A Practical Methodology for Early Buffer and Wire Resource Allocation

Charles J. Alpert, Jiang Hu, Sachin S. Sapatnekar*, Paul G. Villarrubia
IBM Corporation, 11400 Burnet Road, Austin, TX 78758
*University of Minnesota, ECE Dept., 200 Union SE, Minneapolis, MN 55455

Abstract
The dominating contribution of interconnect to system performance has made it critical to plan for buffer and wiring resources in the layout. Both buffers and wires must be considered, since wire routes determine buffer requirements and buffer locations constrain wire routes. In contrast to recent buffer block planning approaches, our design methodology distributes buffer sites throughout the layout. A tile graph is used to abstract the buffer planning problem while also addressing wire planning. We present a four-stage heuristic called RABID for resource allocation and experimentally verify its effectiveness.

1. Introduction
Interconnect's domination of system performance has made buffering a critical component in modern VLSI design methodologies. The number of buffers needed to achieve timing closure continues to rise with decreasing feature size. Achieving timing closure becomes more difficult when buffering is deferred until near the end of the design process; buffers must be squeezed into any remaining space. The problem is particularly acute for custom designs with large IP core macros and custom data flow structures that block out significant areas. ASIC designs can also run into similar headaches if they are dense, or have locally dense hot spots.

Buffers must be planned early in the design, so that the rest of the design flow may account for the required buffering resources. In addition, routability is also a critical problem; one must make sure that an achievable routing solution exists during the physical floorplanning stage. Global wiring must be planned to minimize routing congestion, hot spots, and crosstalk problems later in the flow.

1.1 Buffer Block Planning Methodology
A new body of research on *buffer block planning* has recently established itself [3][4][5][6][7] in response to these issues. These works focus on "physical-level interconnect planning" [2]. The works of [3][6][7] all propose creating "buffer blocks" (top-level macro blocks containing only buffers) that are inserted into the floorplan. Cong *et al*. [3] constructs buffer blocks using *feasible regions*. The feasible region for a net is the largest polygon in which a buffer can be inserted on the net such that the timing constraints are satisfied. Sarkar *et al*. [6] adds the notion of independence to feasible regions and also tries to relieve routing congestion. Tang and Wong [7] proposed an optimal algorithm assuming only one buffer per net. Finally, the multi-commodity flow-based approach of [4][5] allocates buffers to pre-existing buffer blocks. There are two key problems with buffer block planning:

1. Since buffers connect global wires, there will be contention for routing resources in the regions between macro blocks where buffer blocks are placed. The design may not be routable due to congestion between macro blocks.
2. Buffers sometimes must be placed in poor locations since the better locations are forbidden. Some macro blocks may be so large that routing over the block is infeasible, even if buffers are inserted immediately before and after the block, e.g., the signal integrity could degrade beyond recovery. Using wide wires on thick metal may help, but this further exacerbates the wiring congestion problem.

These problems are not a result of buffer block planning per se; rather, it is a reasonable approach for pre-planning buffers for current design flows. However, buffer block planning is really an interconnect-centric idea being applied to a device/logic-centric flow. Ultimately this methodology is not sustainable as design complexity continues to increase. A different methodology is required! Ideally, buffers should be dispersed throughout the design. Clumping buffers together, e.g., in buffer blocks or between abutting macros invites routing headaches. A more uniform distribution of buffers naturally spreads out global wires. *There must be a way to allow buffers to be inserted inside macro blocks.*

1.2 Buffer Site Methodology
We argue that block designers must permit global buffer and wiring resources to be interspersed wherever possible. This resource allocation need not be uniform; a low performance block may be able to allocate more resources for buffer sites than a high performance block. Meanwhile, a cache may not be able to allocate any of its resources. Ideally, as this "hole in a macro" methodology becomes widespread, future IP blocks will have to contain buffer sites.

A designer can allocate a buffering resource within a block by inserting a *buffer site*, i.e., a physical area that can denote either a buffer, inverter, or decoupling capacitor. Until a buffer site is assigned to a net, no logical gate from the technology is inserted, nor is the buffer site connected to any net.

Unused buffer sites can still be utilized in other ways. They can be populated with spare circuits to facilitate late metal-only engineering changes or with decoupling capacitors to enhance local power supply and signal stability. One can afford to allocate more buffer sites than will ever be used.

Buffer sites can also be used within data paths. A data path typically contains regular signal buses routed across collections of data path elements. If the strands of the data bus require buffers, one needs buffer locations within the data path itself. Designing buffer sites into the original data path layout makes it possible to add buffers late in the design cycle while maintaining straight wiring of the buses.

Buffer sites can also be used for flat designs, e.g., a "sea of buffer sites" can be sprinkled throughout the placement. For hierarchical designs, one can flatten the buffer sites to derive a similar sprinkling. The flat view enables a resource allocation algorithm to make assignments to global routes based on buffer site distribution.

2. Problem Formulation

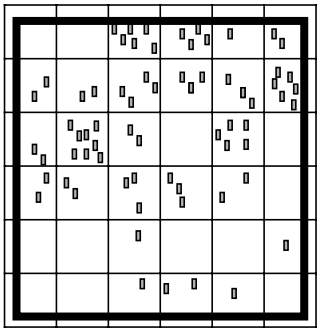
There are two fundamental characteristics of buffer and wire planning which drive our formulation.

1. Finding the absolute optimal location for a buffer is not necessary. Cong *et al.* [3] showed that one can move a buffer considerably from its ideal location and incur only a small delay penalty.
2. At the floorplanning stage, net by net timing constraints are not available since macro block designs are incomplete and global routing and extraction have not been performed. Timing analysis could be performed, but the results can be grossly pessimistic because interconnect synthesis has not occurred. One needs to globally insert buffers while tracking wire congestion to enable floorplan evaluation. For example, in a 200 Mhz design, say that floorplans A and B have worst slacks of -40 and -43 ns, respectively. The designer cannot determine the better floorplan because the slacks are both absurdly far from their targets. Only after buffer and wire planning are performed can the design be timed to provide a meaningful worst slack value.

The first characteristic suggests that precise buffer site locations are unnecessary. Designers can freely sprinkle buffer sites into their blocks so that performance is not compromised; there just needs to be enough buffer sites altogether. We use a *tile graph* to represent the buffer sites, which can potentially run into the thousands. Figure 1(a) shows a chip containing 68 buffer sites and Figure 1(b) abstracts each individual buffer site as one of a set of buffer sites at the tile's center. After a buffer is assigned to a particular tile, an actual buffer site within the tile can be allocated. The tile graph reduces complexity and can be used to manage wire congestion across tile boundaries. The granularity of the tiling depends on the desired accuracy/runtime trade-off and the current stage in the design flow.

The tile graph $G(V, E)$ for a set of tiles V contains edge $e_{u,v}$ if u and v are neighboring tiles. For a tile v, let $B(v)$ be the number of buffer sites within v. The set of global nets is given by $N = \{n_1, n_2, \ldots, n_m\}$. Let $W(e_{u,v})$ be the maximum number of wires that may cross between u and v without causing overflow. If $b(v)$ denotes the number of buffers assigned to v, the buffer congestion for v is given by $b(v)/B(v)$. Similarly, let $w(e_{u,v})$ denote the number of wires which cross between u and v. The wire congestion for $e_{u,v}$ is given by $w(e_{u,v})/W(e_{u,v})$.

The second characteristic suggests that timing constraints are not dependable in the early floorplanning stage. Our formulation relies on a global rule of thumb for the maximum distance between consecutive buffers which was also used by Dragan *et al.* [4]. They note that for a high-end microprocessor design in 0.25µm CMOS technology, repeaters are required at intervals of at most 4500µm to ensure a sufficiently sharp slew rate at the input to all gates.



0	0	6	4	1	2
2	2	4	3	3	6
2	8	2	0	5	0
2	2	3	3	2	0
0	0	1	0	0	1
0	0	1	2	1	0

(a) (b)

Figure 1 (a) A set of 68 buffer sites can be tiled and (b) abstracted to the total number of buffer sites lying within each tile.

For net n_i, let L_i be the maximum number of tiles that can be driven by either the source of n_i or a buffer inserted on n_i. One might alternatively constrain L_i as the tile distance from a source to a sink, but this causes the problem shown in Figure 2. The figure shows a six-sink net where the distance from the driver to each sink is three tiles, yet the source gate drives nine tiles of wire. Our more restrictive constraint will force buffers to be inserted on this net, which are likely needed to fix weak slew rates at the sinks.

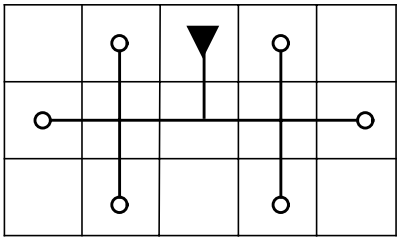


Figure 2 A six-sink net where each source-sink path has length three, yet the source must drive nine tiles of wire.

Problem Formulation: Given a tiling $G(V, E)$ of the chip, nets $N = \{n_1, n_2, \ldots, n_m\}$, buffer sites $B(v)$, and length constraints L_i, assign buffers to nets such that

- $b(v) \le B(v)$ for all $v \in V$, where $b(v)$ is the number of buffers assigned to tile v.
- Each net $n_i \in N$ satisfies its tile length constraint, L_i.[1]
- There exists a routing after buffering such that for all $e_{u,v} \in E$, $w(e_{u,v}) \le W(e_{u,v})$.

[1] One typically uses the same value of L_i for each net; however, nets on higher metal layers will have larger L_i values. Also, a larger value of L_i can be used with wider wire width assignment.

The formulation seeks a solution which satisfies constraints, though secondary objectives can also be optimized (e.g., total wirelength, wire congestion, buffer congestion, and timing). Our heuristic seeks a solution which satisfies the formulation while also minimizing secondary objectives.

Note that the formulation's purpose is not to find the final buffering and routing of the design. Rather, it can be used to estimate buffering and routing resources or as a precursor to timing analysis for floorplan evaluation. Once deeper into physical design, suboptimal or timing-critical nets should be re-optimized using more accurate timing constraints and wiring parasitics.

3. Buffer and Wire Planning Heuristic

We propose a heuristic called RABID (Resource Allocation for Buffer and Interconnect Distribution) which proceeds in four stages: (1) initial Steiner tree construction, (2) wire congestion reduction, (3) buffer assignment, and (4) final post-processing. The algorithm's innovations are contained in the last two stages which handle buffer site assignment. Stages 1 and 2 use traditional rip-up and re-route to deliver an initial congestion-aware global routing solution. One could alternatively start with the solution from any congestion-aware global router.

3.1 Stage 1: Initial Steiner Tree Construction

The purpose of this stage is to construct an initial routing of each net. The route should be timing-driven, yet timing constraints are not necessarily available. We adopt the Prim-Dijkstra construction [1] which generates a hybrid between a minimum spanning tree and shortest path tree. The result is a spanning tree which trades off between radius and wirelength. The spanning tree is then converted to a Steiner tree via a greedy overlap removal algorithm. The algorithm iteratively searches for the two tree edges with the largest potential wirelength overlap. A Steiner point is introduced to remove the overlap. The algorithm terminates when no further overlap removal is possible.

3.2 Stage 2: Wire Congestion Reduction

The purpose of this stage is to rip-up-and-reroute the initial Steiner trees to reduce wire congestion. We first construct the tile graph $G(V, E)$ from the existing Steiner routes and compute $w(e_{u,v})$ for each edge $e_{u,v} \in E$. Instead of ripping up only nets in congested regions, we rip-up and re-route every net, as in [6]. Each net is processed in turn according to a fixed net ordering (sorted from shortest to longest delays). This allows nets which do not actually violate congestion constraints to be further improved, thereby helping subsequent nets that do violate constraints to be successfully re-routed. The algorithm terminates when $w(e_{u,v})/W(e_{u,v}) \leq 1$ for all $e_{u,v} \in E$ or after three complete iterations. We observe only nominal potential improvement exists after the third iteration.

A net is re-routed by first deleting the entire net, then re-routed using an approach similar to [2], as opposed to re-routing one edge at a time. The new tree is constructed on the tile graph using the same Prim-Dijkstra algorithm from Stage 1, except the cost function for each edge is no longer Manhattan distance. The routing occurs across the tile graph using the following congestion-based cost function:

$$\text{cost}(e_{u,v}) = \frac{w(e_{u,v}) + 1}{W(e_{u,v}) - w(e_{u,v})} \quad \text{if} \quad \frac{w(e_{u,v})}{W(e_{u,v})} < 1 \qquad (1)$$

and $\text{cost}(e_{u,v}) = \infty$ otherwise. The cost is the number of wires that would be crossing $e_{u,v}$ divided by the number of available tracks. The cost function increases the penalty as the edge comes closer to full capacity. The re-routing procedure performs a wave-front expansion from the source tile, updating to the lowest tile cost with each expansion. The procedure terminates after reaching each sink, and the tree is recovered by tracing back to the source from each sink.

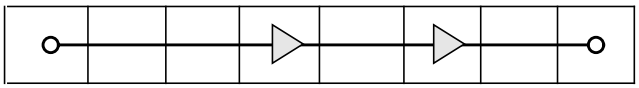
3.3 Stage 3: Buffer Assignment

This stage allocates buffers to each net. We perform this assignment one net at a time in order of net delay, starting with the longest delay. Before assigning buffers, we first estimate the probability of a net occupying a tile. For a net n_i crossing tile v, the probability of a buffer site from v being used for n_i is given by $1/L_i$. Let $p(v)$ be the sum of these probabilities for tile v over all unprocessed nets. The cost $q(v)$ for using a particular buffer site is defined as

$$q(v) = \frac{b(v) + p(v) + 1}{B(v) - b(v)} \quad \text{if} \quad \frac{b(v)}{B(v)} < 1 \qquad (2)$$

and $q(v) = \infty$ otherwise. Observe the similarity between Equations (2) and (1). Both significantly increase the cost penalty as resources become more contentious.

Figure 3 shows an example buffer cost computation. Note that the $p(v)$ values do not include the currently processed net. The cost $q(v)$ is computed for each tile, and $q(v)$ is included in the net's cost if a buffer is inserted at v. If $L_i = 3$, the minimum cost solution has buffers in the third and fifth tiles, resulting in a total cost of $0.5 + 1.0 = 1.5$.



B(v)	8	5	12	3	5	0
b(v)	3	4	2	3	0	0
p(v)	2.5	3.6	2	0.8	4	5
q(v)	1.3	8.6	0.5	∞	1.0	∞

Figure 3 Example of buffer cost computation. For $L_i = 3$, the optimal solution is shown, having total cost 1.5.

An optimal solution can be revealed in linear time (assuming that L_i is constant). The approach uses a van Ginneken (VG) [8] style dynamic programming algorithm, but has lower time complexity because the number of candidates for each node is at most L_i.

We begin with a net n_i with source s and a single sink t. Let $par(v)$ be the parent tile of each tile v in the route. and assume that $q(v)$ has been computed for all tiles on the path from s to t. For each v, the array C_v stores the costs of the solutions from v to t. The index of the array determines the distance downstream from v to the most recently

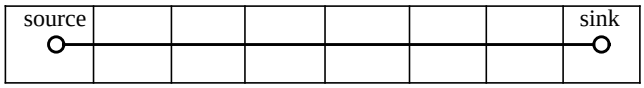
inserted buffer. Thus, the array is indexed from 0 to $L_i - 1$, since v cannot be at distance more than L_i from the last available buffer. The full algorithm is shown in Figure 4.

Step 1 initializes the cost array C_t to zero for sink t. Step 2 iteratively computes the cost array for $par(v)$ given the cost for v. The value of $C_{par(v)}[j]$ for $j > 0$ is simply $C_v[j-1]$ since no buffer is being inserted at v for this case. If a buffer is to be inserted at $par(v)$, then the cost $C_{par(v)}[0]$ is the sum of the current cost for insertion, $q(par(v))$ and the lowest cost seen at v. Step 3 returns the lowest cost; the solution can be recovered by storing at $par(v)$ the index in C_v used to generate the solution.

1. Set $C_t[j] = 0$ for $1 \le j < L_i$ and sink t. Set $v = t$.
2. while $v \neq s$ do
   for $j = 1$ to $L_i - 1$ do
   Set $C_{par(v)}[j] = C_v[j-1]$
   Set $C_{par(v)}[0] = q(par(v)) + min\{C_v[j] \parallel 0 \le j < L_i\}$
   Set $v = par(v)$.
3. Let v be such that $par(v) = s$.
   Return $min\{C_v[j] \parallel 0 \le j < L_i\}$.

Figure 4 Single-sink buffer insertion algorithm.

Figure 5 shows how the cost array is computed for the 2-pin example in Figure 3 (with $L_i = 3$) and the dark lines show how to trace back the solution. Observe from the table that costs are shifted down and to the left as one moves from right to left, with the exception of entries with index zero.



q(v)	1.3	8.6	0.5	∞	1.0	∞	
$C_v[0]$	2.8	9.6	1.5	∞	1.0	∞	0
$C_v[1]$	9.6	1.5	∞	1.0	∞	0	0
$C_v[2]$	1.5	∞	1.0	∞	0	0	0

Figure 5 Execution of the single source algorithm on the example in Figure 3. The optimal solution has cost 1.5 and the dark lines show how this cost is obtained.

Optimality follows from the fact that once a buffer is inserted (i.e., a $C_{par(v)}[0]$ is computed), only the best solution downstream from the buffer needs to be recorded. Since the number of possible candidates at each tile is no more than L_i a space and time complexity of $O(nL_i)$, is obtained, where n is the number of tiles spanned by the net. This is a significant advantage over similar approaches [6][8][8] which have at least $O(n^2)$ time complexity.

For multi-sink nets, one still keeps a cost array at each tile, but updating the cost becomes a bit trickier when a tile has two children. For this case, there are three possible scenarios as shown in Figure 6. A buffer may be used to either (a) drive both branches, (b) decouple the left branch, or (c) decouple the right branch.[2] Let $l(v)$ and $r(v)$ denote the two children of v. If v has only one child, let it be $l(v)$.

Figure 7 presents the complete algorithm. The algorithm is equivalent to Figure 5, except Step 4 handles multiple children. Step 4.1 computes costs when no buffer is inserted at v. Since one tile of wire is driven for both the left and right branches, no buffering at v implies that the cost array only should be updated for $j \ge 2$. Step 4.2 considers the case of Figure 6(a) where a buffer drives both children. Step 4.3 initializes the cost array for index 1, and finally, Step 4.4 updates the cost array with a potentially lower cost from decoupling either the left or right branches. The multi-sink variation has $O(nL_i^2)$ time complexity, due to Step 4.2.

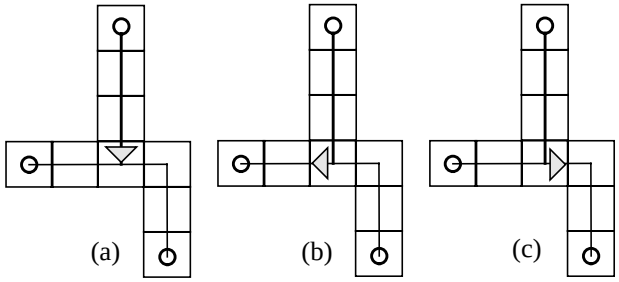


Figure 6 For a tile with two children, a buffer can either (a) drive both branches or (b) decouple the left ((c) right) branch.

1. Pick unvisited v s. t. all descendents of v have been visited.
   while $v \neq s$ do
2. if v is a sink then
   Set $C_v[j] = 0$ for $1 \le j < L_i$.
3. if v has one child $l(v)$ then
   for $j = 1$ to $L_i - 1$ do
   Set $C_v[j] = C_{l(v)}[j-1]$
   Set $C_v[0] = q(v) + min\{C_{l(v)}[j] \parallel 0 \le j < L_i\}$
4. if v has two children $l(v)$ and $r(v)$ then
4.1 for $j = 2$ to $L_i - 1$ do
   $C_v[j] = min\{C_{l(v)}[j_l] + C_{r(v)}[j_r] \parallel j_l + j_r + 2 = j\}$
4.2 $C_v[0] = min\{C_{l(v)}[j_l] + C_{r(v)}[j_r] \parallel j_l + j_r + 2 \le L_i\}$
   Set $C_v[0] = C_v[0] + q(v)$
4.3 $C_v[1] = \infty$
4.4 for $j = 1$ to $L_i - 1$ do
   Let $D_v = min\{C_{l(v)}[j-1], C_{r(v)}[j-1]\}$
   Set $C_v[j] = min\{C_v[j], q(v) + D_v\}$
5. mark v as visited
   pick unvisited v s. t. all descendents of v have been visited.
6. Return $min\{C_s[j] \parallel 0 \le j < L_i\}$.

Figure 7 Multi-sink buffer insertion algorithm.

3.4 Stage 4: Final Post-Processing

The last stage attempts to reduce buffer and wire congestion and the number of nets which fail to meet their length constraint. Using the same flow as in Stage 2, each net is ripped up and re-routed, and the buffers are also removed. However, for multi-pin nets, the net is ripped up one *two-path* at a time, where a two-path is a path in the tree which begins and ends at either a Steiner node, source, or sink and contains only vertices of degree two. The ends of the two-path are then reconnected (using a maze routing algorithm) via the path that minimizes the sum of wire and buffer

[2] A tile could have up to three children yielding seven different scenarios. This case is a straightforward extension of Figure 7.

congestion costs (Eqs. (1) and (2)).

4. Experimental Results

We implemented our heuristic in C++ on an RS6000/595 machine with 1 Gb of memory. We tested our code on ten benchmarks which we obtained from the authors of [3] and embedded the designs in the same 0.18μm technology used in [3]. The first six circuits are from the Collaborative Benchmarking Laboratory and the other four were randomly generated. The circuits' characteristics are summarized in Table 1. The nets and sinks columns present slightly smaller values than in [3] since they reflect only the nets on which Cong *et al*. actually inserted buffers.

test case	blocks	nets	pads	sinks	grid size	tile area	L_i	buffer sites	%chip area
apte	9	77	73	141	30x33	0.36	6	1200	0.13
xerox	10	171	2	390	30x30	0.35	5	3000	0.38
hp	11	68	45	187	30x30	0.42	6	2350	0.25
ami33	33	112	43	324	33x30	0.46	5	2750	0.24
ami49	49	368	22	493	30x30	0.67	5	11450	0.75
plyout	62	1294	192	1663	33x30	0.75	6	27550	1.47
ac3	27	200	75	409	30x30	0.49	6	3550	0.32
xc5	50	975	2	2149	30x30	0.54	6	13550	1.11
hc7	77	430	51	1318	30x30	1.04	5	7780	0.33
a9c3	147	1148	22	1526	30x30	1.08	5	12780	0.52

Table 1: Test circuit characteristics and parameters.

4.1 General Performance

First, we study the performance of each of RABID's four stages. The grid size and number of buffer sites are shown in Table 1. We chose the grid size to have 30 tiles on the shorter side of the chip, then derived the number of tiles for the longest side, so that each tile was roughly square. The tile area is given in square millimeters; except for the last two random test cases, no tile is more than one millimeter long on a side. The number of buffer sites for each test case was chosen so that the total chip area occupied by buffer sites was less than 2%. For each test case, a random nine by nine set of tiles were prohibited from having any inserted buffer sites to correspond to a large cache-like block. The buffer sites were randomly distributed among the other tiles.

The floorplans were supplied by the authors of [3] and were generated from the output from their buffer block planning, with the buffer blocks. removed. The results for each stage and each CBL benchmark are summarized in Table 2. We present only the cumulative results for the four random circuits. The statistics presented are:

- the maximum wire congestion (MWC) over all $e_{uv} \in E$,
- the maximum buffer congestion (MBC) over all tiles,
- wiring overflows (OV), i.e., the sum over all $e_{uv} \in E$ of $w(e_{uv}) - W(e_{uv})$, for whenever $w(e_{uv}) > W(e_{uv})$,
- the number of buffers inserted (bufs),
- the number of nets for which the tile length constraint was not satisfied (fail),
- total wirelength in millimeters,
- maximum and average delay (picoseconds) to each sink,
- and CPU time in seconds.

test case		MWC	MBC	OV	bufs	fail	wire length	delay		cpu (s)
								max	avg	
apte	1	2.00	0.00	225	0	77	1410	5029	1700	0
	2	0.62	0.00	0	0	77	1706	5390	2156	12
	3	0.62	1.00	0	401	7	1706	3256	959	1
	4	0.62	1.00	0	364	5	1718	1854	863	34
xerox	1	2.00	0.00	466	0	171	2537	3361	1529	1
	2	0.80	0.00	0	0	171	3449	5836	2010	25
	3	0.80	1.00	0	1032	6	3449	3216	1326	1
	4	0.60	1.00	0	800	12	3028	1594	692	72
hp	1	3.25	0.00	368	0	68	1405	5672	1995	0
	2	1.00	0.00	0	0	68	1818	6723	2440	11
	3	1.00	1.00	0	432	8	1818	4794	832	0
	4	0.75	1.00	0	335	9	1699	3483	815	38
ami33	1	2.50	0.00	365	0	112	2471	9016	4413	1
	2	1.00	0.00	0	0	112	3028	12735	5690	22
	3	1.00	1.00	0	798	6	3028	5379	1297	1
	4	0.83	1.00	0	704	6	2952	2553	1094	43
ami49	1	2.18	0.00	887	0	368	5881	7601	1730	0
	2	1.00	0.00	0	0	368	8720	28784	3200	54
	3	1.00	0.90	0	1887	17	8720	21150	1164	1
	4	1.00	0.75	0	1277	10	7075	3884	875	103
plyout	1	1.19	0.00	230	0	1294	22555	8633	1989	1
	2	0.34	0.00	0	0	1294	29520	19160	2789	275
	3	0.34	1.00	0	4542	125	29520	12446	1253	6
	4	0.44	1.00	0	3716	44	25881	3405	937	402
ac3	1-4	0.67	1.00	0	855	24	4640	2982	888	137
xc5	1-4	0.90	1.00	0	2941	28	17022	2415	777	520
hc7	1-4	1.00	1.00	0	2015	61	13512	4944	1128	277
a9c3	1-4	1.00	1.00	0	4193	39	28945	3895	1183	714

Table 2: Stage by stage experimental results for ten test circuits. Only summaries are shown for the random test cases.

Since no timing constraints are used, average and maximum source-to-sink delays are reported to give an indication for the timing quality. We make several observations:

- The wire congestion constraint is always satisfied. In Stage 1, which ignores wire congestion, the maximum wire congestion is typically a factor of two to three above capacity and there are several hundred overflows.
- The algorithm never violates buffer site constraints, but typically utilizes at least one tile to full buffer capacity.
- The number of buffers, fails, and delays all improve from Stage 3 to Stage 4, illustrating the effectiveness of post-processing. The number of nets which fail to meet the length constraint is typically small, but not zero. These fails typically cannot be removed due to the large

9 by 9 region with no buffer sites.

4.2 Comparisons with Buffer Block Planning

Our next experiments attempt to compare RABID with the BBP/FR buffer block planning algorithm [3], though RABID does not use buffer blocks. Hence, one cannot simply compare to previously published results. Instead, we obtained code from the authors of [3] and implemented routines to gather statistics from the data. Our results were generated using the same number of buffer sites as in Table 1, but without the 9 by 9 region of blocked tiles.

test case	Algo-rithm	MWC	MTP	OV	bufs	wire length	delay		cpu (s)
							max	avg	
apte	BBP/FR	2.00	2.87	23	233	1827	2026	721	14
	RABID	1.00	0.33	0	417	2010	1935	787	95
xerox	BBP/FR	1.15	5.57	14	508	4096	1486	575	29
	RABID	0.93	0.57	0	957	4541	1531	643	167
hp	BBP/FR	2.50	1.89	107	264	2194	1948	645	16
	RABID	0.83	0.28	0	450	2403	2029	707	67
ami33	BBP/FR	1.19	3.31	34	654	4923	2329	852	44
	RABID	0.69	0.44	0	1150	5232	2256	900	138
ami49	BBP/FR	4.64	4.15	1034	862	6787	2359	768	65
	RABID	0.93	0.36	0	1339	7592	2635	859	167
plyout	BBP/FR	0.99	10.34	0	3422	25930	2727	880	198
	RABID	0.45	0.64	0	3840	27601	3310	947	813
ac3	BBP/FR	1.23	2.86	37	718	5586	1928	763	87
	RABID	0.58	0.33	0	1037	5954	2095	807	208
xc5	BBP/FR	4.70	16.40	3528	3186	25241	2194	655	181
	RABID	0.84	0.81	0	4410	27060	2343	700	694
hc7	BBP/FR	3.82	4.88	1363	2684	20011	2935	861	174
	RABID	0.82	0.35	0	2983	21523	3349	941	386
a9c3	BBP/FR	2.54	4.79	1329	4041	29060	2726	1093	222
	RABID	0.60	0.44	0	4225	30723	2786	1170	502

Table 3: Comparisons of RABID to BBP/FR [3].

As in [3], but unlike the experiments in Table 2, we decomposed each multi-pin net into several 2-pin nets. Cong *et al*. [3] report timing results by measuring the number of nets which fail to meet their delay constraint and chose the timing constraint to be between 1.05 and 1.20 of the optimum achievable delay. This constraint generation is unrealistic since they imply that all constraints are tight, yet potentially satisfiable. In practice, some of the 1.05x-1.20x timing constraints will be so tight that buffer insertion is insufficient to satisfy timing. For these cases, feasible regions are not well defined. In addition, some nets may have such loose constraints that detours can be taken while still meeting delay constraints. Since RABID and BBP/FR have different criteria for buffer insertion, we use source-sink delays to quantify timing performance.

Table 3 presents comparisons with BBP/FR. The statistic MTP (maximum tile percentage) is the maximum percent, over all tiles, of the tile area occupied by inserted buffers (e.g., for ami49, buffers inserted by BBP/FR occupy 4.15% of the area of one of the tiles). We observe the following:

- RABID always satisfies the wire congestion constraints while BBP/FR does not. The BBP/FR results even include a post-processing step which tries to minimize congestion for the current buffering solution without increasing wirelength.
- RABID inserts more buffers than BBP/FR due to wire congestion avoidance.
- Because our methodology invites spreading, MTP is significantly less for RABID. In the worst case, BBP/FR has one tile with 16.40% of its area devoted to buffers, while RABID never has an MTP higher than 0.81%.
- The CPU time for BBP/FR is significantly less. Stages 2 and 4 of our algorithm cause much greater runtimes, but they are not prohibitive.
- The delays for RABID are quite comparable to those of BBP/FR despite using a length-based algorithm and satisfying wire congestion constraints. In some cases the delays are even better.

5. Conclusions

We proposed a methodology for buffer and wire resource allocation that uses pre-distributed buffer sites. This enables one to model the problem via a tile graph and also plan both wires and buffers. Our four stage RABID heuristic includes a novel algorithm for length-based buffer insertion. Experimental results show that RABID generates effective solutions in terms of several practical criteria.

Acknowledgments

The authors sincerely thank Jason Cong, David Pan, and Tianming Kong for not only supplying code and circuits but for also running experiments and helping with debugging.

References

[1] C. J. Alpert, T. C. Hu, J. H. Huang, A. B. Kahng and D. Karger, “Prim-Dijkstra Tradeoffs for Improved Performance-Driven Routing Tree Design”, *IEEE Trans. on Comput.-Aided Design*, 14(7), 1995, pp. 890-896.

[2] J. Cong, “An Interconnect-Centric Design Flow for Nanometer Technologies”, *Int. Symp. on VLSI Technology, Systems, and Applications*, Taipei, Taiwan, June 1999, pp. 54-57.

[3] J. Cong, T. Kong and D.Z. Pan, “Buffer Block Planning for Interconnect-Driven Floorplanning”, *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design*, 1999, pp. 358-363.

[4] F. F. Dragan, A. B. Kahng, I. Mandoiu, S. Muddu, “Provably Good Global Buffering Using an Available Buffer Block Plan”, *IEEE/ACM ICCAD*, 2000, pp. 104-109.

[5] F. F. Dragan, A. B. Kahng, I. Mandoiu, S. Muddu, “Provably Good Global Buffering by Multiterminal Multicommodity Flow Approximation”, *ASP-DAC*, 2001, pp. 120-125.

[6] P. Sarkar, V. Sundararaman and C.-K. Koh, “Routability-Driver Repeater Block Planning for Interconnect-Centric Floorplanning”, *Intl. Symp. Phys. Design*, 2000, pp. 186-191.

[7] X. Tang and D.F. Wong, “Planning Buffer Locations by Network Flows”, *Intl. Symp. Physical Design*, 2000, pp. 180-185.

[8] L.P.P.P. van Ginneken, “Buffer Placement in Distributed RC-tree Networks for Minimal Elmore Delay”, *Proc. IEEE Int. Symp. Circuits Syst.*, 1990, pp. 865-868.