Ford (BF) can be avoided in global routing. Recall that each BF pass performs $E \times O(1)$ *relaxation* steps. When no relaxations in a pass result in improvement, no further improvement is achieved in later passes. Thus, BF can be terminated early without the loss of optimality.

During global routing, we consider one point-to-point connection $(S \rightarrow T)$ at a time. Routing is limited to a subgrid $G'(V', E') \subseteq G$ enclosed in an isothetic (coaxial) bounding box that contains S and T. To generate a route, we visit the nodes of G' in a specified ordering $v_0, v_1, \ldots, v_{|V'|-1}$.[2] While the Bellman-Ford algorithm supports any node visitation ordering, we specify an ordering that not only affords us the highest memory locality, but also caters to the common case of monotonic paths. Starting from S, the nodes are traversed in the row containing S, and at each node v, relaxation is performed (lines 8-9 in Algorithm 1) along the four v-incident edges pointing toward T. The nodes in the next row closer to T are traversed, and so on until the row that contains T. When the node traversal follows the in-memory array layout (by rows or by columns), this method maintains the locality of memory access.

We propose to optimize BF passes with *duplex-edge relaxation*. At each edge considered by this technique, relaxations are attempted in both directions, but only *forward-looking edges* are considered at each vertex. While the same number of edges is considered per pass, cache utilization and memory locality are improved because for each adjacent vertex (and edge cost) loaded from memory, two relaxations can be attempted rather than just one. Furthermore, if the first relaxation succeeds, the second one cannot occur — this saves an extra comparison. For example, at node v with coordinate (x, y), we relax either the outgoing NORTH edge $(x, y) \rightarrow (x, y+1)$ *or* the incoming SOUTH edge $(x, y+1) \rightarrow (x, y)$. A similar duplex relaxation is performed in the EAST and WEST directions. By explicitly modeling via costs within these traversals [4, Section 3.4], BF will prefer fewer-bend routes.

Monotonic routing with one linear-time BF pass. As a special case, an optimal monotonic route can be found by (i) considering only forward edges (e.g., NORTH and EAST), and (ii) fixing the considered space to the bounding box b with dimensions $w \times h$, $w = x_T - x_S + 1$ and $h = y_T - y_S + 1$, that minimally contains S and T. Let t be the $w \times h$ matrix where $t[x][y]$ stores the partial cost from S with coordinates $(0, 0)$ to a node $v = (x, y)$. By construction, the cost at (x, y) depends solely on the costs at $(x-1, y)$ and $(x, y-1)$. Therefore, by visiting the nodes in row order (or column order) from S toward T, we visit every node in b exactly once. Since b has $w \times h$ nodes, the runtime complexity is $O(wh)$.

[2]Edges are traversed in the increasing order of adjacent vertices.

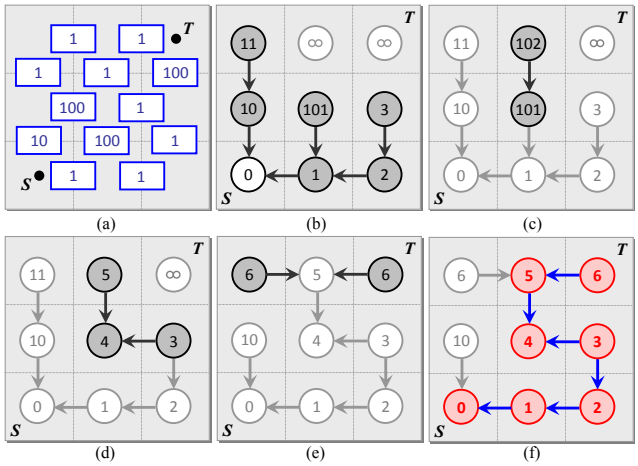

Figure 1: Applying one BF pass with duplex-edge relaxation and echo-relaxation to a point-to-point connection $S \rightarrow T$ without via-cost modeling. Arrows point to the previous node in the path. (a) The routing grid and edge costs (congestion). Let S have coordinate $(0, 0)$. (b) The partial costs of the first row and the center-left node have been populated. (c) Relaxing the NORTH $(1, 1) \rightarrow (1, 2)$ and SOUTH $(1, 2) \rightarrow (1, 1)$ edges at node with coordinate $(1, 1)$. (d) Relaxing the EAST $(1, 1) \rightarrow (2, 1)$ and WEST $(2, 1) \rightarrow (1, 1)$ edges at node with coordinate $(1, 1)$. The cost at $(1, 1)$ has been updated by the WEST edge and is propagated to $(1, 2)$. (e) The remaining nodes are considered, and partial costs are populated through T. (f) An optimal path with three monotonic segments is found in a single BF pass.

Non-monotonic routing with one linear-time BF pass. Recall that BF supports any node (and edge) ordering. Some optimal non-monotonic routes can be found in linear time within the bounding box b that minimally contains π by (i) employing duplex-edge relaxation and (ii) *echo-relaxation* if the relaxation succeeded in the direction opposite to node ordering (from a greater-numbered node to a smaller-numbered node). That is, in the forward-going node ordering, if a backward edge at node $v(x, y)$ results in a smaller-cost route, we forward-propagate the smaller cost through all recently-relaxed edges incident to v. Figure 1 illustrates finding an optimal route with three distinct monotonic segments in one BF pass. This improvement is effective in detouring short nets, and a majority of nets are short in practice. A more powerful variant of echo-relaxation would propagate costs through *all* incident edges, and allows one BF pass to find longer detours (not used in this work).

Non-monotonic routing with BF and Yen's improvement. J. Y. Yen [23] suggested that *reversing* the node ordering between BF passes reduces the number of passes required to find an optimal path. We refer to the Bellman-Ford algorithm with early termination and Yen's improvement as BFY. Two and three BFY passes can quickly find long detours, as illustrated in Figure 2. This is especially applicable for large nets.

THEOREM 1. *Let π be a point-to-point connection. Finding a minimal-cost route r_π^{min} with m (distinct) monotonic segments requires at most m BFY passes.*

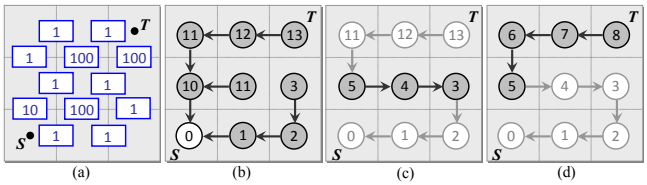

Figure 2: Applying BFY to a point-to-point connection $S \rightarrow T$ without via-cost modeling. (a) The routing grid and edge costs (congestion). (b) The first forward pass finds the optimal monotonic path of cost 13. (c) The backward pass finds a detour. (d) The second forward pass finds the optimal path of cost 8.

Theorem 1 (proved in Appendix A) is significant in practice because many connections are routed with very few monotonic segments. In particular, most connections have very few bends [14], and the number of monotonic segments is upper-bounded by the number of bends. Furthermore, a route with many bends can still be monotonic (Figure 6b). In this context, Theorem 1 suggests that BFY typically finds shortest-path routes in $O(1)$ passes, and explains why BFY finds optimal paths faster than A*-search for most nets in our experiments. In a $w \times h$ bounding box, m BFY passes take $O(mwh)$ runtime. Limiting the number of passes further reduces BFY runtime. This may lead to a small loss of optimality in standalone BFY, but our main focus is on incremental calls to BFY during routing estimation (see the "LIRE in CoPR" columns in Table 1).

Incremental routing with BFY can use any existing route, including those previously found by A*-search. Instead of propagating the costs in an ∞-initialized table, we record the partial costs along an existing route (this is significantly faster than populating the entire table). Subsequent BFY passes find an optimal route, but require less runtime when a near-optimal initial route is available (Figure 6). Multiple such initial routes can be recorded in the BFY table before the first pass.[3] *This type of incrementality speeds up not only rip-up-and-reroute and negotiated-congestion methods, but also repeated invocations of LIRE during placement* (Section 4 also outlines other types of incrementality supported by LIRE).

Coarse-grid routing is based on the observation that large nets often admit near-optimal routes with long flat segments. Therefore, we reduce the search space by only considering every i^{th} row and j^{th} column. This allows us to find a reasonable-cost route quickly, and then incrementally relax it on a finer subgrid (multilevel BFY).

2.2 Fast and Accurate Estimation

Unlike true global routing, constructive congestion estimation needs not optimize routes in congestion-free regions. Finding routes that avoid congested GCells is sufficient. Therefore, existing methods first evaluate several pattern routes (L, C, Z) and invoke more sophisticated algorithms only when needed. For similar reasons, eligible GCells are initially limited to the bounding box of the connection, which is gradually expanded until an acceptable route is found. LIRE too estimates congestion by catering to the common case first. For each point-to-point connection $\pi = S \rightarrow T$, LIRE initially limits the search space to the bounding box b with size $w \times h$ that minimally contains π. It considers the two L routes, with preference for congestion-free routes. If both routes are congested, BFY finds a route with $O(1)$ monotonic segments. If this route is congested, LIRE expands b to to $W \times H$, $w < W \leq X, h < H \leq Y$, which may be based on congestion [13].

[3] A speed-up common in A*-search (for large nets) limits search to GCells in narrow corridors around known routes. This speed-up is equally applicable to BFY, but not used in our implementation.

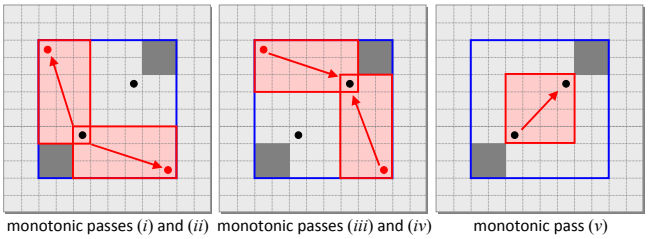

Figure 3: Non-monotonic routing using the Bellman-Ford Algorithm with an expanded bounding box. The red arrows represent monotonic passes.

Within this expanded rectangular bounding box (Figure 3), we only consider a partial rectilinear bounding box for two reasons. First, we reduce the overall runtime by limiting the bounding box. Second, we observe that the space omitted only contributes to routes that require multiple detours. Since congestion estimation needs not generate heavily-detoured routes, we omit this search space to reduce detours and runtime. Within the rectilinear bounding box B, consider the upper-left (UL) and bottom-right (BR) corners. We perform five monotonic-routing passes: (i) $S \rightarrow UL$, (ii) $S \rightarrow BR$, (iii) $UL \rightarrow T$, (iv) $BR \rightarrow T$, and (v) $S \rightarrow T$. For nets with extreme aspect ratios, we found that this is more effective than repeating many monotonic passes. Our implementation expands bounding boxes up to twice the original size.

3. CONGESTION RELIEF

The main precept of routability-driven placement is to increase the *porosity* of placement regions with high routing congestion. Regardless of how congestion is estimated, porosity has traditionally been increased in two ways: (i) *after global placement*, by shifting cell locations [17, 24] and using congestion-driven detailed placement [3, 5, 8], and (ii) *during global placement*, by inflating cells based on early congestion estimates and pin density [3, 5, 8].

While studying the impact of these techniques on challenging IC layouts, we observed their insufficiency. Modifications performed after the global-placement phase must preserve the structure of resulting placement or risk unbearable deterioration of interconnect length. Cell inflation performed during global placement is more flexible and powerful. However, when inflated cells move outside the congested region, new cells must be inflated, and this process may consume all available whitespace without addressing the root cause of congestion in a given region (this phenomenon was confirmed to us by several industrial colleagues and academic colleagues). Further analysis revealed two previously unreported types of routing congestion, which we include below as types 2 and 3.

1) *cell-based* congestion caused by cell-to-cell proximity,
2) *local layout-based* congestion caused by static design properties, such as blockages and reduced routing capacities,
3) *remotely-induced layout-based* congestion attributed to non-local factors, e.g., long nets.

These congestion types are illustrated in Figure 4. The distinctions among them can be blurred by inaccurate congestion maps and also during congestion reduction *after* global placement [17, 24] when cells do not drastically move across the layout. However, these distinctions become sharper when guiding global-placement iterations by accurate congestion maps. Conceptually, type-2 congestion requires whitespace injection into relevant regions *in such a way that whitespace remains in these regions even when cells relocate*.

Cell-based congestion. As the placer spreads cells, it often implicitly keeps cells close together to decrease HPWL. However, this

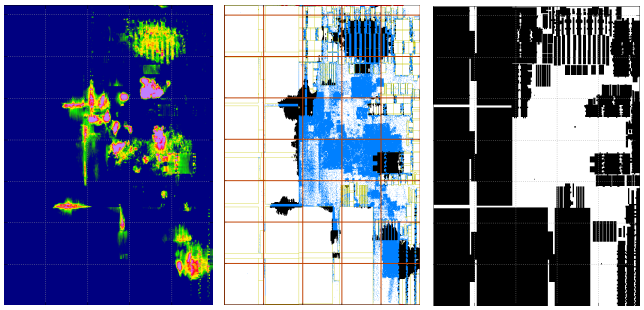

Figure 4: Congestion map produced after one BFG-R [4] iteration (left), placement map of cell locations (center), and blockages (right) for SUPERBLUE2 [19]. In the center, blue indicates movable cells, and black indicates congested GCells over blockages. Congestion is present around blockages (layout-based) and blockage-free regions (cell-based).

"clumping" creates difficult-to-route regions, as there may be too few tracks to accommodate all incident nets. This type of congestion is easily mitigated through cell inflation. However, inflating too many cells *or* inflating some cells by too much can exhaust whitespace too soon, inhibit convergence and undermine quality. To ensure steady improvement, we inflate cells in the top 5% most congested GCells. Details are given in Appendix B.

Layout-based congestion. During HPWL-driven placement, the target density is often high, as this facilitates low-HPWL placement solutions. However, if the placer is not congestion-aware, it may pack cells in regions of high congestion. To this end, we seek to locally increase whitespace to encourage cells to spread elsewhere. However, analytic placement frameworks are not always amenable to techniques that change (local) target density. Instead, we enforce non-uniform target densities in localized regions. We distinguish layout-based congestion as either *local*, which is caused primarily by static constraints such as custom routing-edge capacity reductions, or *remotely-induced*, where congested GCells contain no standard cells but have few routing tracks traversed by long nets. While the former can be addressed through locally injecting whitespace, the latter cannot, as there are no cells to move out of the congested region. In the remainder of this section, we discuss our method of enforcing non-uniform target density by (i) creating a *packing peanut* (fixed cell) at the center of every GCell, and (ii) modifying its size based on congestion.

Implementation. To address *local layout-based congestion*, we modify the size of packing peanuts during the initial HPWL optimization stage based on *pin density*, and during the global placement stage based on routing congestion. During initial placement, we coarsely estimate routing congestion of the design based on available routing capacity and cell pin density. We first divide the layout into 8×8 GCell regions and compute the number of pins in each region. We then (pessimistically) estimate that each pin in the region will occupy two routing tracks, and increase the packing peanuts' size based on the ratio of estimated usage and routing capacity. This approach of coarsely dividing the layout gives the placer a high-level outlook and encourages cells spreading to regions of lower pin density. We define two parameters: (i) the maximum expandable area $PA(g)_{max}$, which is based on the surrounding non-overlapping GCell areas, and (ii) the minimum area $PA(g)_{min}$, which is based on GCell pin density. Let $C(g)^k$ be the congestion of GCell g at routing iteration k. Then the packing peanut area $PA(g)$ at g is initially $PA(g)_{min}$ and

$$PA(g) + 0.15 \cdot \big(PA(g)_{max} - PA(g)\big) \qquad (1)$$

if $C(g)^k > C(g)^{k-1}$ and $C(g)^k > 1$. If the congestion is reduced but not removed, i.e., $C(g)^{k-1} > C(g)^k > 1$, then the packing peanut size remains the same. Otherwise, if congestion is removed, the size is $\max\{PA(g)_{min}, PA(g) \times 40\%\}$.

To address *remotely-induced layout-based congestion*, we increase the packing peanuts in GCells *closest to the congested region* and their neighboring GCells. Such modifications often occur around placement blockages. Across placement iterations, the packing peanuts increase placement porosity by reducing the demand in regions without blockages as well as customizing the resource distribution around blockages. Unlike rectangular macro expansion [5], our approach affords the placer a high degree of flexibility to where long nets can be shifted (Figure 7). It complements cell-inflation techniques by preventing allocated whitespace from moving away.

4. COORDINATED PLACE AND ROUTE

The integration of routing estimation within placement allows us to leverage the existing infrastructure and avoids task redundancy. Giving LIRE up-to-date access to cell locations simplifies the construction of new congestion maps when placement changes.

Incremental placement updates. After its first invocation, LIRE maintains the overall congestion map and keeps track of the GCells traversed by each point-to-point connection π. At subsequent LIRE invocations, if the endpoints of π remain in the same GCells (despite changes in their continuous-valued locations), π's route and its contribution to the congestion map are left unchanged. While this type of incrementality has limited use in early placement iterations, its effect is more pronounced in later iterations and during detailed placement, when the locations have stabilized.

Incremental route updates. When invoked for the first time, LIRE generates routes from scratch. Subsequently, it tries to reuse existing routes where possible. Nets whose terminals relocated to different GCells are rerouted using the original net ordering, as outlined in Section 2. For remaining nets, we check if their routes are congested. Congestion is mitigated by single incremental BFY passes. This helps replicate the accuracy of a maze router and improves runtime by reusing full and partial routes.

Placement-routing interface for coordinated place-and-route:

- `LIRE::Initialize()` reads in the benchmark information, sets up the routing environment, and computes the *static* routing-edge capacities (e.g., due to blockages or custom capacity reductions). Dynamic capacity adjustments such as pin blockages in Section 6, are accounted for by `LIRE::updatePlacement()`.
- `LIRE::updatePlacement()` restructures the nets based on any placement changes, and maintains lists of nets that require full modification, as well as those that can be reused. Dynamic routing capacities are adjusted due to cell-location updates.
- `LIRE::route()` generates routes on an as-needed (lazy) basis. It decomposes each multi-pin net into two-pin subnets based on its MST, and follows the protocol outlined in Section 2.
- `LIRE::genCongMap()` translates edge capacities and usages to a GCell-centric congestion map as in [8], where a GCell is congested if any surrounding edge is congested.

The handling of design hierarchy is entrusted to the placer and does not add complexity to the place-and-route interface (Figure 8). In summary, we advocate a *coordinated* integration style of physical optimizations, where each component uses algorithms that are independently-meaningful and independently-efficient, but also are capable of taking external suggestions. Unlike *simultaneous place-and-route* advocated in [8], this type of integration limits software complexity, allows for component replacement and unit testing. It eases the integration of timing analysis and other components necessary for effective timing closure of modern SoC designs [10].

5. COMPARISONS TO PRIOR ART

Comparing our techniques to prior art, we consider (i) point-to-point routing algorithms, (ii) using global routes versus probabilistic congestion maps, (iii) incremental routing techniques, and (iv) handling congestion around blockages.

Fast routing. The closest publication to our material in Section 2 is [13]. It advocates replacing A*-search with fast linear-time routing algorithms that exploit a different notion of monotonic routes (our work was completed before [13] was published or available to us). It uses multiple passes to find non-monotonic routes and does not claim optimality. It does not consider CPU cache effects and the connection we establish to the Bellman-Ford algorithm with Yen's improvement. Empirically, the RCE estimator [13] is not used to drive a competitive global placer, whereas we report successful results for coordinated place-and-route using LIRE. We believe that congestion-driven bounding-box expansion pioneered in RCE can be valuable, but have not had the time to implement and evaluate it.

The only modern description of an industry router that we could find is in [10]. It concedes that Dijkstra's algorithm [1, Section 24.3] (from which A*-search is derived) is "much too slow" for large modern netlists, even with Fibonacci heaps. However, rather than replace Dijkstra with linear-time algorithms as we do, the authors speed it up with sophisticated data structures (interval-based route-cost representations) and algorithms (sharper admissible functions for A*-search based on landmarks). Direct comparisons would be difficult to make, even if we had access to their source code, because advanced data structures use more memory and require significant up-front set-up, along with maintenance. However, a single-threaded version of LIRE takes only $<$15% of runtime in our entire place-and-route flow, despite frequent ($>$10) invocations by the placer (Table 1). Speeding it up further would have limited impact. Most importantly, we have advanced the goal of our research — to tame the complexity of place-and-route — by entirely avoiding sophisticated routing algorithms and data structures.

Congestion estimation must accurately identify hotspots and guide the placer to relieve congestion. While *probabilistic congestion maps* are easy to implement, they can be slower per net than constructive routing, as shown in [22]. They are also highly inaccurate, as recently articulated by IBM researchers [11]. Nevertheless, most routability-driven placers [3, 5] still use probabilistic methods. As illustrated in Figure 5, congestion maps built using LZ-routing [14] significantly differ from router-based maps. Table 2 compares total overflow (TOF) between L-routing, LZ-routing, LIRE, and maze routing [4]. On average, LIRE overestimates TOF by 4% with no significant outliers.

Incremental routing techniques. All modern routability-driven placers [3, 5, 8] use built-in congestion estimation to construct new estimates from scratch on every invocation. This process is unnecessarily time-consuming, especially when the placement has not changed significantly. While some prior techniques rip-up and reroute some congested nets [25], they assume a static routing (and placement) instance. In contrast, our incremental techniques account for dynamic placement (and routing) instances, take advantage of previous (partial) routes, and update routes on an as-needed basis. These techniques are especially applicable to congestion estimators based on constructive global routing, but also should be helpful in full-fledged routers. Empirically, we matched the accuracy of a full global router with limited runtime overhead.

Placement and routing blockages, e.g., macro blocks, often lead to congestion around their borders. Previous work [5] proactively reserves resources by expanding macros. However, (rectangular) macro inflation is rather crude in controlling whitespace — it either allows all cells or prevents all cells in a given rectangular region. Our non-uniform target density, as implemented with packing peanuts, provides much more flexible control of whitespace, as shown in Figure 7. By increasing the packing peanut sizes in areas of congestion *and* in selected neighboring GCells, we allow cells to move into congestion-free regions around macro borders, whatever shape those regions may assume.

6. EMPIRICAL VALIDATION

Our algorithms are implemented in a tool called CoPR (pronounced "copper") in C++ using the OpenMP library [2] and compiled with g++ 4.7.0. Our global placer is derived from SimPL [9], which was the case for three out of the top four teams at the ICCAD 2012 Contest [20]. Thus, the choice of the global placement algorithm is not a significant factor in relative performance.

Empirical results are reported on the ICCAD 2012 benchmark suite [20] derived by IBM researchers from industry designs. Some of these benchmarks were released only after the results of the ICCAD 2012 Contest were announced. The overall figure of merit combines quality metrics (interconnect length, routing congestion evaluated by a router, and pin blockage) and runtime. Table 1 compares CoPR to official contest results [20] for the top three contestants. With quality metrics based on the NCTUgr router (without runtime), CoPR outperforms NTUplace4h by 7% and SimPLR by 2%, while matching the overall quality of Ripple, which is 5.7$\times$ slower. CoPR runtime intentionally matches SimPLR (the fastest top-3 contestant, which trails CoPR in quality) so that officially-reported runtime ratios between SimPLR and other contestants also apply to CoPR. The last two columns in Table 1 show that LIRE is called by CoPR 14-22 times per run, amounting to $<$15% of CoPR's runtime. With quality metrics based on the BFG-R router, CoPR outperforms NTUplace4h by 3%, SimPLR by 6% and Ripple by 2%, respectively (Table 3). With respect to scoring formulas used at the ICCAD 2012 contest, CoPR outperforms the winner SimPLR (from which CoPR was derived).

7. CONCLUSIONS

Our work deals with an alarming trend in the design or digital random-logic blocks, where interconnect's dominance in area, volume, delay, power and signal-integrity is increasing with every new technology node [7]. If unchecked, this trend is threatening to render Moore's law irrelevant — packing more devices on a chip is useless if they cannot be effectively connected. The most direct and effective remedy known today is to reduce interconnect demand, which can be done by optimizing standard-cell locations and wire routes. As articulated recently by IBM researchers, design flows with separate placement and routing steps have become ineffective for modern ICs [19], but combining the two brings tangible and significant benefits in IC cost [17]. However most of physical-design research continues focusing on standalone optimizations, partly due to the complexities involved in place-and-route integration. These complexities include sophisticated data structures and elaborate multistep optimizations used by state-of-the-art algorithms [10], unmaintainable source-code bases that are unnecessarily entangled, large sets of tuning parameters that may need to be adjusted to individual benchmarks, and of course significant runtime. In this work, we develop an algorithmic framework for coordinated place-and-route (CoPR) that combines independently-meaningful components and systematically reduces the complexities of place-and-route. Our contributions fall into four categories: (i) dramatic acceleration of constructive routing estimation through linear-time cache-friendly algorithms that do not require sophisticated data structures, (ii) significant reductions in the amount of work through pervasive incrementality at the interface between

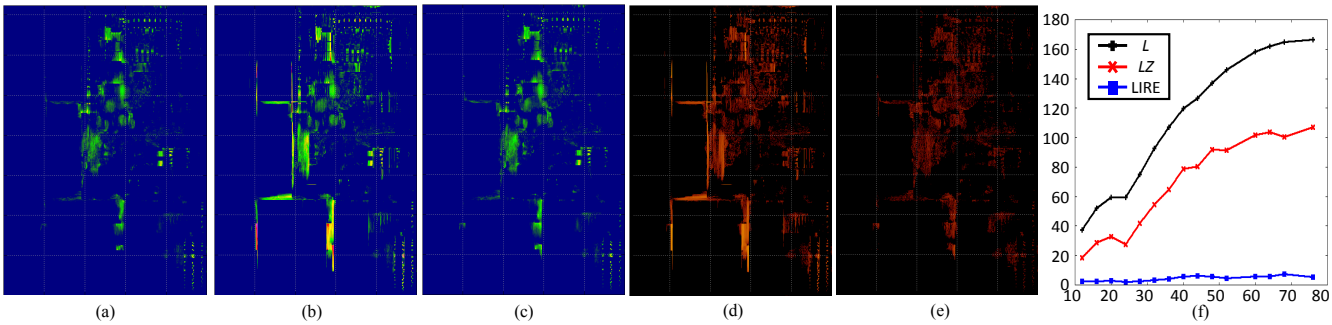

Figure 5: Comparison of different routing estimation techniques on the SUPERBLUE2 benchmark [19]. The congestion map in (a) is produced by BFG-R [4], in (b) — by LZ-routing, and in (c) — by LIRE. Images in (d) and (e) show how well (b) and (c) match (a) — ratios of congestion values are plotted. Orange areas indicate large differences and black areas — no difference. (f) plots the error percentage of total overflow for L-routing, LZ-routing, and LIRE relative to (a) over the placement iterations of CoPR. While all techniques overestimate congestion, LZ-routing and L-routing produce many false positives, whereas LIRE does not.

			Quality metrics using NCTUgr [12] (e8)				Runtime ratio	LIRE in CoPR	
Benchmark	Nodes	Nets	SimPLR (1)	Ripple (2)	NTUplace4 (3)	CoPR	CoPR/SimPLR	calls	LIRE %
SUPERBLUE1	847K	822K	2.789	2.889	2.850	2.860	1.058	14	12.4%
SUPERBLUE3	920K	898K	3.439	3.604	4.477	3.457	0.962	15	14.8%
SUPERBLUE4	600K	567K	2.434	2.269	2.360	2.366	1.445	22	13.9%
SUPERBLUE5	772K	787K	3.603	3.486	4.217	3.510	1.089	14	11.8%
SUPERBLUE7	1.36M	1.34M	4.313	4.291	4.137	4.360	1.099	15	11.9%
SUPERBLUE10	1.20M	1.15M	6.909	6.111	7.190	6.505	1.054	22	16.4%
SUPERBLUE16	699K	697K	2.857	2.840	2.833	2.797	0.687	14	15.3%
SUPERBLUE18	1.27M	469K	1.823	1.791	1.709	1.676	0.816	17	19.0%
Ratios of averages (×)			1.02×	1.00×	1.07×	1.00×	1.01×		14.3%

Table 1: Quality metrics (based on NCTUgr [12]) *without runtime* for the top three contestants as reported at the ICCAD 2012 Routability-driven Placement Contest [20]. CoPR runtimes are compared to those of the fastest top-3 contestant SimPLR by running both tools on the same server. The last two columns show the runtime of LIRE as a percent of total CoPR runtime, and the number of LIRE invocations on each benchmark. Full results for SimPLR, RippleCUHK and NTUplace4h are available at [20]. Using BFG-R [4] (rather than NCTUgr) to evaluate results results in greater advantage for CoPR as shown in Table 3.

placement and routing, (iii) identification of two new types of routing congestion, as well as mechanisms by which a global placer can diagnose them and respond effectively, and (iv) strong empirical results on the most recent benchmarks from IBM Research.

Our place-and-route improvements should (a) lead to more compact (less costly) IC layouts in a way illustrated in [17], and (b) reduce back-end turn-around-time so that IC designers can evaluate a greater number of micro-architectural configurations.

8. REFERENCES

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to Algorithms*, Second Edition, MIT Press and McGraw-Hill, 2001.

[2] L. Dagum and R. Menon, "OpenMP: An Industry Standard API for Shared-memory Programming," *Computational Science and Engineering* 1998, pp. 46-55.

[3] X. He, T. Huang, L. Xiao, H. Tian, G. Cui and E. F. Young, "Ripple: An Effective Routability-driven Placer by Iterative Cell Movement", *ICCAD* 2011, pp. 74-79.

[4] J. Hu, J. A. Roy and I. L. Markov, "Completing High-quality Global Routes", *ISPD* 2010, pp. 35-41.

[5] M.-K. Hsu, S. Chou, T.-H. Lin and Y.-W. Chang, "Routability-driven Analytical Placement for Mixed-size Circuit Designs", *ICCAD* 2011, pp. 80-84.

[6] L. Hsu, R. Iyer, S. Makineni, S. Reinhardt and D. Newell, "Exploring the Cache Design Space for Large Scale CMPs," *Computer Architecture News* 2005, pp. 24-33.

[7] International Technology Roadmap for Semiconductors (ITRS).

[8] M.-C. Kim, J.Hu, D.-J. Lee and I. L. Markov, "A SimPLR Method for Routability-driven Placement", *ICCAD* 2011, pp. 67-73.

[9] M.-C. Kim, D.-J. Lee and I. L. Markov, "SimPL: An Effective Placement Algorithm", *TCAD* 31(1) (2012), pp. 50-60.

[10] B. Korte, D. Rautenbach and J. Vygen, "BonnTools: Mathematical Innovation for Layout and Timing Closure of Systems on a Chip", *Proc. IEEE* 95(3) (2007), pp. 555-572.

[11] Z. Li, C. J. Alpert, G.-J. Nam, C. Sze, N. Viswanathan and N. Y. Zhou, "Guiding a Physical Design Closure System to Produce Easier-to-route Designs with More Predictable Timing", *DAC* 2012, pp. 465-470.

[12] W.-H. Liu, W.-C. Kao, Y.-L. Li, K.-Y. Chao, "Multi-threaded Collision-aware Global Routing with Bounded-length Maze Routing", *DAC* 2010, pp.200-205.

[13] W.-H. Liu, Y.-L. Li, C.-K. Kok,"A Fast Maze-free Routing Congestion Estimator With Hybrid Unilateral Monotonic Routing",*ICCAD* 2012, pp.713-719.

[14] M. Pan, Y. Xu, Y. Zhang and C. Chu, "FastRoute: An Efficient and High-quality Global Router", *VLSI Design* 2012, 18 pages.

[15] S. K. Raman, V. Pentkovski and J. Keshava, "Implementing Streaming SIMD Extensions on the Pentium III Processor", *Micro* 20(4)(2000), pp. 47-57.

[16] P. N. Parakh, R. B. Brown and K. A. Sakallah, "Congestion Driven Quadratic Placement, *DAC* 1998, pp. 275-278.

[17] J. A. Roy, N. Viswanathan, G.-J. Nam, C. J. Alpert and I. L. Markov, "CRISP: Congestion Reduction by Iterated Spreading during Placement", *ICCAD* 2009, pp. 357-362.

[18] N. Viswanathan, C. J. Alpert, C. Sze, Z. Li, G.-J. Nam and J. A. Roy, "The ISPD-2011 Routability-driven Placement Contest and Benchmark Suite", *ISPD* 2011, pp. 141-146.

[19] N. Viswanathan, C. J. Alpert, C. Sze, Z. Li, Y. Wei, "The DAC 2012 Routability-driven Placement Contest and Benchmark Suite", *DAC* 2012, pp. 774-782.

[20] N. Viswanathan, C. J. Alpert, C. Sze, Z. Li and Y. Wei, "ICCAD-2012 CAD Contest in Design Hierarchy Aware Routability-driven Placement and Benchmark Suite", *ICCAD* 2012, pp. 345-348. `cad_contest.cs.nctu.edu.tw/CAD-contest-at-ICCAD2012/problems/p2/p2.html`

[21] Y. Wei, C. Sze, N. Viswanathan, Z. Li, C. J. Alpert, L. N. Reddy, A. D. Huber, G. E. Terez, D. Keller and S. S. Sapatnekar, "GLARE: Global and Local Wiring Aware Routability Evaluation", *DAC* 2012, pp. 768-773.

[22] J. Westra and P. Groeneveld, "Is Probabilistic Congestion Estimation Worthwhile?" *SLIP* 2005, pp. 99-106.

[23] J. Y. Yen, "An Algorithm for Finding Shortest Routes From All Source Nodes to a Given Destination in General Networks", *Proc. Quarterly of Applied Mathematics* 27 (1970), pp.526-530.

[24] Y. Zhang and C. Chu, "CROP: Fast and Effective Congestion Refinement of Placement", *ICCAD* 2009, pp. 344-350.

[25] Y. Zhang and C. Chu, "GDRouter: Interleaved Global Routing and Detailed Routing for Ultimate Routability", *DAC* 2012, pp. 597-602.

Appendix A. Proof of Theorem 1

Let t be the $w \times h$ matrix, where $t[x][y]$ stores the cost of the optimal path from its cardinal neighbors. Consider the first pass where partial costs are not yet propagated. By construction, $t[x][y]$ only depends on $t[x-1][y]$ and $t[x][y-1]$, and requires one BF(Y) pass. Therefore, an optimal route r_π with $m = 1$ monotonic segments is found after $m = 1$ passes. Consider the general case where r_π has k distinct monotonic segments. By assumption, r_π is formed using k BFY passes. By the early termination criterion, if BFY changes no costs in t, then $r_\pi = r_\pi^{min}$. If relaxation is successful during the backward pass, then r_π is allowed to *detour* through some intermediate node v' such that the route cost from S to v is reduced by going through v'. If such v' exists, then there exists a new path from S to v through v' such that the new path has an additional monotonic segment. During the forward pass, the full path of S to T through v. If going through v reduces the cost, then there is an additional monotonic segment $v \rightarrow T$. Therefore, for two additional BFY passes for an r_π with k distinct monotonic segments, we will generate a new path with $k + 2$ monotonic segments. Because we consider all intermediate nodes v as detours, the best-cost path will be stored. Therefore, if r_π^{min} has $m = k + 2$ monotonic segments, it will require $k + 2$ BFY passes. □

Appendix B. Cell Inflation

We inflate each cell in the top 5% most congested GCells by computing its new width as follows.

$$\max\{\text{width}(cell) + 1, 1 + \theta(G) \cdot \Lambda(cell) \cdot \deg(cell)\} \quad (2)$$

Here, $cell$ is a movable cell in a congested GCell, $\text{width}(cell)$ and $\deg(cell)$ are the width and connectivity of $cell$, respectively. $\theta(G)$ is an adaptive function (described below) of the routing grid G, and $\Lambda(cell)$ is the number of times $cell$ has been in a congested GCell. We define θ similarly to [8, Equation 12], except that we upperbound θ to limit how much a cell can be inflated.

$$\theta = \min\{0.5, \max\{0, \alpha \cdot \eta(G) \cdot \xi(G) + \beta\}\} \quad (3)$$

Here, $\eta(G)$ and $\xi(G)$ represent the respective *difficulty* and *routability* of the design, where $\eta(G)$ is the sum of every GCell congestion in G, and $\xi(G)$ is the ratio of the total GCell congestion in G. α and β are constants based on linear regression. Unlike previous cell-inflation approaches [8], our formula *does not include* the GCell's congestion. By excluding the numeric congestion value, we only rely on the routing estimator's accuracy for congestion *locations*, and less on the reported congestion value. This prevents excessive inflation, and facilitates a smooth placement transition.

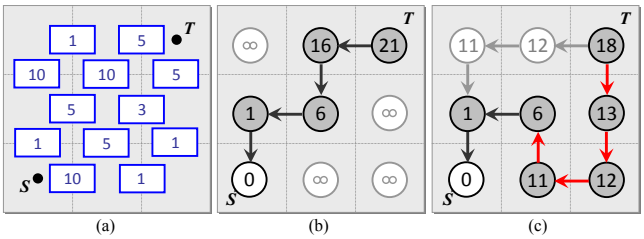

Figure 6: Applying BFY to an initial route for a point-to-point connection $S \rightarrow T$. (a) The routing grid and edge costs (congestion). (b) The initial route with cost 21. (c) Through relaxation, BFY can preserve part of the route, and find a better partial segment, resulting in a new route with cost 18.

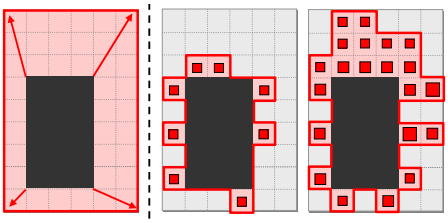

Figure 7: Congestion-driven rectangular macro expansion [5] (left) versus our technique (right).

Iter.	Total overflow (e5)				Comparison vs. maze		
#	maze	L	LZ	LIRE	L	LZ	LIRE
12	31.04	42.59	36.78	31.80	1.372	1.185	1.024
16	20.41	31.00	26.30	20.91	1.519	1.289	1.024
20	16.00	25.49	21.22	16.45	1.594	1.327	1.039
24	15.13	24.13	19.31	15.13	1.595	1.276	1.020
28	11.68	20.44	16.58	11.96	1.749	1.420	1.024
32	7.880	15.17	12.16	8.149	1.925	1.544	1.034
36	6.424	13.29	10.59	6.684	2.069	1.649	1.041
40	5.452	11.99	9.745	5.755	2.199	1.787	1.056
44	5.051	11.44	9.108	5.359	2.266	1.803	1.061
48	4.636	10.98	8.895	4.898	2.369	1.919	1.057
52	4.375	10.75	8.382	4.575	2.458	1.916	1.046
60	3.825	9.876	7.721	4.043	2.582	2.019	1.057
64	3.718	9.736	7.572	3.931	2.618	2.036	1.057
68	3.697	9.796	7.410	3.964	2.650	2.004	1.072
76	3.503	9.337	7.254	3.684	2.665	2.071	1.052
Avg					2.06×	1.65×	1.04×

Table 2: Total overflow estimation comparisons of L-routing, LZ-routing, the initial (maze) routing of BFG-R [4], and LIRE inside CoPR for the SUPERBLUE2 benchmark [19] (Figure 5f).

Benchmark	Quality metrics using BFG-R [4] (e8)			
	SimPLR	Ripple	NTUplace4	CoPR
SUPERBLUE1	3.023	3.341	2.962	3.084
SUPERBLUE3	3.803	3.906	4.609	3.757
SUPERBLUE4	2.865	2.659	2.773	2.530
SUPERBLUE5	3.980	3.654	3.919	3.646
SUPERBLUE7	4.479	4.502	4.283	4.439
SUPERBLUE10	8.114	7.080	7.810	7.378
SUPERBLUE16	3.117	2.929	3.032	2.989
SUPERBLUE18	2.461	2.207	1.838	2.163
Ratios (×)	1.06×	1.02×	1.03×	1.00×

Table 3: Quality metrics (based on BFG-R [4]) *without runtime* for the top three contestants as reported at the ICCAD 2012 Routability-driven Placement Contest [20] and CoPR.

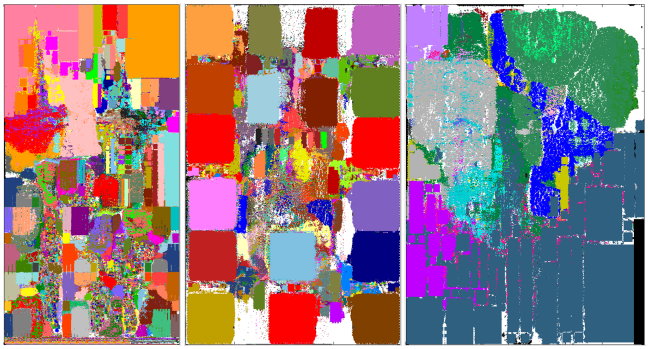

Figure 8: CoPR placements of the SUPERBLUE7 (left), SUPERBLUE10 (center), and SUPERBLUE18 (right) testcases [20].