Static and Dynamic Memory Footprint Reduction for FPGA Routing Algorithms

SCOTT Y. L. CHIN and STEVEN J. E. WILTON
University of British Columbia

This article presents techniques to reduce the static and dynamic memory requirements of routing algorithms that target field-programmable gate arrays. During routing, memory is required to store both architectural data and temporary routing data. The architectural data is static, and provides a representation of the physical routing resources and programmable connections on the device. We show that by taking advantage of the regularity in FPGAs, we can reduce the amount of information that must be explicitly represented, leading to significant memory savings. The temporary routing data is dynamic, and contains scoring parameters and traceback information for each routing resource in the FPGA. By studying the lifespan of the temporary routing data objects, we develop several memory management schemes to reduce this component. To make our proposals concrete, we applied them to the routing algorithm in VPR and empirically quantified the impact on runtime memory footprint, and place and route time.

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

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: FPGA, routing, scalability, memory, CAD

ACM Reference Format:
Chin, S. Y. L. and Wilton, S. J. E. 2009. Static and dynamic memory footprint reduction for FPGA routing algorithms. ACM Trans. Reconfig. Techn. Syst. 1, 4, Article 18 (January 2009), 20 pages. DOI = 10.1145/1462586.1462587. http://doi.acm.org/10.1145/1462586.1462587.

1. INTRODUCTION

Advances in process technology and architectural enhancements have led to a dramatic increase in the capacity of field-programmable gate arrays (FPGAs). This has allowed FPGAs to implement entire systems and has opened up new markets for FPGAs. However, this scaling places increasing demands on the FPGA CAD tools. Already, for very large designs, compile times of several days

The authors gratefully acknowledge the support of Altera Corporation, and the NSERC of Canada.
Authors' address: University of British Columbia, 2332 Main Mall, Vancouver, BC, V6T 1Z4 Canada; email: scottc@ece.ubc.ca.

are common, and memory requirements that exceed what would be found in a common desktop workstation are the norm. As FPGAs continue to grow, the problem will become worse. Unless the scalability of FPGA CAD tools is addressed, the long runtimes and large memory footprints will become a hindrance to future FPGA scaling, leading to increased costs and design times for companies which use these devices.

Reducing the compile time for CAD tools is an active area of research. Approaches include reducing the solution space explored [Sankar and Rose 1999; Mulpuri and Hauck 2001], parallelization [Chan et al. 2000; Haldar et al. 2000], and hardware acceleration [DeHon et al. 2002]. These works generally do not consider the memory footprint of the CAD tools. Unlike a CAD tool targeting an ASIC, a CAD tool targeting an FPGA must be aware of every programmable switch and routing resource on the device. As devices grow, the memory footprint needed to store this information grows. Already, CAD tools from some vendors require 64-bit machines to map circuits to large devices due to the memory requirements of the placement and routing algorithms. Other vendor tools run on 32-bit machines, but increasing FPGA size will mean even these will soon require larger, more expensive workstations.

Reducing the memory requirements of FPGA CAD tools are important for a number of reasons:

(1) Some users typically have access to state-of-the art workstations. However, many designers are designing circuits for FPGAs using low-cost workstations that contain limited amount of RAM. FPGAs provide a low-cost low-risk entry to integration; this advantage is somewhat reduced if FPGA CAD tools require expensive workstations. As a concrete example, academic users performing FPGA architecture experiments often have a large bank of low-cost workstations to perform repeated place and route experiments. In such an environment, it is far more important to be able to compile a benchmark circuit with the systems available; the runtime of each experiment is secondary.
(2) When a workstation has insufficient physical memory to perform a compile, it begins to use the swap memory on the hard disk. Accessing the hard disk is much slower than accessing RAM memory. If the process is constantly reading and writing memory to the hard drive (thrashing), then the perceived speed of the workstation is drastically reduced. Furthermore, if the requirements exceed the amount that can be addressed by the operating system (4GBytes for 32-bit machines), then the tool cannot be executed at all.
(3) One of the most promising approaches to reduce compile time is to parallelize the CAD algorithms. One of the challenges with parallelization is coordinating access to memory. It may be that small data structures that can be replicated across processors are more suitable than a single large data structure that must be shared by all processors.

One of the most memory-intensive steps in the FPGA CAD flow is routing [Lysecky et al. 2004]. The goal of the router is to efficiently use the

prefabricated routing resources to implement each connection in a circuit. In order to do this, the router needs to store at least two types of data in its execution. First, the router needs to store architectural data. This data includes a map of all physical routing resources available on the FPGA. As the number of programmable elements and wiring tracks increase, this map grows very quickly in size, and tends to be a dominating factor in the overall memory footprint of the FPGA CAD flow. This storage is *static*, in that it does not change while routing a netlist. Second, the router needs to maintain various scoring and traceback information for each routing resource. This data also has poor scalability since it is required for every single routing resource. However, it is much smaller than the architectural data requirements. This storage is *dynamic* since the amount of data that must be maintained changes over time during the routing process. Additional storage is required, for example, for the priority queue used in wavefront expansion; however, this is small and will not be considered in this article.

In this article, we focus on improving the scalability of both the static and dynamic memory components in an FPGA router. Initial work on reducing the static architectural data memory requirements was published in Chin and Wilton [2007]. This paper expands on that study by considering the dynamic temporary routing data as well. We present a new study on the lifespan of the temporary routing data objects. Based on this study, we propose several schemes that reduce the peak memory requirements of the temporary routing data. Our ideas can apply to any FPGA router but to make our ideas concrete, we have implemented our ideas in the commonly used academic Versatile Place and Route (VPR) tool [Betz et al. 1999]. Our modified version of the VPR tool is available to the research community.

This article is organized as follows. Section 2 provides background material as well as related work. Section 3 presents our method for improving the scalability of the architectural data. Section 4 presents the results for this technique. Section 5 presents a study on the temporary routing data and a technique to reduce the memory requirements of this temporary data. Section 6 quantifies the memory savings using this technique, and Section 7 concludes the article.

2. BACKGROUND

In this section, we describe how the architectural data is represented in an FPGA router, as well as related work. Although our discussions focus on the VPR router, other FPGA routers would have similar storage requirements.

2.1 Routing Resource Graphs

The map of the physical segments and programmable switches can be represented by a Routing Resource Graph (RRG) [Betz et al. 1999]. In the VPR router, the RRG is a directed graph where each wire and logic pin on the FPGA is represented by a node. In addition to wire and pin nodes, there are also source and sink nodes to model pins that are logically equivalent. Programmable connections between the resources are represented by directed edges.

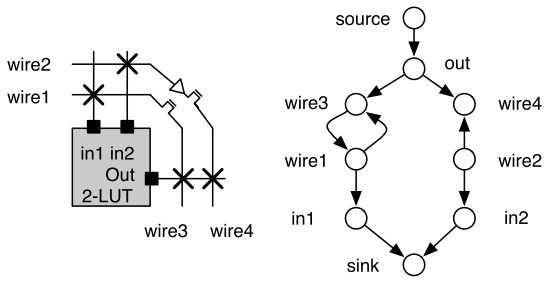

Fig. 1. Example portion of the routing resource graph.

Directional connections such as tristate buffers are modeled by a single edge, and bidirectional connections such as pass transistors are modeled by a pair of edges. Figure 1 shows an example of the RRG for one logic block tile containing a 2-input lookup table (LUT) and a channel width of 2.

In the VPR router, each node of the RRG includes not only connectivity information, but also coordinates for mapping each node to its corresponding physical resource, information for the timing model (capacitance and resistance of each physical resource), and the capacity of each node.

2.2 Related Work

There have been only a few papers focusing on memory reduction. The most relevant proposes a Just-in-Time (JIT) compiler for FPGAs, which configures an FPGA when it is about to execute [Lysecky et al. 2004]. Their routing algorithm uses a simplified graph where each node represents a switch box, and edges represent connections between the switch boxes. Although their technique produces a significant reduction in memory footprint, it only applies to simplified architectures aimed for JIT compilation. Carroll and Ebeling [2006] improve the memory scalability of the QuickRoute [Li and Ebeling 2004] algorithm used to route on FPGAs with pipelined routing architectures. They employ a range encoding algorithm to compress the bit-vectors used to store path exploration history.

3. ARCHITECTURAL DATA

In this section, we present a new representation for the static architectural data that requires significantly less storage.

Our technique takes advantage of the tiled nature of FPGAs. Rather than explicitly representing all the tracks and programmable resources on the FPGA, we divide our FPGA into tiles. Each tile corresponds to a small number of tile types, where all tiles within a tile type are identical (with the possible exception of the starting and ending point of wires that span more than one tile). For example, a tile type might contain a single logic block and its surrounding routing channels. An FPGA with embedded blocks would contain additional tile types consisting of the embedded block and its surrounding routing. Additional tiles are required for the ring of I/O blocks.

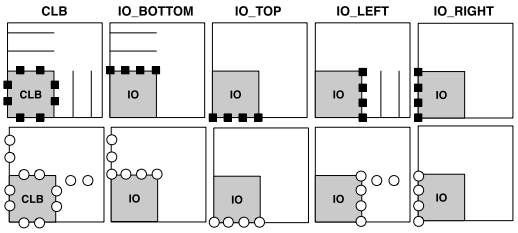

Fig. 2. The five tiles required to model an FPGA consisting of logic and IO blocks.

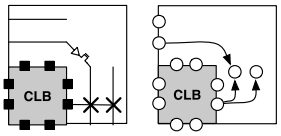

Fig. 3. Connections within the same tile instance.

Each tile type is modeled in detail using a graph similar to a RRG. Connections between neighbouring tiles are then represented by adding additional "wrap-around" edges. Special care is taken to model wire segments that span more than one tile. The resulting graph is significantly smaller than a full RRG, yet contains the exact same information. Most importantly, this new representation does not grow with the FPGA tile array size.

Sections 3.1 to 3.6 describe our technique in more detail. This technique requires some modifications to the routing algorithm; in Section 3.7, we show how our proposal can be incorporated into an FPGA router.

3.1 Tile Type Representation

We illustrate our technique on an FPGA without embedded memories or DSP blocks (extending this idea for these blocks is straight forward). For such an FPGA, we require five tile types: *CLB*, *IO_TOP*, *IO_BOTTOM*, *IO_LEFT*, *IO_RIGHT*. These five tiles are shown in the first row of Figure 2. The second row shows a representation of each tile, where each pin has been replaced by a graph node.

Programmable connections are modeled differently depending on whether they are between nodes in the same tile, or between nodes in different tiles. In the following three sections, we describe each of these separately.

3.2 Connections within the Same Tile Type

For programmable connections between resources that lie within the same tile instance, the edges are modeled in the traditional manner, as shown in Figure 3.

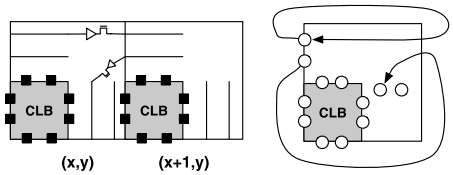

Fig. 4. Connections to different instances of the same tile type.

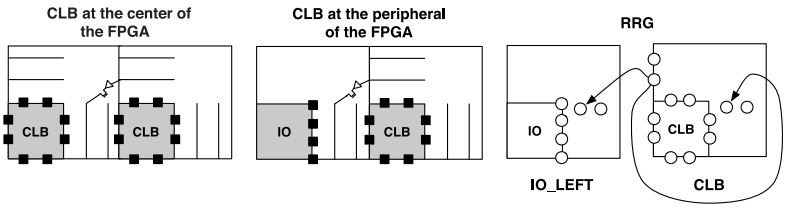

Fig. 5. Instance-dependant connections.

3.3 Connections between Different Tiles of the Same Type

When a programmable connection exists between routing resources that belong to different instances of the same tile type, we use an edge that *wraps around* as shown in Figure 4. The upper connection in this example is a connection between two horizontal wire segments. In our graph of the logic block tile, both of these wire segments are represented by the same node. When this happens the node emits an edge that returns to itself. The edge is labeled with two new quantities: *deltax* and *deltay*. These values specify to which adjacent tile instance the edge connects. In this example, the delta values for the edge are $(+1, 0)$. All edges are labeled in this way.

In the lower connection in Figure 4, a horizontal wire segment connects to a vertical wire segment. In this case, the two resources are represented by different nodes but are still part of the same tile type. By specifying the $(deltax, deltay)$ values correctly ($(-1, 0)$ in this example) the connectivity information is completely preserved.

3.4 Connections between Different Tile Types

Programmable connections between tiles belonging to different tile types can be handled in the same way. However, sometimes the type of an edge's destination tile depends on the coordinates of the tile in the FPGA array. For example, consider a logic block tile at the centre of an FPGA. The neighboring tiles in this case will be other logic block tiles. On the other hand, a logic block tile on the periphery of the FPGA would have some neighbors that are IO block tiles.

In this case, edges are added from the source node to all potential destination nodes in all potential tile types. This is shown graphically in Figure 5, in which two edges are sourced from the horizontal track in the right-most logic block tile. One edge wraps around to the same logic block tile, while the other

edge connects to the corresponding I/O block tile type. The two edges are given the same (*deltax*, *deltay*) labels ((−1, 0) in this case).

Clearly, in this situation, only one of the two edges corresponds to a physical routing switch for any given tile; which edge depends on the coordinates of the tile. It is up to the routing algorithm to keep track of the (x, y) tile location during maze routing, and only expand those edges that correspond to physical switches. More details regarding the router are in Section 3.7.

3.5 Modeling Long Wires

Wires that span more than one tile require special attention. The graph that we have described thus far implies the use of unit-length wire segments. Consider a horizontal wire segment that spans two tiles. We can think of this as two unit-length wire segments connected by a nonprogrammable connection. In our graph, we model these connections using a special type of edge called a *delayless edge*. Similar to the different types of programmable edges, the router will determine when to expand a delayless edge or a programmable edge.

3.6 Depopulated Connection and Switch Boxes

The discussion thus far has implicitly assumed that the connection blocks within each logic block tile are identical. However, as described in [Betz et al. 1999], this may not be the case; in a depopulated connection box architecture, some logic block tiles may have different values of F_c. Although this could be modeled by creating a separate tile type for each logic block with a unique connection block pattern, it is more memory efficient to represent all possible connection block connections and leave it to the router to determine at runtime which edges correspond to physical switches (as in Section 3.4) based on the (x, y) location that is currently being explored. Architectures with depopulated switch blocks (in which not every switch block contains connections to all tracks) can be handled in the same manner.

3.7 Changes to the Routing Algorithm

In general, the router has two new tasks:

(1) To keep track of the (x,y) tile coordinate of each routing resource that is visited during maze expansion.
(2) To determine which edges represent physical connections at the given (x,y) tile location and expand only these edges.

Figure 6 shows the pseudo-code for the Pathfinder-based VPR router. As long as the sink has not been found, the node with the best score is removed from the priority queue (PQ) and its neighboring nodes are added back into the PQ. Since the router always knows the (x, y) coordinates of the source and sinks being routed, the (x, y) coordinates of every subsequent node that is added to the PQ can be calculated using the (x, y) coordinate of the source node as the starting point, and the relative positioning information stored in the (*deltax*, *deltay*) edge labels.

```
Let: RT(i) be the set of nodes, n,
    in the current routing of net(i).

while (overused resources exist)
  for (each net i)
    rip-up routing tree RT(i)
    RT(i) = source(i)
    for(each sink j of net(i))
      PriorityQueue = RT(i)

      /* Wave expansion */
      while (sink(i,j) not found)
        m = dequeue(PriorityQueue);
        expand_neighbours(m)

      /* Backtrace */
      for(each node n in path to sink j)
        Add n to RT(i)

  Update historical cost for all n
```

Fig. 6. Pseudo-code for the VPR router.

Traditionally, when expanding the neighbors of a node, the router iterates across the node's list of fanout edges and adds each and every node to the PQ. As described in the previous sections, not all edges in our graph correspond to physical switches. A routine called $fanout\_exists()$ uses the (x, y) coordinates of the tile being explored along with the parameters describing the detailed-routing architecture to compute whether each edge corresponds to a physical switch. The details of these calculations have been omitted due to space but are straightforward. The router will add this node to the PQ only if the connection represents a physical connection. When the router encounters a delayless edge that corresponds to a physical connection on the FPGA, the router immediately performs neighbor expansion on the node pointed to by the delayless edge. In this sense, the use of delayless edges is merely a concept and does not affect the size of the priority queue. The modified *neighbor expansion* pseudo-code is shown in Figure 7.

In terms of overheads, the router changes for handling task 1 requires the addition of several new fields to each PQ data object. These fields are used to track the (x,y) location of the associated routing resource, and to properly handle traceback. This leads to a negligible memory overhead that is dependent on the PQ size. Handling task 2 leads to a runtime overhead and is quantified in Section 4.6.

4. EXPERIMENTAL RESULTS—ARCHITECTURAL DATA

In this section, we measure the memory footprint savings obtained by our proposal. We first describe our experimental framework in Section 4.1. In Sections 4.2 to 4.4, we measure the memory footprint savings as a function of the FPGA array size, the channel width, and the segment length. Finally, we quantify the proposed techniques impact on runtime in Section 4.5.

```
expand_neighbours( m ) {

 /* Expanding wire nodes */
 if(m is a wire)
   (x,y) = (x,y) of wire start tile
   while((x,y) != (x,y) of wire end tile)
     for (all fanout nodes n of node m)
       if(fanout_exists(m,n)
         enqueue(n)

     (x,y) += (deltax,deltay) of
                       delayless edge

 /* Expanding all other nodes */
 else
   for ( all fanout nodes n of node m )
     if(fanout_exists(m,n))
       enqueue(n)
```

Fig. 7. Pseudo-code highlighting changes to neighbor expansion.

4.1 Experimental Framework For N, W, and L

For the experiments in Section 4.2, 4.3, and 4.4, we pause the VPR process after it loads all of the architectural and temporary routing data structures, but before the routing algorithm actually starts.

After pausing, we measure the virtual memory usage of the VPR process. Virtual memory is the amount of memory that the process requires regardless of how much physical memory is actually installed on the machine. We take this measurement using the UNIX command top. Tops ability to give detailed memory usage information is sufficient for our purpose.

In the following experiments, we route the same circuit to a number of different architectures that vary in N (array size), W (channel width), and L (wire segment length). For each architecture, we assume that logic blocks have 22 inputs and consists of ten 4-input LUTs and registers.

4.2 Experimental Results for Array Size N

Figure 8 shows the memory requirements of our router and the original VPR router as a function of the array dimension N. Each FPGA consists of an array of NxN logic block tiles. For small array sizes (i.e., N = 25), the memory footprint of the proposed technique is only slightly better (1.7X smaller). As the array size increases, the difference in memory footprint becomes more dramatic. At N = 200, the memory footprint of the proposed technique is 6.5X smaller. Note that a 200x200 architecture with a channel width of W = 150 is comparable to current commercial devices such as those offered by Altera[1] and Xilinx.[2]

Although the storage required for the architectural data is no longer a function of N, the storage required to implement the temporary routing structures

[1]http://www.altera.com/literature/hb/stratix-iv/stratix4_handbook.pdf
[2]http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf

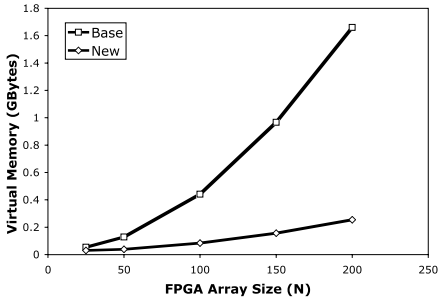

Fig. 8. VPR Memory footprint for various FPGA sizes. W = 150, L = 4.

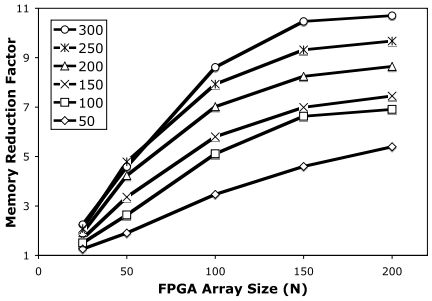

Fig. 9. Memory reductions for L=8 across various values of N and W.

is still proportional to N. This explains why our storage requirements still grow slightly as N increases.

Figure 9 shows how the memory savings depends on N for various fixed channel widths. The vertical axis is the *baseline memory footprint*/*new memory footprint* (this is the memory savings). Each line corresponds to a different channel width. The wire length is $L = 8$ (we also looked at $L = 1$, $L = 4$, and $L = 16$ and the conclusions were similar). As shown in the graph, as N increases, the memory savings increases but reaches an asymptote.

The asymptote can be explained as follows. At large values of N, the temporary routing data dominates since the architectural data requirements are now constant. Therefore, the asymptote is the ratio between the memory footprint of an architectural data object versus the memory footprint of a temporary routing data object. By using the case of $N = 200$, we can conservatively approximate the asymptote. Figure 10 shows this approximation where each line represents a different wire length. This figure shows that for small channel widths ($W = 50$), we get approximately a 3X–6X reduction in memory footprint. For larger channel widths, we can get approximately 5X–13X reduction depending on L.

4.3 Experimental Results for Channel Width W

Figure 11 shows the storage requirements as a function of channel width for a family of architectures with $N = 100$, and $L = 4$. The storage requirements of

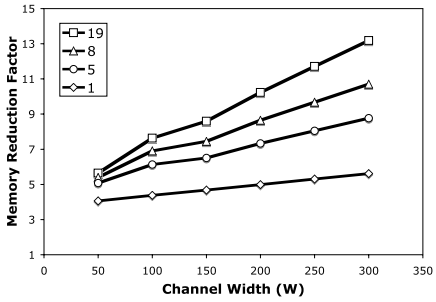

Fig. 10. Memory reductions for N=200 across various values of L and W.

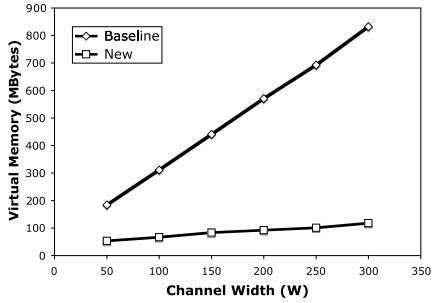

Fig. 11. Memory footprint of VPR for various channel widths. N = 100, L = 4.

both representations increase linearly but differ by a constant factor. This is due to the absence of the dependance on N in the new representation. As an example, suppose we increase W by 1 track. In the CLB tile of the proposed technique, the number of nodes is increased by two (one for the horizontal channel and one for the vertical channel). But using the original representation, the number of nodes for CLB channels is increased by $2 * N^2$ because there are N^2 CLB tiles.

4.4 Experimental Results for Wire Length L

Figure 12 shows that the memory reduction also changes with L. As L increases, less storage is required for wire resources due to the presence of fewer wire segments. This reduces both the connectivity and temporary information in the original representation. In the proposed technique, this only reduces the temporary information. Since the connectivity information in the proposed technique is fixed and relatively small, its overall footprint decreases much faster than the full RRG, leading to an overall increase in memory reduction.

4.5 Impact on Routing Time

The additional steps added to the router described in Section 3.7 will increase the router's execution time. In this section, we measure this increase.

Both versions of VPR were instrumented to count the number of executed clock ticks at three points in the flow: immediately before placement, after

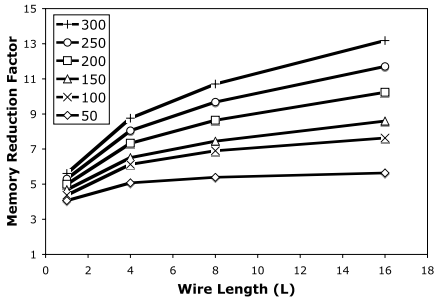

Fig. 12. Memory reductions for N=200 for various values of W and L.

Table I. Compile Time Results For Synthetic Benchmark Circuits in Seconds

Circuit	N	W	Place Time	Baseline Runtimes		New Runtimes		Ratio	
				Route	Total	Route	Total	Route	Total
30seq20com	43	72	321.07	100.16	421.23	224.88	545.95	2.25	1.30
3lrg50sml	35	56	171.14	44.48	215.62	103.19	274.33	2.32	1.27
50mixedsize	44	54	298.85	59.73	358.68	134.67	433.62	2.25	1.21
60mixedsize	43	62	312.91	92.29	405.20	208.52	521.43	2.26	1.29
70s1423	40	47	205.47	68.18	273.65	151.28	356.75	2.22	1.30
Lotsaffs2	33	59	131.49	29.97	161.46	69.95	201.44	2.33	1.25
70sml	33	99	198.79	89.04	287.83	194.88	393.67	2.19	1.37
Average								2.26	1.28

placement, and after routing. These three values allow us to calculate the placement time, routing time, and total compile time. All measurements were performed on an Intel Core2 Duo 1.87GHz processor. Only a single core was used.

We are interested in how the proposed technique performs across different benchmark circuits because the number of nets and sinks affect the amount of time spent on routing. We use two suites of benchmark circuits. The first suite consists of the twenty largest MCNC circuits. The second suite consist of seven synthetic circuits created by stitching together a number of MCNC circuits using the technique described in Lamoureux and Wilton [2007]. Although these circuits are much larger than the twenty largest MCNC circuits, they are still not as big as some of the architectures that we investigated in the previous subsections. However, we found the impact on compile-time to be generally independent of circuit size.

We assume an architecture that uses length 4 wires, and logic blocks with 22 inputs and ten 4-input LUTs. Each circuit was placed and routed to the minimum array size and channel width.

Table I reports the compile time results for the suite of synthetic circuits. Columns 2–3 show the minimum array size and channel width required to place and route the circuit. Columns 4–6 show the placement, routing, and overall run-times when using the original version of VPR. Columns 7–8 show the new routing and overall runtimes and Columns 9–10 show the ratios. The average increase in routing time is 2.26X and the average increase in overall compile time is 1.28X. The overhead for each circuit is close to the average,

meaning that the compile time overhead is generally independent of the FPGA size. This is because the overhead depends mainly on the extra time required to perform the *fanout_exists()* calculations.

The average increase in route time and overall compile time for the MCNC suite were found to be 2.16X and 1.29X respectively. The detailed results for this suite have been omitted due to space but yielded the same conclusions.

4.6 Impact on Routing Solution

An important property of this technique is that all of the the architectural information is maintained exactly. This means that there is no change to the detailed routing architecture as seen by the router. This was verified by building the full RRG from our modified graph, and then comparing it to the RRG built in the traditional manner.

5. TEMPORARY ROUTING DATA

After the memory requirement of the architectural data is reduced as described in the previous section, the storage for the temporary routing data becomes dominant. This storage consists of a Temporary Routing Data Object (TRDO) for each routing resource on the FPGA. Each TRDO contains various scoring parameters, traceback information, and other bookkeeping information such as whether the resource is occupied. Unlike the architectural data, the TRDO's are dynamic and do not need to be maintained throughout the entire routing process.

In this section, we first investigate the lifespan of these TRDOs. Using observations from this discussion, we show how simple memory allocation and deallocation schemes can be used to reduce the peak memory required to store these temporary structures. Finally, we present a design-adaptive algorithm that balances peak memory usage with router runtime.

The techniques described in this section are independent of, and cumulative with, the technique presented to reduce the architectural data. However, it is important to note that they do not have a significant impact unless the architectural data is no longer the dominant component.

5.1 Lifespan of TRDOs

In the VPR router, TRDOs are allocated for every routing resource and they persist throughout the routing process. We can reduce this memory requirement by taking advantage of three observations:

(1) We only need to allocate TRDO storage for routing resources that are explored during the maze-routing process. Unexplored resources are not scored and do not require any bookkeeping information.
(2) In practice, the number of resources explored is always less than the total number of resources on the chip [Betz et al. 1999]. For low- and

medium-density designs, the pins (and source/sink routing resource nodes) associated with unused logic blocks will never be explored. These circuits will also typically not use all tracks in each channel, further reducing the number of nodes that must be explored during routing. Even in high-density designs, the routing congestion often varies across the design [Tom and Lemieux 2005]. This leaves regions of low routing-congestion and thus a high amount of unused routing resources.

(3) The routing of different nets, or even the same net between different routing iterations, leads to the exploration of different routing resources. This means that the TRDO for a given routing resource may not have to persist throughout the entire routing process.

An upper and lower bound can be written for the number of nodes that will be explored. In the worst case, all routing resources are explored during the routing of a net. This should never happen in a negotiation based router unless there is an architectural flaw that makes a route impossible even when ignoring congestion. In the best case, the router only explores the resources used in the final routing solution. Although this never happens in practice, a directed-wavefront expansion during maze-routing brings the number of explored resources close to the best case.

5.2 Simple Memory Management Schemes

Using the observations in the previous subsection, we can reduce the peak temporary routing data memory requirements by intelligently allocating storage for a TRDO only when we know for certain that a routing resource will be explored. In addition, we can deallocate the space for a TRDO when we know that the data in a TRDO will no longer be used. An important task is to determine when and how often to perform this deallocation such that we do not incur a significant runtime overhead.

In the next section, we will evaluate three memory deallocation schemes. Each scheme allocates memory for a resource when it is first explored (in VPR, a node is deemed "explored" when it is removed from the priority queue). Memory deallocation is handled as follows:

Scheme 1. In this scheme, TRDOs are allowed to persist until the end of the routing process. In other words, no dynamic memory deallocation is employed.

Scheme 2. At the end of a routing iteration, the TRDO for a routing resource is deallocated if the routing resource is not occupied by a net and if its scoring parameters have not been changed from initial default values.

Scheme 3. In this scheme, memory deallocation occurs at the end of routing each net. This can lead to a significant runtime overhead for circuits with a large number of nets. However, this is also the most aggressive deallocation scheme possible and will lead to the largest memory reduction. To reduce the runtime overhead, it is possible to perform

deallocation after routing every x nets where $x > 1$. However, we do not investigate that in this article.

In Section 6 we will compare the reduction in the peak memory requirements and impact on runtime for each of these schemes.

5.3 Design-Adaptive Memory Management Algorithm

In this section, we describe a design-adaptive algorithm that employs a memory-aware net ordering strategy.

In VPR, nets are routed in an arbitrary order. The key idea in our design-adaptive scheme is to order the nets intelligently to minimize the amount of allocation and deallocation required. To accomplish this, we first divide the FPGA into (possibly overlapping) square *regions* of size N_{region}xN_{region} logic blocks, where N_{region} depends on the design (to be discussed below). Each net is then assigned to the region that shares the most overlap with that net's bounding box. During routing, the router iterates across each region and routes the nets belonging to that region before proceeding to the next region. Note that while routing a net, all routing resources (including those not in the region) are available to be explored. Intuitively, however, most routing will occur within the region. After routing the nets in a region, the router performs a deallocation pass.

Compared to Scheme 3 in Section 5.2, this new algorithm will reduce the number of unnecessary deallocations, improving the runtime. The exploration for nets within a region will often visit the same routing resources. By not deallocating the TRDOs for these routing resources until all nets assigned to a region have been routed, fewer unnecessary deallocations will be performed. The peak memory reduction capabilities, however, will be somewhat worse than Scheme 3, since a net will rarely be routed entirely within a region. In the next section, we experimentally evaluate the run-time and memory reduction capabilities of this algorithm.

An important consideration is how to choose the size, shape, and location of the regions. In our experiments, we fix $N_{region} = \frac{\overline{B}}{2}$ where $\overline{B}$ is the average half-perimeter of the bounding box of each net in the design (so a design that tends to have smaller net bounding boxes will have smaller regions). We also assume that regions are overlapped such that the start coordinate of a region is $\frac{N_{region}}{2}$ logic blocks to the right of the previous region and each row of regions is $\frac{N_{region}}{2}$ logic blocks below the previous row of regions.

Unlike all other techniques in this paper, reordering the nets may change the routing solution slightly. However, experimentally we determined that the impact on the minimum channel width was within plus or minus one track, and overall circuit delay was negligible.

6. EXPERIMENTAL RESULTS—TEMPORARY ROUTING DATA

In this section, we quantify the impact on memory and runtime of the algorithms described in Section 5.

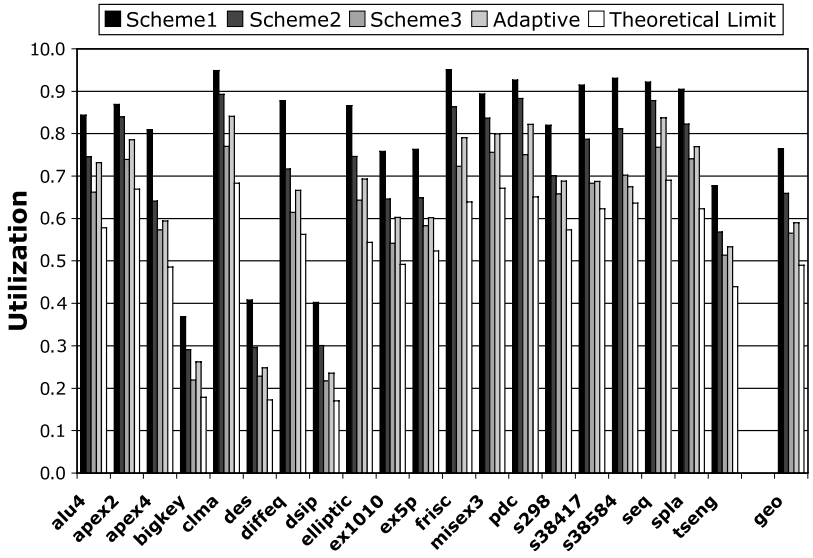

Fig. 13. TRDO peak memory reduction routing to minimum channel width FPGAs.

6.1 Impact on Memory

To experimentally measure the impact of our algorithms on the memory required to store the temporary routing data, we mapped the 20 largest MCNC circuits to a minimum-sized channel width FPGA. We assumed a clustered architecture in which each cluster contains ten 4-input LUT and register pairs and has 22 and 10 outputs. We also assumed each routing track spans four logic blocks, and that the Wilton switch box is used.

Rather than measuring the peak memory usage directly, we keep track of the maximum number of TRDOs allocated at any given time while routing to a minimum-channel width FPGA. Figure 13 shows this quantity normalized by the total number of routing resources available on the FPGA for each benchmark circuit (this is equal to the reduction in memory needed to store all TRDOs, compared to a router such as VPR that maintains a TRDO for every resource). For each circuit, we show the results for all three simple memory deallocation schemes, as well as the design-adaptive scheme. We also show the theoretical minimum that could be obtained; this quantity is the ratio of the number of routing resources used in the final routing solution divided by the total number of routing resources on the chip.

As the graph shows, the memory reductions are significant. On average, our algorithms reduce the peak memory needed to store TRDOs by between 24% and 43% of the original value, compared to a theoretical minimum of 51%.

We repeated the experiments assuming channel widths of $1.25W_{min}$, $1.5W_{min}$, and $2W_{min}$ where W_{min} is the minimum channel width required to route the circuit. These situations arise in industry when designers must choose from a family of FPGAs with predefined sizes. In many cases, the design will not utilize the full channel width of the FPGA. In academia, these

Table II. Average Temporary Routing Data Memory Reduction

Channel Width	Ratio				
	Scheme1	Scheme2	Scheme3	Design Adaptive	Theoretical Limit
$1.00W_{min}$	0.76	0.66	0.57	0.59	0.49
$1.25W_{min}$	0.75	0.61	0.51	0.53	0.44
$1.50W_{min}$	0.72	0.56	0.46	0.49	0.40
$2.00W_{min}$	0.68	0.50	0.39	0.41	0.34

Table III. Runtime Overhead of Each Scheme in Seconds

Design	Baseline	Scheme				Ratio			
		1	2	3	Design Adaptive	1	2	3	Design Adaptive
alu4	1.15	1.20	1.22	2.09	1.26	1.047	1.057	1.821	1.100
apex2	1.56	1.64	1.67	2.82	1.56	1.052	1.069	1.807	1.001
apex4	0.52	0.53	0.54	0.93	0.38	1.028	1.049	1.800	0.742
bigkey	1.38	1.43	1.46	2.94	2.19	1.036	1.054	2.126	1.588
clma	10.93	11.64	11.77	37.50	12.15	1.064	1.076	3.430	1.112
des	3.02	3.15	3.21	7.92	3.92	1.044	1.064	2.627	1.300
diffeq	0.42	0.44	0.45	1.03	0.46	1.046	1.070	2.436	1.084
dsip	1.87	1.94	1.99	4.75	1.90	1.039	1.067	2.543	1.018
elliptic	2.28	2.38	2.42	6.90	2.46	1.045	1.065	3.033	1.078
ex1010	0.22	0.23	0.23	0.39	0.20	1.038	1.060	1.798	0.919
ex5p	0.05	0.06	0.06	0.08	0.07	1.025	1.057	1.479	1.350
frisc	3.27	3.44	3.51	9.04	3.50	1.055	1.074	2.767	1.071
misex3	1.14	1.18	1.20	2.06	1.19	1.039	1.054	1.808	1.043
pdc	5.36	5.65	5.72	12.54	6.18	1.055	1.068	2.340	1.154
s298	0.47	0.49	0.50	0.84	0.46	1.045	1.058	1.805	0.990
s38417	2.59	2.73	2.78	16.34	3.43	1.055	1.074	6.303	1.326
s38584	3.11	3.28	3.34	21.55	3.79	1.056	1.076	6.939	1.222
seq	1.83	1.91	1.94	3.39	2.00	1.045	1.063	1.857	1.100
spla	3.47	3.66	3.71	8.07	3.52	1.054	1.070	2.323	1.013
tseng	0.41	0.42	0.43	1.01	0.47	1.038	1.066	2.499	1.153
geo						1.045	1.064	2.441	1.104

scenarios occur when performing binary-search routing to determine the minimum channel-width required by a design. These results are summarized in Table II. As the table shows, the reduction is more pronounced for architectures with wider channels, since a smaller proportion of the routing tracks need to be explored.

6.2 Impact on Runtime

Table III summarizes the route time results. Column 2 lists the routing time for each benchmark using an unmodified version of VPR. Columns 3–6 list the routing time using the three different memory deallocation schemes schemes as well as the design-adaptive method. Columns 7–10 show the ratio of each runtime to the runtime of the unmodified VPR. We verified that the change in route times are independent of the channel width, and hence only show results for the minimum channel width case.

There are three potential sources of runtime overhead. The first source is a potential decrease in cache performance due to memory fragmentation. The

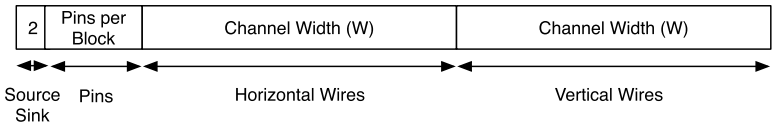

Fig. 14. TRDO memory organization for a single tile.

second source is the need to check whether a TRDO has been allocated each time a routing resource is removed from the priority queue. The third source is the time to perform the memory deallocation. In the following subsections, we consider each of these sources.

6.2.1 *Cache Performance Overhead.* First consider the overhead due to possible cache performance degradation. This quantity can be estimated as follows. Observe that in Scheme 1, the runtime overhead is due to a possible cache performance degradation as well as the code required to check if a TRDO has been allocated for a given resource. To see whether cache performance degrades, we create Scheme 1a by modifying Scheme 1 such that memory for every routing resource is allocated up front, as in the baseline VPR. However, we preserve the code that checks whether to dynamically allocate memory for a routing resource. This eliminates the potential cache performance degradation but retains the second source of runtime overhead. We found that there was virtually no change in runtime between these two schemes, and conclude that any change in cache performance is negligible.

This result is an artifact of how data is stored in the original VPR router. In this router, allocation of TRDOs proceeds from one tile to the next. Within a tile, TRDOs for each type of resource are allocated together, as shown in Figure 14. The number of TRDOs for each resource type is shown in each rectangle. During routing, a TRDO is accessed whenever its corresponding routing resource is removed from the priority queue for exploration. In most cases, the resource being explored is a neighbor of the previously explored node. However, neighboring resources are usually not within the same tile. Thus, there is little cache locality in the original router, and hence our algorithms do not degrade the cache effectiveness. This observation may be different in commercial FPGA routers that have been optimized for cache performance.

6.2.2 *Additional Code Overhead.* As concluded from Section 6.2.1, all of the overhead in Scheme 1 is from determining when to allocate memory. As shown in the table, this increases the runtime by 4.5%. In Scheme 2, there is an additional overhead of another 2%; this is the time required to perform memory deallocation once after each routing iteration. The circuits in this suite required 25-30 routing iterations to complete. The runtime overhead in Scheme 3 is very significant (144%). This is because we perform a memory deallocation pass after each net is routed. As expected, the runtime of the design-adaptive algorithm is more modest (10.4%).

7. CONCLUSIONS

In this article, we presented techniques for reducing the runtime memory footprint of FPGA routing algorithms. We focused on two types of storage: the static storage required to maintain architectural data and the dynamic storage required to maintain temporary routing information.

To reduce the memory footprint of the architectural data, we proposed a technique that takes advantage of the regularity in FPGAs. By doing so, we showed that the memory footprint of the routing step, one of the most memory-intensive steps in the entire CAD flow, can be reduced by an order of magnitude without loss of information. More importantly, the architectural data requirements are no longer dependent on the FPGA size. Due to the extra steps carried out by the router, we found that the average routing time increased by 126% and the overall place and route compile time increased on average by 28%.

To reduce the storage required for the temporary routing data, we took advantage of the lifespan of the temporary routing data objects. We proposed several schemes for dynamically allocating and deallocating this data. Combining this with an intelligent net ordering algorithm, we were able to reduce this component of the memory footprint by 41% while incurring a runtime penalty of 10.4%.

Although the techniques described in this paper traded off runtime for memory-footprint, it is important to remember that we only need to do this when we are up against the memory limits of the computing machine. When the memory requirements of the software exceeds the physical memory of the machine, a drastic increase in runtime will occur due to constant swapping of memory on and off of the hard-disk (thrashing). If we can prevent thrashing despite an increase in the nonthrashing runtime, then an overall improvement in perceived runtime may be achieved. To further this point, if the requirements exceed the amount that can be addressed by the machine (4GBytes for 32-bit operating systems), the route cannot be completed at all.

This paper has presented an important first step towards tackling the issue of computing requirement scalability of FPGA CAD tools. Code optimizations may delay the problem somewhat, but a long-term solution requires significant changes in CAD algorithms or FPGA architectures. Finally, we have made our modified version of VPR available.[3]

REFERENCES

ALTERA. 2008. Stratix iv device family overview. http://www.altera.com/products/devices/stratix-fpgas/stratix-iv/overview/stxiv-overview.html.

BETZ, V., ROSE, J., AND MARQUARDT, A. 1999. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers.

CARROLL, A. AND EBELING, C. 2006. Reducing the space complexity of pipelined routing using modified range encoding. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'06)*. 1–6.

[3]http://www.ece.ubc.ca/~scottc/downloads.html

CHAN, P., SCHLAG, M., EBELING, C., AND MCMURCHIE, L. Aug 2000. Distributed-memory parallel routing for field-programmable gate arrays. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst. 19*, 8, 850–862.

CHIN, S. AND WILTON, S. 2007. Memory footprint reduction for FPGA routing algorithms. In *Proceedings of the International Conference on Field-Programmable Technology*. 1–8.

DEHON, A., HUANG, R., AND WAWRZYNEK, J. 2002. Hardware-assisted fast routing. In *Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. 205–215.

HALDAR, M., NAYAK, A., CHOUDHARY, A., AND BANERJEE, P. 2000. Parallel algorithms for FPGA placement. In *Proceedings of the 10th Great Lakes Symposium on VLSI*. ACM, 86–94.

LAMOUREUX, J. AND WILTON, S. 27-29 Aug. 2007. Clock-aware placement for fpgas. In *Proceedings of the International Conference on Field Programmable Logic and Applications*. 124–131.

LI, S. AND EBELING, C. 2004. Quickroute: a fast routing algorithm for pipelined architectures. In *Proceedings of the IEEE International Conference on Field-Programmable Technology*. 73–80.

LYSECKY, R., VAHID, F., AND TAN, S. X.-D. 2004. Dynamic FPGA routing for just-in-time FPGA compilation. In *Proceedings of the 41st Annual Conference on Design Automation*. ACM, 954–959.

MULPURI, C. AND HAUCK, S. 2001. Runtime and quality tradeoffs in FPGA placement and routing. In *Proceedings of the ACM/SIGDA 9th International Symposium on Field Programmable Gate Arrays*. ACM, 29–36.

SANKAR, Y. AND ROSE, J. 1999. Trading quality for compile time: Ultra-fast placement for FPGAs. In *Proceedings of the ACM/SIGDA 7th International Symposium on Field Programmable Gate Arrays*. ACM, 157–166.

TOM, M. AND LEMIEUX, G. 2005. Logic block clustering of large designs for channel-width constrained FPGAs. In *Proceedings of the 42nd Design Automation Conference*. 726–731.

XILINX. 2007. Virtex 5 family overview.
http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf.

Received July 2008; accepted August 2008