Dynamic FPGA routing for just-in-time FPGA compilation — Source link

R. Lyseckya, Frank Vahid, Sheldon X.-D. Tan

Institutions: University of California, Riverside

Published on: 07 Jun 2004 - Design Automation Conference

Topics: Place and route, Just-in-time compilation, Routing (electronic design automation), Field-programmable gate array and System on a chip

Dynamic FPGA 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 previously been used in many applications to enable standard software binaries to execute on different underlying processor architectures. However, embedded systems increasingly incorporate Field Programmable Gate Arrays (FPGAs), for which the concept of a standard hardware binary did not previously exist, requiring designers to implement a hardware circuit for a single specific FPGA. We introduce the concept of a standard hardware binary, using a just-in-time compiler to compile the hardware binary to an FPGA. A JIT compiler for FPGAs requires the development of lean versions of technology mapping, placement, and routing algorithms, of which routing is the most computationally and memory expensive step. We present the Riverside On-Chip Router (ROCR) designed to efficiently route a hardware circuit for a simple configurable logic fabric that we have developed. Through experiments with MCNC benchmark hardware circuits, we show that ROCR works well for JIT FPGA compilation, producing good hardware circuits using an order of magnitude less memory resources and execution time compared with the well known Versatile Place and Route (VPR) tool suite. ROCR produces good hardware circuits using 13X less memory and executing 10X faster than VPR's fastest routing algorithm. Furthermore, our results show ROCR requires only 10% additional routing resources, and results in circuit speeds only 32% slower than VPR's timing-driven router, and speeds that are actually 10% faster than VPR's routability-driven router.

Categories and Subject Descriptors

B.7.2 **[Integrated Circuits]**: Design Aids – Placement and Routing.
C.3 **[Special-Purpose and Application-Based Systems]**: Real-time and embedded systems

General Terms

Algorithms, Performance, Design.

Keywords

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

1. INTRODUCTION

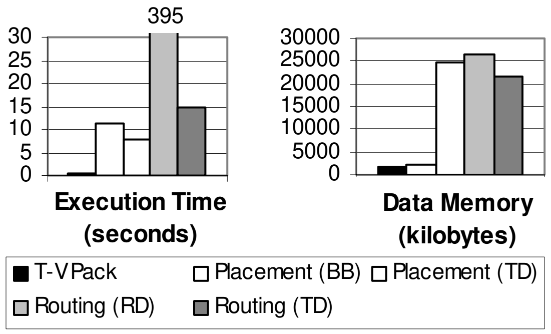
Just-in-time (JIT) compilation, now commonplace, provides powerful benefits for platforms executing software. We seek to extend those benefits to platforms that include configurable logic. Just-in-time compilation involves downloading a general software binary format onto a chip, and then dynamically 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. A form of JIT compilation is seen in most modern x86 processors, wherein x86 binaries are dynamically translated to and optimized for the chip's underlying RISC or VLIW instruction set [12]. Another popular example of dynamic binary translation is that of Transmeta's Crusoe and Efficeon processors [22]. 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 into more end-products, such as TV set-top boxes, cell phones, video game consoles, medical equipment, security screening equipment, image-processing video cameras, 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. Those configuration bits might describe an FPGA netlist, a microprocessor program, or one or more of both.

Ideally, a designer could create a standard binary for an FPGA, and then map that standard binary to any of multiple FPGA architectures. 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 processor generations still support older binaries, and works even when newer boxes contain different processors, since the binary may 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.

Unfortunately, there presently does not exist the concept of a "standard" binary for FPGAs. Netlist formats are specific to a

Figure 1: Comparison of execution *(seconds)* and data memory usage *(kilobytes)* of technology mapping *(T-VPack)* and VPR's placement and routing tools.



particular FPGA architecture, and FPGA architectures vary significantly, with several new architectures appearing every year. Thus, upgrading FPGAs presently requires that a netlist specific to a particular FPGA be created and then downloaded to the FPGA, making upgrades to FPGAs cumbersome.

We are therefore developing a JIT compiler for FPGAs, in order to develop the concept of a standard binary for FPGAs and portability of binaries across FPGA architectures. A JIT compiler for FPGAs would take a netlist in a standard netlist binary format, and execute technology mapping, placement, and routing. Such compilation would be done on a small lean processor coexisting on the FPGA for this purpose, taking less than 5% of the chip area. We point out that JIT compilation for FPGAs is *not* intended to replace regular FPGA-architecture-specific synthesis, but rather to fill a particular need in certain FPGA applications where portable binaries are desired.

JIT compilation for FPGAs is also useful, in fact essential, for warp processors that we are developing, which perform dynamic hardware/software partitioning, wherein an executing binary is dynamically optimized by moving software kernels to configurable logic [20]. At the heart of warp processors, a JIT compiler implements the synthesized hardware circuits onto the on-chip configurable logic fabric.

2. ROUTING AS THE MAIN TASK OF JIT FPGA COMPILATION

JIT compilation for FPGAs requires the development of lean versions of technology mapping, placement, and routing algorithms. Starting with the standard hardware binary, the JIT compiler will first use technology mapping to map the hardware onto to the lookup-tables (LUTs) and flip-flops within the configurable logic and pack the LUTs and flip-flops into configurable logic blocks (CLBs). Once mapped, the location of each CLB within the configurable logic is determined during placement. The placement algorithm will attempt to assign locations to the CLBs to reduce the critical path of the circuit and ensure the circuit can be routed. Finally, routing is performed during which the actual wire segments used to connect CLBs together are determined.

We analyzed the execution time and memory usage requirements of existing technology mapping, placement, and routing algorithms. Figure 1 provides the average execution time and memory requirements of the T-VPack technology mapping tool [5] and the Versatile Place and Route (VPR) [5] tools, available at [7], for 18 MCNC benchmark circuits [24] executing on a 1.6 GHz Pentium workstation. We analyzed the requirements of VPR, as VPR produces the highest quality results compared to other published algorithms and incorporates enhancements to improve execution time [6]. Furthermore, VPR's algorithms have been incorporated into commercial FPGA CAD tools developed by Altera [1] and Cypress [9]. For placement, we analyzed the requirements of VPR's bounding box *(BB)* and timing-driven *(TD)* placement algorithms. While the bounding box placement algorithm's goal is to place blocks as close as possible, the timing-driven placement algorithm further strives to place blocks to maximize circuit speed. For routing, we analyzed VPR's routability-driven *(RD)* router and VPR's timing-driven *(TD)* router.

On average, technology mapping is the least demanding task, requiring only 0.3 seconds and less than 2 megabytes (MB) of data memory. For placement, VPR's bounding box placement algorithm requires an average execution time of 11.5 seconds and just over 2 MB of data memory. Alternatively, the timing-driven placement algorithm has a faster execution time of only 8 seconds on average, but requires extremely large memory resources of 25 MB. Similarly, both VPR's routing algorithms require very large memory resources as well, requiring on average 26 MB for the routability-driven algorithm and 21 MB for the timing-driven algorithm. With regards to execution time, the timing-driven router is much faster, requiring only 14.8 seconds on average, whereas the routability-driven algorithm requires over 5 minutes. In developing a JIT compiler for FPGAs, we could potentially incorporate tools using the algorithms employed by T-VPack's technology mapping and VPR's bounding box placement. However, incorporating existing FPGA routing algorithms within a JIT compiler is not possible given the extremely large memory resources and long execution times. Therefore, we focus our initial efforts on developing a lean routing algorithm and present the Riverside On-Chip Router (ROCR), designed to execute on-chip as part of a JIT FPGA compiler. We designed ROCR using the same techniques as existing routing algorithms, but applying those techniques differently in order to achieve very fast execution times using limited memory resources.

3. ROUTING-ORIENTED FPGA FABRIC

While many configurable logic architectures are currently available, traditional FPGAs were not designed 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 [23] and Altera [1], design FPGAs with complex CLBs, possibly containing varying sizes and number of lookup tables, embedded memory cells, large routing resources, large input/output resources, etc. Additionally, routing resources of commercially available FPGAs are capable of routing between CLBs of varying distance apart and often include routing channels spanning the entire length of the FPGA. While traditional FPGA architectures are beneficial in terms of creating fast and compact designs, 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, occupying over two thirds of the configurable logic fabric. 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, with the compiler including technology mapping, placement, and routing tools [15]. Figure 2(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 connected to a single switch matrix to which all inputs and outputs of the CLB can be connected. We handle 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 2(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 [19]. 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 2(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.

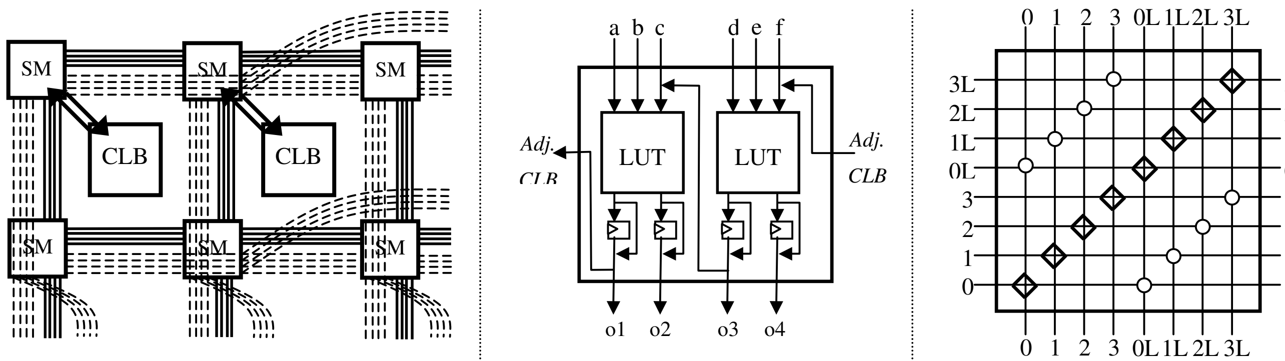
While we found that developing our own configurable logic architecture helped to develop JIT compilation for FPGAs, implementing the required lean CAD tools for on-chip execution is not trivial. Existing FPGA CAD tools are capable of producing highly optimized hardware circuits. However, these tools suffer from very large data memory usage, often exceeding 100 megabytes, and long execution, ranging anywhere from minutes to hours. We must design 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. However, our on-chip CAD tools create hardware circuits of *acceptable* quality.

4. FAST ROUTING ALGORITHM FOR JIT COMPILATION

Most FPGA routing algorithms rely on constructing a routing resource graph, in the form of a directed graph, to represent the available connection between wires and CLBs within the FPGA architecture. Using a resource graph, the FPGA router must 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, in which the router routes each net using Dijkstra's shortest path algorithm to connect the net's source and sinks [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, presented in [10], introduced the idea of negotiated congestion routing. In each routing iteration, Pathfinder routes each net using the best path possible allowing routing resources to be overused. 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. As long as illegal routes exist, the Pathfinder algorithm will continue to route, rip-up, and reroute all nets. Pathfinder's negotiated congestion algorithm produces good hardware circuits by routing nets along the critical path using the shortest path possible, even where routing congestions exists. On the other hand, the Pathfinder algorithm can route non-critical nets using longer paths away from the routing congestion.

Figure 2: (a) Simple configurable logic fabric (b) configurable logic block (CLB), and (c) switch matrix (SM) architecture.

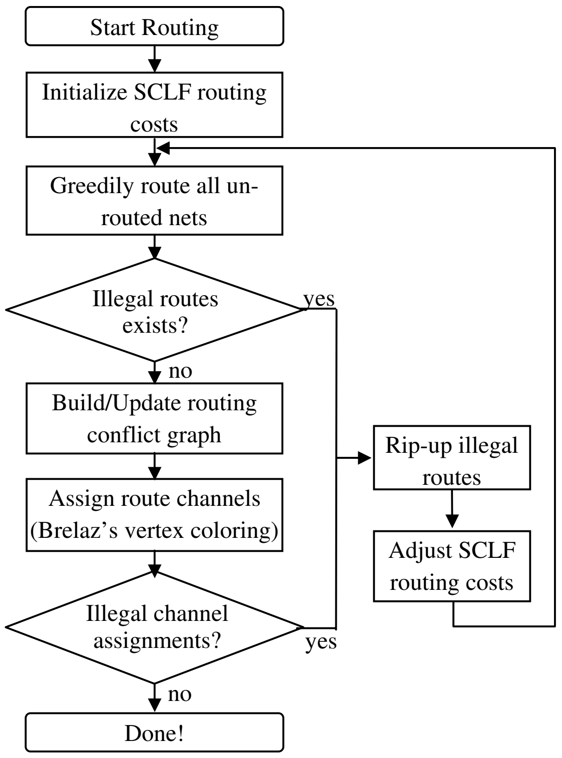


Configurable Logic Fabric
(a)

Configurable Logic Block
(b)

Switch Matrix (SM)
(c)

Figure 3: Riverside On-Chip Router (ROCR) algorithm overview.


Start Routing
Initialize SCLF routing costs
Greedily route all un-routed nets
Illegal routes exists?
yes
no
Build/Update routing conflict graph
Assign route channels (Brelaz's vertex coloring)
Illegal channel assignments?
yes
no
Done!
Rip-up illegal routes
Adjust SCLF routing costs

VPR's routability-driven router and timing-driven router algorithms rely on a modified version of the Pathfinder algorithm to decrease routing execution times and increase hardware circuit speed. The routability-driven algorithm focuses on increasing the performance of the Pathfinder algorithm and using the fewest tracks possible by incorporating a different routing cost model. VPR's timing-driven routing algorithm 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 routing algorithms' reliance on constructing a routing resource graph of the configurable logic fabric makes those algorithms difficult to use for JIT compilation.

Therefore, we 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 3 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 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, we chose to use 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.

5. RESULTS

We evaluate our lean routing tools for JIT FPGA compilation, comparing ROCR's performance, memory usage, and circuit quality to VPR's routability-drive and timing-driven routers. We created an architecture description file for our SCLF to use with the VPR tools. Starting with each benchmark's netlist, we performed technology mapping using T-VPack. The resulting mapped circuit was placed using VPR's bounding box placement algorithm. From the placement results, we determined the minimum required size for our SCLF. After placing all benchmark circuits, we used VPR's routability-driven router to route each circuit to determine the minimum routing channel width required to support all benchmark circuits. Thus, in order to support all 18 MCNC benchmarks, we considered a configurable logic fabric consisting of a 67x67 array of CLBs with a routing channel width of 30. Finally, using the bounding box placement of the benchmark circuits, we routed each circuit using VPR's routability-driven router, VPR's timing-driven router, and ROCR.

For on-chip execution, a JIT compiler requires very fast execution times and must not have large memory requirements. In designing ROCR, we strove to provide very fast execution times while minimizing the amount of memory used. Table 1 presents the execution time *(seconds)* and data memory usage *(kilobytes)* for VPR's routability-driven and timing driven routers and for ROCR for the 18 MCNC benchmark circuits. For a fair comparison, all results were obtained using a 1.6 GHz Pentium workstation. VPR's timing-driven router requires on average 14.8 seconds to route the circuits, with a maximum execution time of close to one minute. VPR's routability-driven router requires on average over 6 minutes routing the benchmarks. ROCR, in contrast, is extremely fast, requiring only 2.9 seconds on average to route the circuits and with a maximum of execution time of only 13.8 seconds. Table 1 also presents the speedup in execution time of ROCR compared with VPR's timing-driven router. On average, ROCR is 10X faster than VPR's fastest routing method (timing-driven), and up to 21X faster for one example. (Note in the table that the proper way to compute average speedup, namely averaging the individual speedups, results in a value that is different from dividing the average times).

Even more important is ROCR's more efficient memory use. While fast execution times are good for providing fast JIT compilation, if a routing algorithm requires large memory usage to achieve that speedup, such an algorithm is not feasible for JIT compilation. ROCR is able to achieve fast execution times while only requiring a maximum of 3.6 megabytes. On the other hand, VPR's routing algorithms have a maximum memory usage of roughly 48 MB, over 13X more memory than required by ROCR.

ROCR also produces good hardware circuits, both in terms of circuit speed and in terms of the amount of routing resources used to route the circuit. Figure 4 presents the critical path in nanoseconds of the hardware circuits produced by VPR's routability-driven router, VPR's timing-driven router, and ROCR. ROCR produces a circuit with a critical path on average 32% longer than VPR's timing-driven router, and 10% shorter than VPR's routability-driven router.

Table 1: Execution time *(seconds)* and data memory usage *(kilobytes)* for VPR's routability-driven (RD) and timing-driven (TD) routing algorithms and ROCR and speedup of ROCR compared to VPR's timing-driven router.

Bench-mark	VPR (RD)		VPR (TD)		ROCR		
	Time	Mem	Time	Mem	Time	Mem	S
alu4	221	16508	8.3	12312	0.6	3484	13.8
apex2	316	18780	12.4	14552	4.3	3496	2.9
apex4	214	14332	7.8	11128	0.6	3468	13.0
bigkey	404	47944	13.5	37648	1.3	3512	10.4
des	376	52276	12.8	49980	1.0	3496	12.8
diffeq	136	15484	5.8	12576	0.4	3480	14.5
dsip	231	47796	10.4	37496	0.9	3500	11.6
E64	19	6296	1.0	5644	0.1	3428	10.0
elliptic	770	33524	33.7	26244	7.8	3572	4.3
Ex5p	188	12612	6.3	9840	0.3	3460	21.0
frisc	866	33468	35.1	27112	13.8	3564	2.5
misex3	191	15628	6.8	11508	0.4	3468	17.0
s1423	4	4240	0.5	3548	0.1	3428	5.6
s298	265	18984	11.6	15384	0.7	3496	16.6
s38417	1428	57380	49.9	44180	8.7	3680	5.7
s38584.1	1110	51760	36.0	43700	8.8	3692	4.1
Seq	291	17198	11.4	13800	2.2	3488	5.2
tseng	74	11048	3.1	8960	0.2	3464	15.5
Average:	395	26403	14.8	21423	2.9	3510	**10.4**

Figure 4: Critical path *(nanoseconds)* for several MCNC benchmark circuits using VPR routability-driven (RD) and timing-driven (TD) router and ROCR.

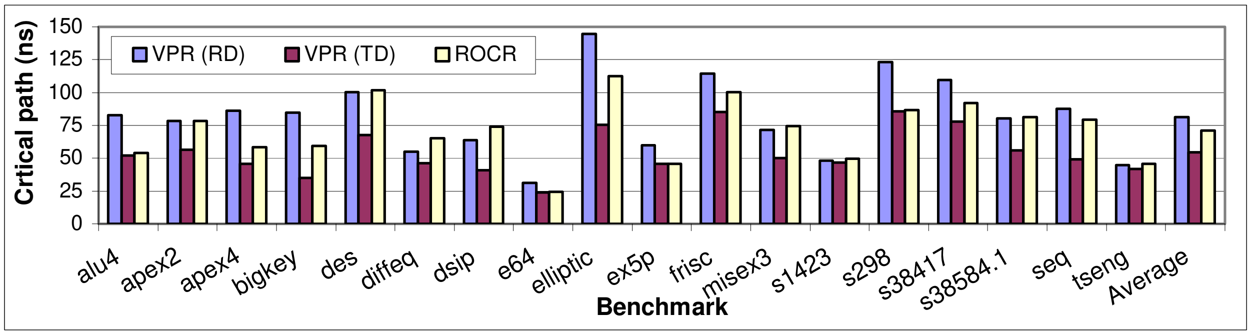
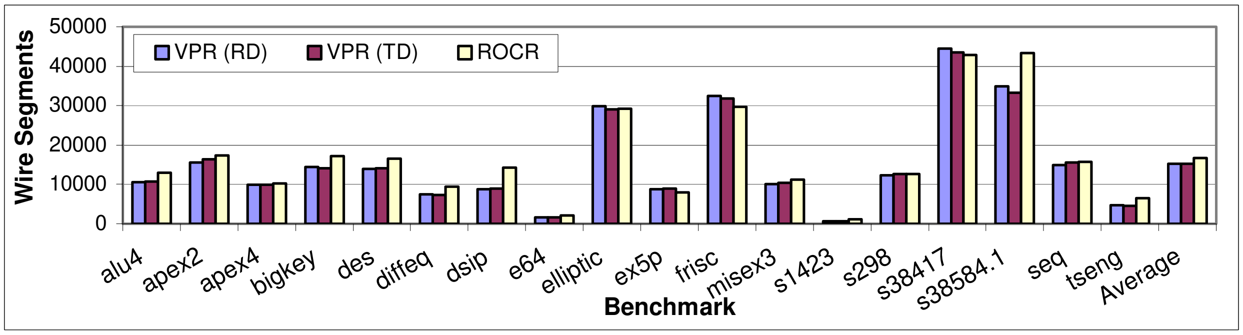


Figure 5: Total wire segments required to route several MCNC benchmark circuits using VPR routability-driven (RD) and timing-driven (TD) router and ROCR.



We can also evaluate circuit quality by the amount of routing resources required to route a circuit. Figure 5 presents the total number of wire segments used within our SCLF to route all 18 benchmark circuits using VPR's routability-driven router, VPR's timing-driven router, and ROCR. While the critical path of many hardware circuits greatly differs between VPR's timing-driven and routability-driven routers, the total number of wire segments used to route each design is very similar, both requiring roughly 15,200 wire segments. Compared with VPR's routing algorithms, ROCR performs very favorably, only requiring an average of 10% additional wire segments, or 16,700 wire segments.

These were the expected tradeoffs – ROCR runs faster and uses far less memory at the expense of quality. However, note that the quality of the results is still quite acceptable.

6. CONCLUSONS AND FUTURE WORK

JIT compilation for FPGAs enables the development of a standard binary for FPGAs and facilitates the portability of binaries across FPGA architectures. A JIT compiler must be capable of performing technology mapping, placement, and routing, of which routing is the most computationally and memory expensive task. The compiler should run on a small embedded processor in an on-chip environment, where memory resources and processing power are limited. We presented the Riverside On-Chip Router (ROCR), a lean, fast router for JIT compilation. ROCR is capable of producing good hardware circuits using 13X less memory and executing 10 times faster than VPR's fastest routing algorithm. Circuits routed with ROCR required only 10% additional routing resources than VPR, with circuit speeds only 32% slower than VPR's timing-driven router, and actually circuits speeds 10% faster than VPR's routability-driven router. With less execution time and a small memory footprint, ROCR is suitable for use within a JIT FPGA compiler.

We are currently working on developing lean versions of technology mapping and placement algorithms for our JIT compiler. While existing technology mapping and placement algorithms could be directly incorporated into our JIT compiler, we are working on developing such algorithms designed for our SCLF. Specifically, we are developing a placement algorithm designed in conjunction with both our configurable logic fabric and ROCR hoping to be able to exploit this information to choose a placement such that ROCR can perform routing more efficiently and create designs with shorter critical paths using fewer routing resources. We are also considering extending ROCR to incorporate a lean timing-driven routing algorithm to improve circuit speed and quality. Other future considerations include incorporating optimization tools within our JIT compiler. While the standard hardware binary should already be optimized from a netlist perspective, a JIT compiler can possibly further optimize the design using information about the underlying configurable logic architecture. Finally, we plan to extend JIT FPGA compilation to support more complex FPGA architectures.

7. ACKNOWLEDGEMENTS

This research was supported in part by the National Science Foundation (CCR-0203829, CCR-9876006) and by the Semiconductor Research Corporation (grant CSR 2002-RJ-1046G). We would also like to acknowledge Kees Vissers for his valuable input on developing JIT FPGA compilation.

8. REFERENCES

[1] Altera Corp. http://www.altera.com, 2003.

[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, 2003.

[4] Bala, V., E. Duesterwald, S. Banerjia. Dynamo: A Transparent Dynamic Optimization System. Conference on Programming Language Design and Implementation, 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] Cypress Semiconductor. http://www.cypress.com, 2004.

[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. Design Automation and Test in Europe Conference, 2004.

[16] Lysecky, R., F. Vahid. A Codesigned On-chip Logic Minimizer. International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2003.

[17] Lysecky, R., F. Vahid. On-chip Logic Minimization. Proceedings of the 40th Design Automation Conference (DAC), 2003.

[18] Marquardt, A., V. Betz, J. Rose. Speed and Area Trade-offs on Cluster-based FPGA Architectures. IEEE Transactions on Very Large Scale integration (TVLSI), 2000.

[19] 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.

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

[21] Stitt, G., F. Vahid. Hardware/Software Partitioning of Software Binaries. IEEE/ACM International Conference on Computer Aided Design (ICCAD), 2002.

[22] Transmeta Corporation. http://www.transmeta.com, 2004.

[23] Xilinx, Inc. http://www.xilinx.com, 2004.

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