A Study of the Scalability of On-Chip Routing for Just-in-Time FPGA Compilation

Roman Lysecky[a], Frank Vahid[a,*], Sheldon X.-D. Tan[b]
[a]Department of Computer Science and Engineering
[b]Department of Electrical Engineering
University of California, Riverside
{rlysecky, vahid}@cs.ucr.edu, stan@ee.ucr.edu
*Also with the Center for Embedded Computer Systems at UC Irvine

Abstract
Just-in-time (JIT) compilation has been used in many applications to enable standard software binaries to execute on different underlying processor architectures. We previously introduced the concept of a standard hardware binary, using a just-in-time compiler to compile the hardware binary to a field-programmable gate array (FPGA). Our JIT compiler includes lean versions of technology mapping, placement, and routing algorithms, of which routing is the most computationally and memory expensive step. As FPGAs continue to increase in size, a JIT FPGA compiler must be capable of efficiently mapping increasingly larger hardware circuits. In this paper, we analyze the scalability of our lean on-chip router, the Riverside On-Chip Router (ROCR), for routing increasingly large hardware circuits. We demonstrate that ROCR scales well in terms of execution time, memory usage and circuit quality, and we compare the scalability of ROCR to the well known Versatile Place and Route (VPR) timing-driven routing algorithm, comparing to both their standard routing algorithm and their fast routing algorithm. Our results show that on average ROCR executes 3 times faster using 18 times less memory than VPR. ROCR requires only 1% more routing resources, while creating a critical path 30% longer VPR's standard timing-driven router. Furthermore, for the largest hardware circuit, ROCR executes 3 times faster using 14 times less memory, and results in a critical path 2.6% shorter than VPR's fast timing-driven router.

Keywords
Place and route, just-in-time (JIT) compilation, hardware/software partitioning, FPGA, configurable logic, platforms, system-on-a-chip, dynamic optimization, codesign, warp processors, standard hardware binary.

1. Introduction
Just-in-time (JIT) compilation for FPGAs enables the development of a standard hardware binary as well as new technologies such as warp processing [19]. Standard hardware binaries would provide portability, allowing designers to use a single hardware netlist to configure a multitude of different FPGAs with different underlying architectures. An FPGA supporting JIT compilation would be capable of mapping the standard hardware binary to the FPGA's configurable logic while optimizing the hardware design for that FPGA.

In software design, just-in-time compilation provides powerful benefits. JIT compilation involves downloading a software binary format onto a chip, and then dynamically and transparently re-compiling that binary to the instruction set of the particular processor on that chip. The main benefit is that of binary portability – standard tools can be used to create a binary, and that same binary can be downloaded onto many different platforms. Modern x86 processors, including Intel's Pentium and Transmeta's Crusoe and Effecion processors, incorporate localized JIT compilation wherein x86 binaries are dynamically translated to and optimized for the chip's underlying RISC or VLIW instruction set [12][20].

Related to such a JIT compilation is dynamic transparent recompiling of a binary from one architecture to another, such as compiling x86 binaries to an Alpha architecture. Another form of JIT compilation involves distributing software as Java bytecode, which is then JIT compiled to a processor's native instruction set for improved performance compared to the execution on a Java Virtual Machine [13]. A related benefit of JIT compilation is that of dynamic optimization, wherein software hotspots are detected and dynamically recompiled for performance optimization [4][11].

As FPGAs continue to find their way alongside microprocessors into more end-products, such as TV set-top boxes, digital cameras, network routers, medical equipment, etc., the concept of a "binary" changes from that of a microprocessor program, to a more general concept of the configuration bits for a chip, possibly providing the configuration for an FPGA, a software program, or both. Ideally, a designer could create a standard binary for an FPGA and then map that standard binary to any of multiple FPGA architectures. Unfortunately, there presently does not exist the concept of a "standard" binary for FPGAs. Netlist formats are specific to a particular FPGA architecture, and FPGA architectures vary significantly.

Consider the example of a TV set-top box. Cable TV companies often transparently upgrade software within such boxes, by downloading new binaries. This works even though newer boxes may contain more advanced versions of the microprocessor, since newer processors often support older binaries or the binary can be JIT compiled to the different processor. Yet, such boxes increasingly rely on FPGAs for video processing, and so ideally we could download new binaries for the FPGAs as well, either to add new features or to fix bugs. However, newer boxes may contain newer or different FPGA architectures. Incorporating a JIT compiler within the FPGAs, a standard hardware binary could be transmitted and JIT compiled to the different FPGAs.

JIT compilation for FPGAs is also useful, in fact essential, for warp processors that perform dynamic hardware/software partitioning. Warp processors dynamically optimize an executing binary by moving software kernels to on-chip configurable logic, resulting in better performance and lower energy consumption [16][19]. At the heart of warp processors, a JIT compiler is

required to implement the synthesized hardware circuits onto the on-chip configurable logic fabric.

In designing warp processors and developing JIT FPGA compilation tools for standard hardware binaries, we previously developed a configurable logic fabric specifically designed to facilitate the development of a JIT FPGA compiler [15]. Furthermore, we developed a JIT FPGA compiler that performs technology mapping, placement, and routing [17][19].

Rapid increases in IC transistor capacities are enabling the design and use of increasingly larger FPGAs. A JIT compilation tool must be able to scale well to these larger FPGAs and larger hardware circuits. Furthermore, within a JIT compiler, as well as desktop-based FPGA CAD tools, routing is the most computationally intensive task, requiring larger memory resources and longer execution times than both technology mapping and placement algorithms. Hence, ensuring the scalability of the routing algorithm used within the JIT FPGA compilation tool is our first priority. In this paper, we present a study of the scalability of routing within a JIT FPGA compilation framework, comparing the scalability of our Riverside On-Chip Router (ROCR) algorithm with the Versatile Place and Route's (VPR) timing-driven router. We compare the scalability of our on-chip router and VPR in terms of execution time, memory usage, and circuit quality.

2. Just-in-Time FPGA Compilation

2.1 Configurable Logic Fabric

While many configurable logic architectures are currently available, traditional FPGAs are not well suited for JIT compilation. Traditional FPGAs are typically designed to handle an extremely wide variety of designs and are frequently used to prototype ASIC circuits. To support these vastly different designs, FPGA vendors, such as Xilinx [21] and Altera [1], design FPGAs with complex configurable logic blocks (CLBs), possibly containing varying sizes and number of lookup tables (LUTs), embedded memory cells, large routing resources, large input/output resources, etc. Traditional FPGA architectures are beneficial in terms of creating fast and compact designs, but such complexity requires complex technology mapping and complex place and route tools, which are not targeted for very fast or lean execution.

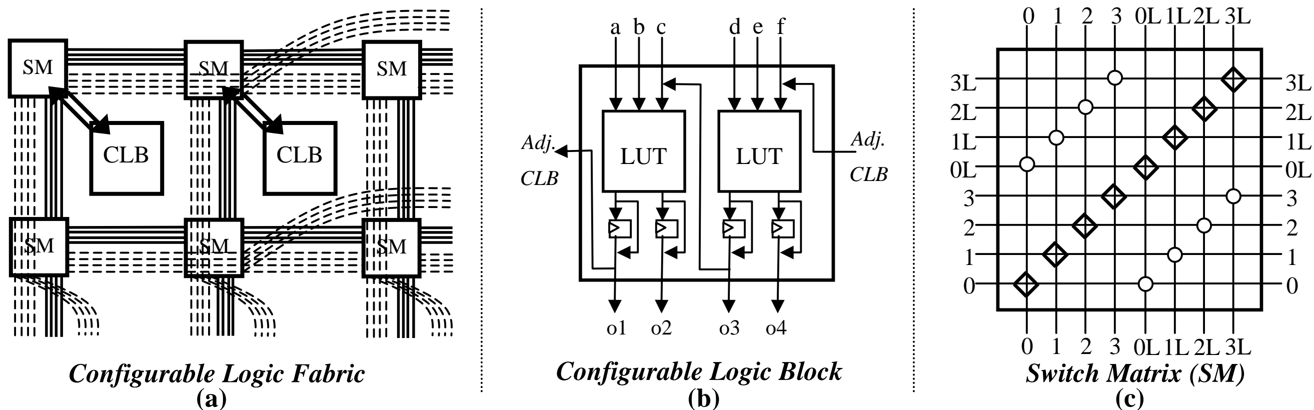
While most existing FPGAs are not designed with the goal of enabling extremely fast CAD tools, the Programmable Logic and Switch Matrix (Plasma) architecture was specifically designed to allow automatic routing of the entire configurable logic in three seconds [2]. To achieve such fast routing, the Plasma configurable logic architecture was designed with extremely large hierarchical routing resources. The plentiful routing resources enabled fast CAD tools for routing a circuit. However, the Plasma architecture requires a very large silicon area, which limits the applications in which using the Plasma architecture is feasible. Additionally, the routing tools were designed for fast execution time, but likely still require very large memory usage to achieve such fast routing, as is the case with existing FPGA routing algorithms.

We previously developed a simple configurable logic fabric (SCLF) specifically designed to enable the development of a lean JIT compiler for FPGA. Figure 1(a) shows a version of our SCLF, extended from that in [15] to support sequential logic by incorporating sequential elements within the CLBs. Our SCLF consists of an array of configurable logic blocks (CLBs) surrounded by switch matrices (SM) for routing between CLBs. Each CLB is directly connected to a single switch matrix to which all inputs and outputs of the CLB can be connected. Our SCLF handles routing between CLBs using the switch matrices, which can route signals in one of four directions to an adjacent SM *(represented as solid lines in the figure)* or to a SM two rows apart vertically or two columns apart horizontally *(represented as dashed lines)*.

Figure 1(b) shows our configurable logic block architecture. Each CLB consists of two 3-input 2-output LUTs and four flip-flops optionally connected to each of the four outputs. Choosing the proper size for the CLBs is important, as the size of the CLB directly impacts area resources and delays within our configurable logic fabric [18]. Our CLB design provides a reasonable trade-off between area and delay while allowing us to simplify our technology mapping and placement algorithms.

Finally, Figure 1(c) shows our switch matrix architecture. Each switch matrix is connected using *short* channels for routing between adjacent switch matrices and *long* channels for routing between every other switch matrix. Routing through the switch matrix can only connect a wire from one side with a given channel to another wire on the same channel but a different side of the switch matrix. Additionally, each *short* channel is paired with a *long* channel and can be connected together within the switch matrix *(indicated as a circle where two channels intersect)* allowing nets to be routed using *short* and *long* connections. Designing the switch matrix in this manner simplifies the routing algorithm of our JIT compiler by restricting the routing of each net to a single pair of channels throughout the configurable logic fabric.

Figure 1: (a) Simple configurable logic fabric (b) configurable logic block (CLB), and (c) switch matrix (SM) architecture.



2.2 Just-in-Time FPGA Compiler

While we found that developing our own configurable logic architecture helped to develop JIT compilation for FPGAs, implementing the required lean CAD tools for on-chip execution is not trivial. Existing FPGA CAD tools are capable of producing highly optimized hardware circuits. However, these tools suffer from very large data memory usage and long execution times. We designed our JIT compiler by focusing on developing lean algorithms that use as little data memory as possible and have fast execution times. These design goals will inherently restrict the ability of our JIT compiler to produce designs as highly optimized as their desktop counterparts will. However, our on-chip CAD tools create hardware circuits of *acceptable* quality.

Our existing JIT compiler for FPGAs consists of lean versions of technology mapping [15], placement [19], and routing algorithms [17]. Starting with the standard hardware binary, our JIT compiler performs technology mapping to map the hardware onto the LUTs within the configurable logic and further packs the LUTs into CLBs using a hierarchical, bottom-up graph clustering algorithm. Once mapped, we determine the location of each CLB within the configurable logic using our dependency-based positional placement algorithm. The placement algorithm attempts to assign location to the CLBs to reduce the critical path of the circuit while ensuring the circuit can be routed. Finally, we perform routing, using ROCR, in which the actual wire segments used to connect CLBs together is determined.

2.3 Scalability Requirements

Scalability is often a concern when developing any algorithm. However, for JIT FPGA compilation, scalability is also a necessity to ensure circuits of increasing size can be mapped to FPGAs, also of increasing size. In desktop-based FPGA CAD tools, routing can require tens to hundreds of megabytes of memory and require execution times ranging from minutes to hours. An on-chip JIT compiler must be able to execute much faster while using a very small memory footprint.

We previously demonstrated the feasibility of our lean on-chip routing algorithm, ROCR, which requires on average only 10 seconds and less than 4 megabytes of memory to route several benchmark circuits. However, our lean on-chip routing algorithm must be able to scale well to larger FPGAs as well as be able to route larger circuits quickly. For traditional FPGA routers, execution time is the primary concern when evaluating how well a specific router scales. For JIT FPGA compilation, we are also interested in evaluating how well memory usage and circuit quality scale.

3. Riverside On-Chip Router (ROCR)

Most FPGA routing algorithms rely on constructing a routing resource graph to represent the available connections between wires and CLBs within the FPGA architecture. An FPGA router must then find a path within the graph to connect the source and sinks of each net. During this routing process, a good FPGA router will attempt to route each net using the shortest path possible while also ensuring all nets can be routed. Most FPGA routing algorithms rely upon a maze routing algorithm [14]. Such routing algorithms also rely upon multiple routing iterations, in which the router rips-up some or all of the routes either to eliminate overuse of routing resources or to optimize the circuit speed.

The popular Pathfinder routing algorithm [10] further introduced the idea of negotiated congestion routing. During each routing iteration, Pathfinder routes each net using the best path possible allowing overuse of the routing resources. At the end of each routing iteration, the costs of the routing resources are adjusted relative to the amount of overuse in the previous routing iteration and all routes are ripped and rerouted in the next iteration. Pathfinder's negotiated congestion algorithm produces good hardware circuits by routing nets along the critical path using the shortest path possible while routing non-critical nets away from the routing congestion.

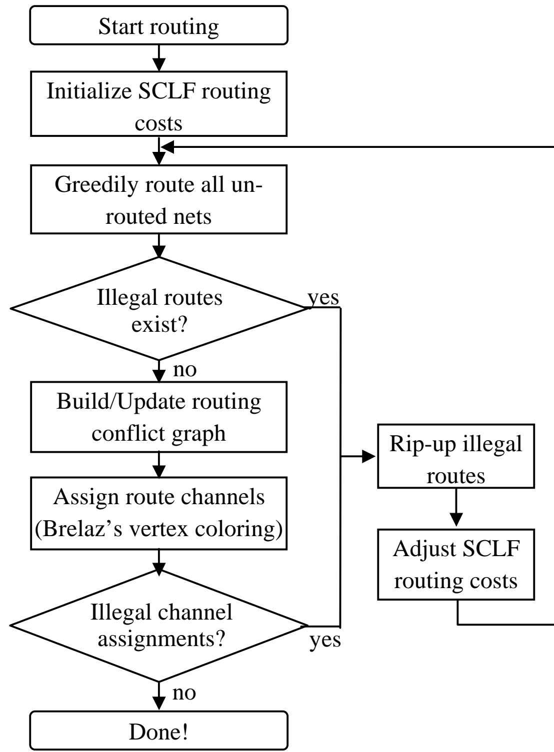
VPR's timing-driven router relies on a modified version of the Pathfinder algorithm to decrease routing execution times [5]. VPR's timing-driven routing algorithm also uses an Elmore delay model for optimizing the circuit speed instead of the linear delay model of the Pathfinder algorithm to improve circuit speed. However, VPR's reliance on constructing a routing resource graph, requiring tens to hundreds of megabytes of memory, makes those algorithms difficult to use for JIT compilation.

Therefore, we previously developed the Riverside On-Chip Router (ROCR), specifically designed for lean on-chip execution in JIT compilation for FPGAs. ROCR utilizes the general approach of VPR's routability-driven router allowing overuse of routing resources and illegal routes, and eliminates illegal routing through repeated routing iterations. ROCR also uses the basic routing cost model of VPR. However, unlike VPR, ROCR routes a hardware netlist using a much smaller routing resource graph and therefore much less memory usage. We designed our simple configurable logic fabric to allow us to represent routing between CLBs as routing between the switch matrices to which the CLBs are connected. Subsequently, our SCLF allows our routing algorithm to represent the routing resources using a very small routing resource graph. Our routing resource graph is a directed graph where the nodes of the graph correspond to switch matrices and the edges of the graph correspond to the routing resources between switch matrices. Our resource graph incorporates two types of edges in order to distinguish between the short and long routing wires. Furthermore, each edge of our routing resource graph is also associated with the routing costs used during the routing process

Figure 2 presents ROCR's overall routing algorithm. ROCR starts by initializing the routing costs within our routing resource graph. For all un-routed nets, ROCR uses a greedy routing approach to route the net. During the greedy routing process, for each sink within the net, we determine a route between the un-routed sink and the net's source or the nearest routed sink. At each step, we restrict the router to only choosing paths within a bounding box of the current sink and the chosen location to which we are routing. After all nets are routed, if illegal routes exist – the result of overusing routing channels – then ROCR rips-up only the illegal routes and adjusts the routing costs of the entire routing resource graph. While we use the same routing cost model of VPR's routability-driven router, ROCR also incorporates an adjustment cost. During the process of ripping-up illegal routes, we add a small routing adjustment cost to all routing resources used by an illegal route. During the routing process, an early routing decision can force our routing algorithm to choose a congested path. Hence, the routing adjustment cost discourages our greedy routing algorithm from selecting the same initial routing and enables our algorithm to attempt a different routing path in subsequent routing iterations.

Once we determine a valid global routing, ROCR performs detailed routing in which we assign the channels used for each

Figure 2: Riverside On-Chip Router (ROCR) algorithm overview.



route. The detailed routing starts by constructing a routing conflict graph. Two routes conflict when both routes pass through a given switch matrix and assigning the same channel for both routes would result in an illegal routing within the switch matrix. ROCR assigns the routing channels by determining a vertex coloring of the routing conflict graph. While many approaches for vertex coloring exists, ROCR uses Brelaz's vertex coloring algorithm [8]. Brelaz's algorithm is a simple greedy algorithm that produces good results while not increasing ROCR's overall memory consumption. If we are unable to assign a legal channel assignment for all routes, for those routes that we cannot find a valid channel assignment, ROCR rips-up the illegal routes, adjusts the routing costs of all nodes along the illegal route (as described before), and reroutes the illegal routes. ROCR finishes routing a circuit when a valid routing path and channel assignment has been determined for every net.

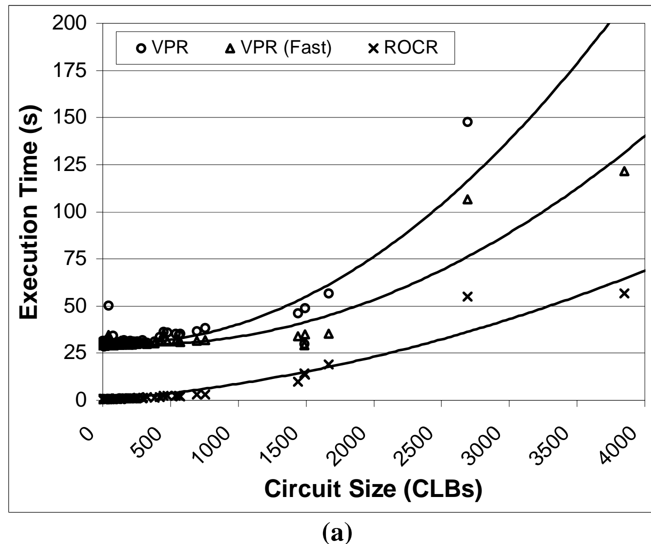
4. Experiments

We evaluate the scalability of our lean router, ROCR, comparing the execution time, memory requirements, and circuit quality of ROCR with VPR's timing-driven routing algorithm and VPR's timing-driven routing algorithm with the fast option enabled for 123 MCNC benchmark circuits [22], specified using the Berkeley Logic Interchange Format (BLIF). The circuits range in complexity from small circuits with only a few LUTs to large circuit with tens of thousands of LUTs. Similarly, the number of nets within the circuits ranges from small circuits with only a few nets to the largest circuit with more than 10,000 nets.

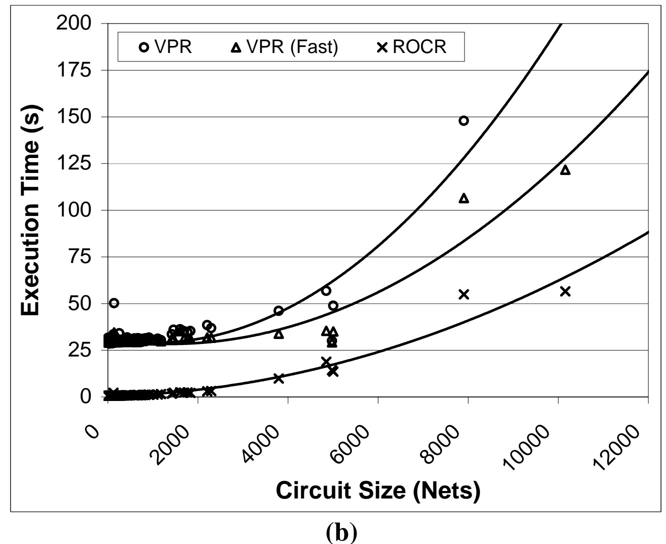
We considered a large configurable logic fabric consisting of a 100x100 array of CLBs. By considering such a large FPGA, we can evaluate the scalability of routing algorithms with respect to the circuit size, rather than the availability of CLBs or routing resources within the configurable logic. Starting with the BLIF specification for the benchmarks circuits, we first mapped each circuit to 3-input 1-ouput LUTs using FlowMap [9]. We then packed the LUTs together into the 3-input 2-output LUTs and further into CLBs using VPR's T-VPack [6][7]. We then determined the placement for each circuit using the VPR's bounding box placement algorithm [5][7]. Next, we routed the circuits using VPR's routability-driven router to determine the minimum number of routing channels required to successfully route all of the benchmark circuits, determining that our configurable logic fabric needs a routing channel width of 34. Finally, we routed the benchmark circuits using VPR's timing-driven router and ROCR.

Figure 3(a) and Figure 3(b) present the execution time in seconds of VPR's standard timing-driven router *(VPR)*, VPR's fast timing-driven router *(VPR Fast)*, and ROCR for all 123

Figure 3: Execution time *(seconds)* for VPR's timing-driven routing algorithm, VPR's fast timing-driven (Fast) routing algorithm, and ROCR, for MCNC benchmark circuits plotted against circuit size in terms of: (a) number of CLBs and (b) number of nets.



(a)



(b)

Figure 4: Minimum, maximum, and average data memory usage (kilobytes) for VPR's timing-driven routing algorithm, VPR's fast timing-driven (Fast) routing algorithm, and ROCR, for MCNC benchmark circuits.

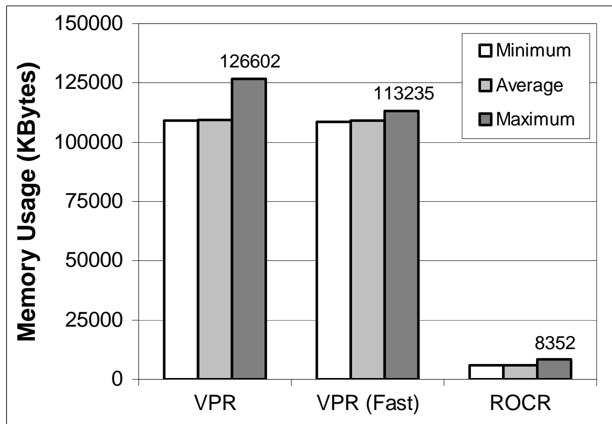
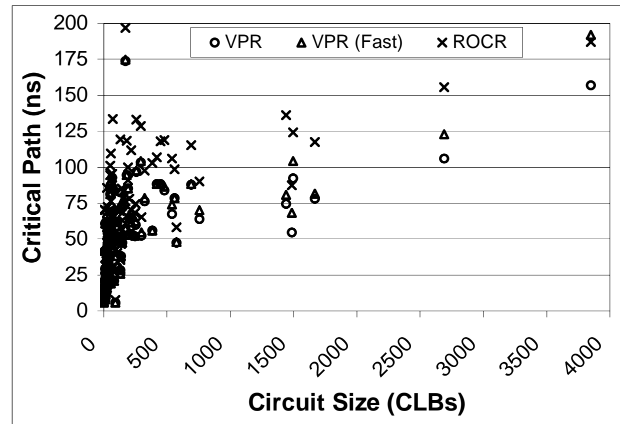


Figure 5: Critical path (nanoseconds) for MCNC benchmark circuits using VPR's timing-driven routing algorithm, VPR's fast timing-driven (Fast) routing algorithm, and ROCR, plotted against circuit size in terms of number of CLBs.

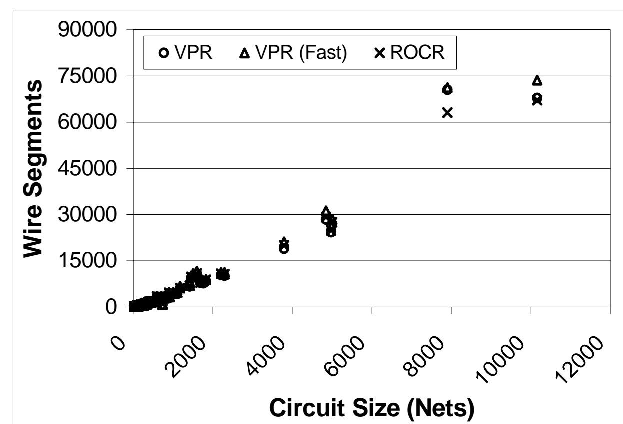


MCNC benchmark circuits plotted against circuit size in terms of number of CLBs and number of nets respectively. Additionally, the figures include trend lines for all three routing algorithms to demonstrate the projected behavior when larger circuits are considered. For a fair comparison, we obtained all results using a 1.6 GHz Pentium workstation. On average, ROCR requires only 2.2 seconds to route the benchmarks compared with an average execution time of 33.5 seconds for VPR's standard router and 31.6 seconds for VPR's fast router. When considering extremely small circuits with less than 100 CLBs and less than 300 nets, ROCR is over 40X faster than VPR. Such small designs typically do not require multiple routing iterations and ROCR's simplified approach provides a very large advantage in terms of execution time. For larger circuits with more than 1000 CLBs and 3000 nets that require multiple routing iterations, ROCR is on average 3X faster than VPR's standard routing algorithm and 2X faster than VPR's fast routing algorithm. Furthermore, as demonstrated by the projected trends lines for larger circuits, ROCR scales better than both VPR algorithms.

Figure 4 presents the minimum, average, and maximum memory usage for VPR's standard timing-driven router *(VPR)*, VPR's fast timing-driven router *(VPR Fast)*, and ROCR across all 123 benchmark circuits. ROCR requires a maximum of roughly 8 megabytes of memory while VPR's standard and fast timing-driven routing algorithms required a maximum of over 110 MB. On average, ROCR's simplified routing architecture and corresponding resource graph, allows ROCR to route the circuits using 18X less memory than VPR. For the smallest circuits, the memory usage of both routers is primarily the result of the routing resource graph used to represent the configurable logic fabric. The increase in memory usage for larger circuits is directly related to the memory required to represent a hardware circuit's CLBs and nets and any internal data structure used during routing. Compared with VPR, ROCR uses over 7X less memory to represent the hardware circuit.

ROCR's circuit quality also scales well as the circuit size increases, both in terms of circuit speed and in terms of the amount of routing resources used to route the circuit. For a JIT FPGA compiler, the circuit quality should remain reasonable even when considering large circuits. In other words, our lean routing should still be able to route the circuit reasonably close to desktop-based CAD tools regardless of circuit size. Therefore, we measure the scalability of circuit quality of ROCR by comparing the resulting circuits' critical path and number of used wire segments to that of VPR's standard and fast timing-driven routers. Figure 5 presents the critical path in nanoseconds of the hardware circuits produced by VPR's standard timing-driven router *(VPR)*, VPR's fast timing-driven router *(VPR Fast)*, and ROCR, plotted against the circuit size in terms of number of CLBs. ROCR produces circuits with a critical path on average 30% longer than VPR's standard timing-driven router and 27% longer than VPR's fast timing-driven router. However, for the largest circuits consisting of more than 3800 CLBs, the critical path produced by ROCR is only 19% longer than VPR's standard router, and is actually 2.6% shorter than VPR's fast router. Figure 6 presents the total wire segments used to route the hardware circuits using VPR's standard timing-driven router *(VPR)*, VPR's fast timing-driven router *(VPR Fast)*, and ROCR plotted against circuit size in terms of number of nets. On average, ROCR requires only 1% more wire segments to route the benchmark circuits compared to

Figure 6: Total wire segment required to route MCNC benchmark circuits using VPR's timing-driven routing algorithm, VPR's fast timing-driven (Fast) routing algorithm, and ROCR plotted against circuit size in terms of number of nets.



VPR's standard router and 5% fewer wire segments than VPR's fast router. However, for larger circuits with more than 3000 nets, ROCR actually requires on average 2% and 8% fewer wire segments to route the circuits than VPR's standard router and fast router, respectively.

5. Conclusions

JIT compilation for FPGAs enables the development of a standard binary for FPGAs and facilitates the portability of binaries across FPGA architectures. As FPGAs continue to increase in size and designers need to implement increasingly large circuits on these FPGAs, a JIT FPGA compiler's execution time, memory usage, and resulting circuit quality must scale well. We demonstrated that our ROCR algorithm for on-chip routing, the most computationally intensive component of JIT compilation, scales very favorably as circuit size increases. On average, ROCR routes the circuits 3X faster than VPR's standard timing-driven router and 2X faster than VPR's fast timing-driven router, using on average 18X less memory, and resulting in hardware circuits using fewer routing resources, and with a critical path only 30% longer than VPR's standard router and 27% longer than VPR's fast router. Furthermore, for the largest hardware circuit, ROCR executes 2X faster using 14X less memory, and results in a critical path that is actually 2.6% shorter than VPR's fast router.

Future work includes analyzing how well ROCR performs when considering hardware circuits of increasing size approaching the capacity of the FPGA. Future work also includes improving ROCR to increase performance as well incorporating timing information to improve circuit speed. We are currently working on improving technology mapping and placement used within our JIT FPGA compiler. Our current JIT compiler includes greedy technology mapping and placement algorithms that do not scale well to extremely large hardware circuits. Thus, we are developing lean versions of technology mapping and placement algorithms that scale well to large hardware circuits.

6. Acknowledgements

This research was supported in part by the National Science Foundation (CCR-0203829), the Semiconductor Research Corporation (2003-HJ-1046G), and Xilinx Corp.

7. References

[1] Altera Corp. http://www.altera.com, 2005.

[2] Amerson, R., R. Carter, W. Culbertson, P. Kuekes, G. Snider, L. Albertson. Plasma: An FPGA for Million Gate Systems, Symp. on Field Programmable Gate Arrays, 1996.

[3] Atmel Corp. http://www.atmel.com, 2005.

[4] Bala, V., E. Duesterwald, S. Banerjia. Dynamo: A Transparent Dynamic Optimization System. Conference on Programming Language Design and Implementation (PLDI), 2000.

[5] Betz, V., J. Rose, A. Marquardt. Architecture and CAD for Deep-Submicron FPGAs. Kluwer Academic Publishers, 1999.

[6] Betz, V., J. Rose. VPR: A New Packing, Placement, and Routing for FPGA Research. International Workshop on Field Programmable Logic and Applications (FPLA), 1997.

[7] Betz, V., J. Rose, A. Marquardt. VPR and T-VPack: Versatile Packing, Placement and Routing for FPGAs. http://www.eecg.toronto.edu/~vaughn/vpr/vpr.html, 2003.

[8] Brelaz, D. New Methods to Color the Vertices of a Graph. Communication of the ACM 22, 251-256, 1979.

[9] Cong, J., Y. Ding. FlowMap: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table based FPGA Designs. IEEE Transactions On Computer-Aided Design of Integrated Circuits and Systems (TCAD), Vol. 13, No. 1, pp. 1-12, 1994.

[10] Ebeling, C., L. McMurchie, S. A. Hauck, S. Burns. Placement and Routing Tools for Triptych FPGA. IEEE Transactions on Very Large Scale Integration (TVLSI), Dec. 1995, pp. 473-482.

[11] Gschwind, M., E. Altman, S. Sathaye, P. Ledak, D. Appenzeller. Dynamic and Transparent Binary Translation. IEEE Computer, Vol. 3, pp.70-77, 2000.

[12] Klaiber, A. The Technology Behind Crusoe Processors. Transmeta Corporation White Paper, 2000.

[13] Krall, A. Efficient Java VM Just-In-Time Compilation, in Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, pp. 54-61, 1998.

[14] Lee, C. Y. An Algorithm for Path Connection and its Applications. IRE Transaction on Electronic Computing, Vol. EC=10, pp. 346-365, 1961.

[15] Lysecky, R., F. Vahid. A Configurable Logic Architecture for Dynamic Hardware/Software Partitioning. Proceedings of the Design Automation and Test in Europe Conference (DATE), 2004.

[16] Lysecky, R., F. Vahid. A Study of the Speedups and Competitiveness of FPGA Soft Processor Cores using Dynamic Hardware/Software Partitioning. Proceedings of the Design Automation and Test in Europe Conference (DATE), 2005.

[17] Lysecky, R., F. Vahid, S. X.-D. Tan. Dynamic FPGA Routing for Just-in-Time FPGA Compilation. Proceedings of the 41st Design Automation Conference (DAC), 2004.

[18] Singh, S., J. Rose, P. Chow, D. Lewis. The Effect of Logic Block Architecture on FPGA Performance. IEEE Journal of Solid-State Circuits, Vol. 27, No. 3, 1992.

[19] Stitt, G., R. Lysecky, F. Vahid. Dynamic Hardware/Software Partitioning: A First Approach. Proceedings of the 40th Design Automation Conference (DAC), 2003.

[20] Transmeta Corporation. http://www.transmeta.com, 2005.

[21] Xilinx, Inc. http://www.xilinx.com, 2005.

[22] Yang, S. Logic Synthesis and Optimization Benchmarks, Version 3.0. Technical Report, Microelectronics Center of North Carolina, 1991.