Accelerating FPGA Routing Through Parallelization and Engineering Enhancements Special Section on PAR-CAD 2010

Marcel Gort and Jason H. Anderson

***Abstract*—We present parallelization and heuristic techniques to reduce the run-time of field-programmable gate array (FPGA) negotiated congestion routing. Two heuristic optimizations provide over 3× speedup versus a sequential baseline. In our parallel approach, sets of design signals are assigned to different processor cores and routed concurrently. Communication between cores is through the message passing interface communications protocol. We propose a geographic partitioning of signals into independent sets to help minimize the communication overhead. Our parallel implementation provides approximately 2.3× speedup using four cores and produces deterministic/repeatable results. When combined, the parallel and heuristic techniques provide over 7× speedup with four cores versus the router in the widely used Versatile Place and Route (VPR) FPGA placement/routing framework, with no significant impact on circuit speed or wirelength.**

***Index Terms*—Fast routing, FPGAs, parallel CAD, parallel FPGA routing, partitioning, routing.**

I. Introduction

THE RUNTIME of field-programmable gate array (FPGA) computer-aided design (CAD) tools is a major concern for FPGA vendors and their customers. Two factors are at play in worsening runtimes for the largest designs. First, high current density in modern processors has created a "power wall," limiting the rate of increase of clock speeds in processors, and spawning the multicore era. Second, Moore's Law charges onward—state-of-the-art chips contain two billion transistors and continue to double in size every two years. There is, consequently, a widening gap between the size of chips, and the ability of CAD tools to handle them.

The largest industrial FPGA designs take hours or days to compile from hardware description language-to-bitstream, with placement and routing being the most runtime intensive steps of the CAD flow. Long runtimes reduce engineering productivity, raise cost, and are a strong impediment to the widespread adoption of FPGAs by software developers, who are accustomed to compilation times in seconds or minutes. Improving the value proposition of FPGAs and expanding their user-base are paramount aims for commercial FPGA vendors, and lower tool runtime is key to enabling progress on these fronts. A promising direction to address the runtime challenge is to accelerate CAD tools through parallel computing. Today's FPGA CAD tools are frequently run on commodity processors with four cores, and 8 and 16-core commodity processors are not far down the road. Such processors offer significant potential for runtime reduction through parallelization. By and large, however, the underlying CAD algorithms in today's FPGA tools are single-threaded, and do not take advantage of the available processing power. The importance of leveraging multicore parallelism was underscored recently by Altera, who published techniques for multicore parallel placement [1].

In a previous publication [2], we presented fine and coarse-grained techniques for parallel FPGA routing. Our coarse-grained approach aligns closely with what one would intuitively think of as parallel routing: design signals are partitioned into sets, each of which is routed by a different processor core. Intermittent communication between processor cores is used to communicate routing results, thereby giving each core a global picture of the intermediate routing state.

In this paper, we expand on our previous work by first improving the baseline sequential routing algorithm through two "engineering enhancements" that together deliver a considerable runtime reduction. We then apply our coarse-grained method to parallelize the enhanced sequential router. We also present improvements to our parallel method, including a hierarchical geographic partitioning method, which allows signals to be split between processors such that the need for inter-processor communication is reduced. We use message passing interface (MPI) as the communication protocol [3], [4]. Our approach is implemented within the Versatile Place and Route (VPR) framework [5], and offers deterministic/repeatable results. We experimentally demonstrate the runtime benefits of our enhancements to the baseline sequential router, as well as the proposed parallelization techniques. We further show that our techniques have no appreciable impact on quality-of-result: circuit speed or wirelength.

The remainder of this paper is organized as follows. Section II provides relevant background on FPGA routing algorithms, parallel routing, and introduces the parallel programming techniques used in this paper, namely, MPI. Improvements to the sequential routing algorithm are presented in Section III. The proposed parallel routing techniques are described

Manuscript received February 6, 2011; revised May 31, 2011; accepted July 20, 2011. Date of current version December 21, 2011. This paper was recommended by Associate Editor V. Narayanan.

The authors are with the Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON M4M 2K8, Canada (e-mail: marcel.gort@utoronto.ca; janders@eecg.toronto.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2011.2165715

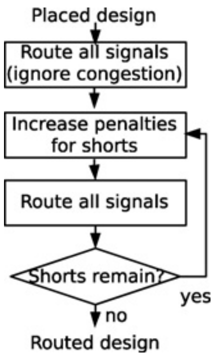
Placed design
Route all signals (ignore congestion)
Increase penalties for shorts
Route all signals
Shorts remain?
yes
no
Routed design

Fig. 1. Negotiated congestion routing flow.

in Section IV. Section V outlines the methods we used to profile the parallel algorithm, and presents the associated results. Section VI explains improvements we made to our parallel approach, including the hierarchical geographic partitioning. An experimental study is presented in Section VII. Conclusions and suggestions for future work are offered in Section VIII.

II. Background

A. FPGA Routing

The two largest FPGA vendors, Xilinx and Altera, use a variant of the PathFinder negotiated congestion routing algorithm [6] in their commercial routers [7], [8]. PathFinder is also used in the publicly available VPR FPGA placement and routing framework [5], which we parallelize in this paper. VPR is also part of the SPEC benchmark suite. Fig. 1 gives an overview of the negotiated congestion approach. First, all signals in a placed design are routed in the best manner possible (e.g., minimum delay or minimum resource usage), permitting shorts between the signals (meaning that two different signals may use the *same* wire on the FPGA). After initial routing, each signal is routed with its ideal routing solution, albeit infeasible, owing to the shorts. Then, the penalties associated with shorts are increased, and the signals are re-routed with consideration of the increased penalties. The process of increasing the penalties for shorts and re-routing the signals continues iteratively until all shorts are removed and the routing is feasible. One pass through the loop of Fig. 1 is referred to as a PathFinder iteration. In essence, signals *negotiate* among themselves for which signal gets to keep a popular/shared FPGA resource. Our parallelization approach accelerates the "route all signals" step in the negotiated congestion flow.

At the heart of negotiated congestion routing is maze expansion [9]—the algorithm used to route a load pin on a signal and the most computationally expensive aspect of FPGA routing. The routing resources in the FPGA are represented as a graph, $G(V, E)$, called the routing resource graph, where each vertex, $v \in V$, represents a routing conductor, i.e., a metal wire segment or a pin on a block. Each edge, $e \in E$, represents a programmable routing switch in the FPGA that may be turned on to electrically connect a pair of conductors. To route a load pin on a signal, maze expansion begins by adding the vertex, s, corresponding to the signal's source pin, to a priority queue. The algorithm then enters a `while` loop that executes as follows: the lowest cost node, u, is removed from the priority queue. If u is the target load pin, then a path from source to target has been found and the expansion terminates (exit the `while` loop). Otherwise, the nodes adjacent to u in G are identified and added to the priority queue. The process of removing a node from the priority queue and *expanding* it to visit its neighbors continues until the target pin is reached (drawn from the priority queue). The costs assigned to nodes in the priority queue can be based on any number of criteria, e.g. delay, capacitance, distance to the target pin, and the number of signals contending to use a node. Node costing itself can be compute-intensive, especially if sophisticated RC delay modeling is used.

B. Parallel FPGA Routing

Chan and Schlag [10] were the first to parallelize negotiated congestion FPGA routing, showing impressive speedup results of $2.5\times$ using three processors. Their work bears some similarity to our approach, with three key differences. First, they target a distributed computing framework of networked workstations, and not a modern multicore processor. Second, we take a more sophisticated approach to load balancing among processors. Third, and most important, their results are not deterministic—different routing results are produced each time the router is executed, making the approach impractical in an industrial context, where vendors and users expect identical results to be produced each time the tools are executed. Repeatability is especially crucial in early design development and debugging. The nonparallel router in VPR is completely deterministic because nodes are added to a priority queue in the same order on each run, and it is important to maintain this property.

Another work by Cabral *et al.* [11] described parallel FPGA routing specifically for a "planar" routing architecture. In a planar architecture, wires in the ith routing track within a channel may only be programmably connected to other wires that are also on the ith routing track. Consequently, routing tracks can be viewed as "planes" that do not intersect with one another. Planarity simplifies the parallelization problem, as processor cores can operate independently on each plane, reducing the need for inter-processor data sharing. While early FPGAs (such as the Xilinx XC4000 [12]) used planar routing, it has since been shown to negatively impact routability. Modern FPGA interconnect is not planar [13], [14], making Cabral's work inapplicable today.

There is considerable prior work on parallelizing the single source shortest path problem [15]–[17] which is related to maze routing, however, these parallelization techniques are not intended to be used in a highly directed (A*) graph search, which is typically used in FPGA routing. In [18], it is observed that in practice, the runtime difference between a directed and breadth first search for FPGA routing is $\sim 53\times$, which far exceeds the speedups observed with parallel shortest path algorithms. In this paper, our speedups are in addition to the algorithmic speedups achieved in [18].

C. Parallel Programming Environment

We use MPI as our parallel programming environment. MPI is a widely used communications protocol that enables separate processes to communicate with one another. The processes may be running on separate cores within a multicore processor, or may even run on entirely separate processors across a network. Unlike threads, concurrently running processes do not share memory. Communication in MPI, therefore, is handled explicitly through *messages* between processes, rather than through access to shared variables in memory. In particular, processes send and receive messages with one another, and such sends and receives can be either *nonblocking* or *blocking*. Blocking messages are used for process synchronization. For example, if a process A issues a blocking send to a process B, process A must wait for a corresponding receive from process B before A continues execution. When nonblocking messages are used, the initiating process does not wait for the destination process to respond—execution continues immediately in the initiating process. We found MPI to be convenient for implementing our parallel routing approach. Using MPI's distributed memory programming model, we were able to parallelize VPR without having to make its data structures thread safe, which reduced the amount of effort necessary to create a functional parallel implementation. One of the drawbacks of using a distributed memory model is that the overall memory usage is proportional to the number of cores that are running VPR in parallel. Section VIII describes future work that could address this issue.

III. Engineering Enhancements to PathFinder Routing

In this section, we describe two enhancements we made to the baseline VPR PathFinder implementation that we believe bring it more in line with modern industrial routers. In our prior work [2], we achieved significant speedups through parallelization of the baseline (unoptimized) VPR. In this paper, we show that similar speedups can also be achieved on a more optimized PathFinder, lending further credibility to our parallelization techniques.

A. Propagating Logic Element Outputs

A key low-level building block in an FPGA is called a logic element (LE), which (at a minimum) consists of a K-input lookup table (LUT) and a register that can be optionally bypassed for implementing combinational logic. A K-LUT can implement any logic function of up to K variables. LEs are grouped together into clusters called *logic blocks*, where a cluster contains N LEs and local intra-cluster interconnect.[1] The local interconnect permits fast connectivity between LEs that are packed into the same cluster. Packing of LEs into clusters happens prior to the placement and routing stages of the FPGA CAD flow. The placer is then concerned with placing the clusters; the router is concerned with routing the inter-cluster connections.

In the most commonly used FPGA architecture model [19], the inputs to a cluster can connect to *any* of the LE inputs within the cluster—there exists a full crossbar between the cluster inputs and the LE inputs. On the other hand, the output

[1] $K = 4$ and $N = 10$ are common parameter values.

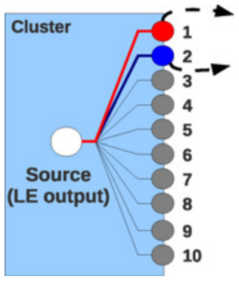

Fig. 2. Example of a signal using multiple cluster pins.

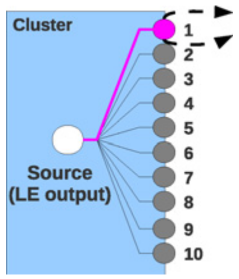

Fig. 3. Example of multiple sinks from a signal using the same cluster output.

of each LE is connected to a *single* cluster output. Despite this output limitation, because LEs are identical and due to the input crossbar, the LEs can be swapped with one another to realize output pin flexibility. This means that the routing algorithm is free to use any cluster output for a signal driven by an LE output. In fact, the routing resource graph used in the baseline VPR has edges between an LE output and all outputs of its cluster.

In the baseline VPR PathFinder implementation, during the maze routing of a multifanout signal, it is legal for that signal's source (LE output) to temporarily use several cluster outputs, as shown in Fig. 2. This can prove useful when the cluster output pins are physically located on different sides of the cluster adjacent to different routing channels, and a signal has sinks (load pins) in different directions. The cluster output used in the route from the source to the first sink may not be on the lowest cost path for subsequent sinks, so it makes sense to allow the temporary use of multiple cluster outputs. For a cluster that is not fully utilized, LEs can be duplicated so that using multiple cluster outputs is possible.

We found that the cluster output pin flexibility in VPR increases the congestion seen on the routing resource nodes corresponding to the cluster outputs, increasing the time required for PathFinder to converge. We modified this aspect of VPR to disallow routing resource node expansion around LE outputs (source) once the first sink of a multifanout signal has been routed. Subsequent sinks *must* use the same cluster output as the first sink, as shown in Fig. 3. This means that subsequent signals beginning at this cluster always have a free pin, thereby reducing contention.

B. Skipping the Re-Routing of Uncongested Signals

In the original PathFinder algorithm [6], as well as in its implementation in VPR, each signal is ripped-up and re-routed in *every* iteration, as depicted in Fig. 1. That is, even signals that do not currently use over-capacity routing resources are re-routed each iteration. We refer to such signals as *uncongested* signals. We experimentally observed that the re-routing of uncongested signals was not necessary for the PathFinder algorithm to converge to a short-free (feasible) state, and moreover, consumed significant runtime. We therefore modified the VPR router to skip the re-routing of uncongested signals—an easy-to-implement change that, despite increasing the required number of PathFinder iterations, provided a considerable overall runtime reduction. We also experimented with only skipping signals that remain uncongested for multiple consecutive PathFinder iterations. Although this reduced the number of necessary PathFinder iterations, it resulted in an overall increase in runtime.

C. *Effectiveness of Engineering Enhancements*

Table I shows the runtime, critical path delay, and total wirelength for three scenarios: baseline sequential VPR (columns labeled *base*), VPR enhanced with output propagation (columns labeled *prop*), and VPR enhanced with both output propagation and the skipping of re-routing uncongested signals (columns labeled *skip*). The results correspond to an architecture with $K = 4$ and $N = 10$ and length-2 wire segments (the full experimental methodology is described in Section V below). The first column lists the names of benchmark circuits, followed by three groups of three columns showing runtime, critical path delay, and post-routed wirelength (number of wire segments used in the routing). The results show that propagating LE outputs to cluster outputs offers a significant benefit to runtime, 1.39× versus baseline VPR, while having a negligible effect on both critical path delay and wirelength. Skipping the rip-up and re-route of uncongested signals significantly improves runtime by 2.38× on top of the output propagation enhancement, while on average incurring a slight penalty to both critical path delay (2%) and wirelength (2%). Not ripping up and re-routing uncongested signals does have the potential to harm quality of results as critical routes may be forced to choose less desirable paths in late PathFinder iterations, as evidenced by the critical path delay numbers for the benchmark circuit "spla" in Table I. For most benchmarks, however, the effect on quality of results is negligible. Together, both enhancements provide over 3× runtime speedup over baseline VPR.

IV. Multicore Parallel Routing

Our parallelization approach is to partition the signals into sets, which are then assigned to separate instances of VPR, with each instance running as a separate process on a separate processor. Each VPR instance routes its own set of signals and maintains its own data structures, including a routing resource graph and associated congestion information. The different VPR instances use MPI messages to communicate intermediate routing results with one another and synchronize their respective views of the overall routing state. By using MPI, we avoided both having to alter most of the data structures in VPR, and the requirement that many VPR functions be made thread-safe. MPI provides a mechanism whereby all VPR instances (processes) can be invoked simultaneously.

A. *Synchronization and Load Balancing*

For PathFinder to converge to a short-free routing for all signals, each VPR instance must have a relatively accurate picture of the congestion contributed by all signals. Simply put, to route signals in a short-free manner, one must know which routing resources are used by other signals and must avoid using such already-used resources. In our router, after a VPR instance routes a signal, it issues a *nonblocking* send message to all other VPR instances, meaning that it does not need to wait for other VPR instances to receive the update before continuing with its execution. The message contains the new route for the signal, as well as load balancing information (described later). Once such an update is sent, it is held in a queue by MPI, available to be received by a destination VPR instance. The key to achieving deterministic results is the use of *blocking* receive calls in the destination VPR instances. Blocking receives are issued by each VPR instance at specific/fixed points during routing. A detailed explanation is given below, but the main idea is that each VPR instance routes one or more of its own signals, then receives updates about other signals (from other VPR instances) before it proceeds to route additional signals. The point at which a VPR instance receives the updates and the specific signals about which it receives updates is *identical* from run-to-run, making our router deterministic.

Ideally, when a VPR instance wants to receive an update about the routing state of other signals, that update will already have been sent by some other VPR instance, and will be available for "pick-up" in an MPI messaging queue. However, if the update has not already been sent, since the receive is blocking, the VPR instance will stall its execution until the update is available (i.e., until the message is sent and arrives). As with any parallel algorithm, it is desirable to minimize processor stall time. In our router, this goal translates into balancing the amount of work between VPR instances, and only issuing a blocking receive at points when it is likely that an update has already been sent.

To balance the amount of work among VPR instances, we must estimate the time needed to route a signal. We considered three different prediction metrics for a signal's route time.

1) *Number of loads*: Fanout was also used as the prediction metric in [20].
2) *Bounding box*: We expect that signals with a large bounding box have a larger distance between pins, implying a longer routing time.
3) *Number of routing resource graph nodes visited during maze routing in the previous PathFinder iteration*: For each signal, we keep track of the total number of nodes visited during maze routing for the signal in the previous PathFinder iteration; we use this to predict the time needed for the signal in the current iteration.

At the beginning of each PathFinder iteration, we partition the signals into N sets, where N is the number of VPR instances (number of processor cores). Partitioning into signal sets is

TABLE I

RUNTIMES, CRITICAL PATH DELAYS, AND TOTAL WIRELENGTHS FOR THREE CASES: BASELINE SEQUENTIAL VPR, PROPAGATE LE OUTPUTS TO CLUSTER OUTPUTS, AND SKIP UNCONGESTED SIGNALS

	Runtime(s)			Critical Path Delay(s)			Wirelength		
Benchmark	Base	Prop.	Skip	Base	Prop.	Skip	Base	Prop.	Skip
cf_cordic_v_18_18_18	7.96	5.68	2.66	5.95E-09	5.95E-09	6.05E-09	38 313	38 068	38 729
cf_fir_24_16_16	38.72	41.12	11.43	1.28E-08	1.28E-08	1.28E-08	39 419	39 414	39 867
clma	22.95	14.55	7.54	1.25E-08	1.25E-08	1.26E-08	77 172	76 717	77 362
des_perf	8.99	6.89	2.23	6.17E-09	6.18E-09	6.18E-09	45 510	45 496	45 600
ex1010	13.49	8.36	5.06	9.60E-09	9.59E-09	9.60E-09	46 199	45 272	46 762
frisc	10.21	5.62	2.53	1.10E-08	1.08E-08	1.09E-08	28 601	28 381	28 837
mac2	25.86	20.67	6.00	3.09E-08	3.08E-08	3.08E-08	84 288	84 187	84 876
paj_raygentop_ hierarchy_no_mem	21.16	18.95	8.64	1.11E-08	1.11E-08	1.11E-08	38 039	37 712	38 651
pdc	24.15	12.84	7.65	1.05E-08	9.61E-09	1.06E-08	55 329	54 631	57 613
rs_decoder_2	12.61	9.09	3.99	1.91E-08	1.92E-08	1.93E-08	21 890	22 557	22 557
s38417	7.83	5.19	2.17	7.05E-09	7.05E-09	7.05E-09	33 949	33 717	34 033
spla	11.99	7.56	4.07	7.68E-09	7.68E-09	9.14E-09	35 066	35 013	36 649
sv_chip0	14.58	10.97	4.24	4.46E-09	4.46E-09	4.46E-09	79 748	79 552	80 739
sv_chip1	144.55	122.05	40.50	1.34E-08	1.34E-08	1.34E-08	156 945	156 922	156 664
sv_chip2	475.21	354.83	137.45	3.18E-08	3.19E-08	3.19E-08	510 158	509 626	509 447
Geomean	22.01	15.78	6.64	1.10E-08	1.10E-08	1.12E-08	56 804	56 632	57 551
Versus baseline	1.00×	1.39×	3.31×	1	0.99	1.01	1	1.00	1.01
Versus propagate		1.00×	2.38×		1	1.02		1	1.02

done such that the sum of the prediction metrics for the signals in each set is approximately equal. Each set is assigned to one VPR instance. Note that metrics #1 and #2 above do not change between PathFinder iterations, so the partitioning of signals is unchanged across PathFinder iterations when either of these metrics are used. Metric #3, on the other hand, is affected by routing congestion and consequently, when this metric is used, the signals may be partitioned into sets differently across successive PathFinder iterations.[2] Note that we cannot use the wall clock time needed to route a signal in the previous iteration as a prediction metric, as wall clock time is nondeterministic.

Having partitioned the signals into sets, parallel routing commences—the VPR instances begin routing their respective signal sets. After a VPR instance routes a signal, it sends a nonblocking update to all of the other VPR instances. The update message contains the signal's routing (as well as the number of routing resource nodes visited while maze routing the signal, if metric #3 above is being used). Since the send is nonblocking, the VPR instance may continue routing signals in its set, or it may decide to receive an update from other VPR instances. The decision on whether to receive an update is based on predicting whether the other VPR instances have indeed already sent an update, and, such a prediction is made using knowledge of which signals are being routed by the other instances, as well as the prediction metrics above.

Each VPR instance maintains *work counters* that estimate the amount of work performed by *other* VPR instances as they route their respective signal sets. A VPR instance uses its counters to predict whether updates have been sent by another VPR instance and whether to issue a blocking receive. We provide an example in Fig. 4. In this example, the number of loads on a signal is used to predict a signal's routing time. A set of five signals is split between two processes so that VPR instance (process) A has two signals and VPR instance (process) B has three signals. The figure shows two arrays, indexed by variable i, containing the number of loads on the signals each process is responsible for. The figure shows that the first signal belonging to process A has six loads, and the second signal has two loads. Arrows in the figure correspond to updates sent by process B and received by process A.

We refer to the work counters for processes A and B as $Work_A$ and $Work_B$, respectively. At the beginning of a PathFinder iteration, the work counters are initialized to the number of loads on the *first* signal for each VPR instance, making $Work_A = 6$ and $Work_B = 3$. Once the first signal is routed by process A, the new route is sent to process B (arrow not shown), though it is not necessarily immediately received. Then, process A compares $Work_B$ with its own work counter, $Work_A$, to determine whether it is likely that process B has sent any updates. Since $Work_B$ is smaller than $Work_A$, process A assumes process B has already sent an update for its first signal. A blocking receive is issued for Bs first signal. Once the update is received, $Work_B$ is incremented by the number of loads on the next signal in process B, making $Work_B = 3 + 2 = 5$. Since 5 is also smaller than $Work_A$, another blocking receive is issued for the second signal of process B. Process A then updates the work counter associated with process B with process Bs third signal so that $Work_B = 5 + 3 = 8$. Since 8 is greater or equal to $Work_A$, it is unlikely that process B has sent an update for its third signal. No receive is issued and process A proceeds to route its second signal (with two loads). Once it is finished routing this signal, it sends a nonblocking route update, and updates $Work_A$ with the number of loads on the signal that was just

[2]When metric #3 is used, there is no history available in the first PathFinder iteration, so we either assign each process an equal number of signals in that iteration, or use the method described in Section VI-A.

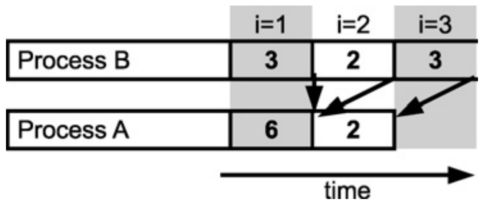

Fig. 4. Blocking receives issued by processes during a PathFinder iteration using number of loads as signal runtime prediction metric.

routed, making $Work_A = 6 + 2 = 8$. In this case, $Work_B$ is not less than $Work_A$, but process A would still like to receive an update since it has no more signals to route.[3]

The pseudocode for our parallel PathFinder implementation is shown in Fig. 5. Note that Fig. 5 only shows the portion of the router relevant to the parallelization. It does not, e.g., show code responsible for updating the congestion costs. The code is written from the perspective of one of N processors participating in the routing. We use the variable j to refer to a processor index; *local* is the index of the local processor on which the algorithm in Fig. 5 executes. $SigSet[j]$ represents the set of signals assigned to processor j for routing. The N-element array *work* holds work counters for each processor. The N-element array *sig* holds estimates of which signal is currently being routed by each processor. The *predict*(*sig*) function estimates the amount of work necessary to route a signal *sig*. The *firstSig*(*SigSet*[j]) function returns the first signal in processor js set of signals to route. Each successive call to the *nextSig*(*SigSet*[j]) function returns the next signal to be routed by processor j.

Lines 1 and 20 in Fig. 5 define a while loop which has so far been referred to as an *iteration* of PathFinder. Line 2 divides the signals into N partitions, where N is the number of processors. Each processor is aware of the set of signals belonging to every other processor. For each processor j aside from the *local* processor, lines 3–6 initialize the *sig*[j] variable to the first signal in *SigSet*[j], and then update the *work*[j] using the *predict* function. Line 7 initializes the *local* work counter to 0. Lines 8 and 19 define a while loop which executes once for each signal in the local processor's set of signals. Line 9 routes a local signal called *sigLocal*. Line 10 sends a nonblocking update to all other processor containing the *sigLocal*s updated route and some load-balancing information. On Line 11, the *predict* function is used to update the local work counter with the signal that was just routed (*sigLocal*). Lines 12 and 18 define a loop which executes once for each processor, j, aside from *local*. This loop is responsible for requesting and receiving any blocking updates from other processors that are predicted to have already been sent. Line 13 ensures that an update is only requested if it has likely already been sent. Line 14 receives a blocking update (i.e., line 15 will not execute until an update has been received). Lines 15 and 16 update the *sig*[j] and *work*[j] variables so that they reflect the next signal to route in *SigSet*[j]. Updates are requested from processor j until the work counters indicate that all sent updates have been received.

[3]We synchronize processes at the end of each PathFinder iteration to ensure that all VPR instances are working on the same iteration at the same time.

```
1:  while shorts exist do
2:      partition signals into $N$ sets
3:      for all $j$ such that $j \neq local$ do
4:          $sig[j] = firstSig(SigSet[j])$
5:          $work[j] = predict(sig[j])$
6:      end for
7:      $work[local] = 0$
8:      for all $sigLocal \in SigSet[local]$ do
9:          route $sigLocal$
10:         send non-blocking update for $sigLocal$
11:         $work[local]$ += $predict(sigLocal)$
12:         for all $j$ such that $j \neq local$ do
13:             while $work[j] < work[local]$ do
14:                 receive blocking update from processor $j$ for signal $sig[j]$
15:                 $sig[j] = nextSig(SigSet[j])$
16:                 $work[j]$ += $predict(sig[j])$
17:             end while
18:         end for
19:     end for
20: end while
```

Fig. 5. Pseudocode for multicore PathFinder with load balancing from the perspective of processor #*local*.

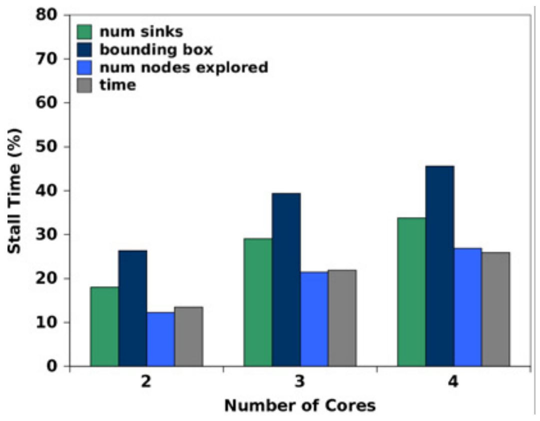

Fig. 6. Stall time (%) for signal route time prediction metrics for `clma`.

B. *Effect of Work Estimate Metric on Stall Time*

Figs. 6 and 7 illustrate the effectiveness of the different signal time prediction metrics for two benchmark circuits: `clma` and `cf_fir_24_16_16`. The vertical axis shows the average stall time over all VPR instances as a percentage of the total time needed to route all signals. Stall time was measured using the hardware counters internal to an Intel Core2Quad microprocessor. The results are given for two, three, and four VPR instances (processes). For each VPR instance, four bars are shown, corresponding to four different prediction metrics: number of loads, bounding box, number of nodes visited in maze routing, and wall clock time. Wall clock time is included for comparison only, as using this metric would lead to nondeterminism.

The figures show that the best metric for predicting the runtime needed to route a signal is the number of routing resource graph nodes visited in maze routing the signal in the prior PathFinder iteration. This metric leads to the lowest amount of stall time (aside from wall clock time), and it is therefore the best proxy for runtime. We observed the same trends for other benchmark circuits and therefore, we use the number of nodes metric for load balancing in all of the results presented in Sections V-A and VII.

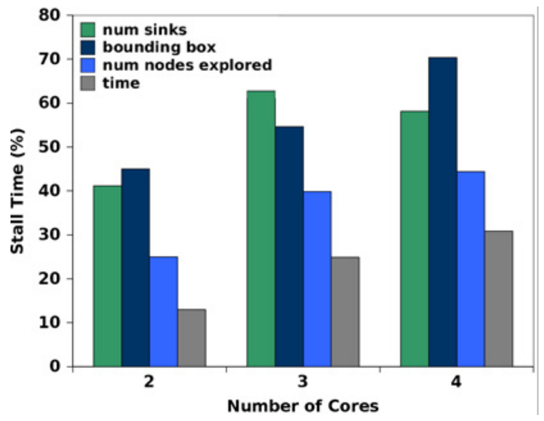

Fig. 7. Stall time (%) for signal route time prediction metrics for `cf_fir_24_16_16`.

C. Assuring Convergence

Near the end of negotiated congestion routing, when most shorts between signals have been resolved, it becomes more important for VPR instances to have an up-to-date picture of the overall routing solution for all signals. Without this, PathFinder may not converge to a short-free state. With this in mind, we may decrease the number of active VPR instances toward the end of routing if we deem that PathFinder is not making adequate progress. When the number of shorts between signals falls below an empirically determined threshold of $50 \times N$, we begin monitoring the rate of decrease in shorts between PathFinder iterations. If shorts decrease by less than 5%, we reduce the number of active VPR instances by one (of course, we always keep at least one VPR instance alive). Eventually, we may be left with a single VPR instance—sequential routing. The motivation for this is that at the end of routing, there is little work left to be done so there is little downside to using fewer processes. By reducing the number of active processes, we reduce the possibility of convergence problems.

V. Preliminary Experimental Study

In this section, we present and analyze results obtained using the parallel methods described in the previous section. We evaluate the parallelization approach against both the original VPR router as well as one which includes the engineering enhancements described in Section III.

Experiments were run on two systems running Linux (Debian 2.6.26-21), each with a Core 2 Quad Q9550 processor and 3 GB of memory. The Core 2 Quads have four cores, each running at 2.83 GHz. Benchmark circuits were selected from the 20 largest MCNC benchmarks commonly used in FPGA CAD research, and also from the set of benchmarks that are packaged with VPR 5.0 [5]. Circuits were mapped into four-input LUTs using ABC [21], then clustered using T-Vpack [22] into logic blocks with 10 four-LUTs and 22 inputs. Table II shows the benchmarks circuits used, along with the number of LUTs, latches, and signals in each. The FPGA routing architecture targeted contains bidirectional wire segments that span 2 logic block tiles. Across all runs, each circuit was routed using a fixed channel width of 1.3× the

TABLE II
Benchmark Circuits with Number of LUTs, Latches, and Signals

Benchmark	LUTs	Latches	Signals
cf_cordic_v_18_18_18	6042	2052	6099
cf_fir_24_16_16	3653	2116	4871
clma	7735	33	7799
des_perf	6580	1984	6702
ex1010	4637	0	4647
frisc	2864	886	2904
mac2	9684	524	9958
paj_raygentop_ hierarchy_no_mem	3397	1457	3848
pdc	4582	0	4598
rs_decoder_2	2612	616	2714
s38417	5884	1463	6245
spla	3756	0	3772
sv_chip0	14 341	11 785	14 523
sv_chip1	15 428	11 788	17 929
sv_chip2	37 302	16 622	34 572

minimum channel width needed to route the circuit in the single-core case with a maximum of 100 PathFinder iterations. As it is more important to achieve runtime reductions on large circuits, only those circuits with single-core runtimes of more than 10 s (when ripping up and re-routing uncongested signals) were included in our experiments; smaller circuits were excluded. The runtimes presented correspond to the time spent routing signals in the PathFinder algorithm, which represents 84% of total router runtime for baseline VPR, on average. The remaining 16% of router time includes loading the benchmark circuit, building the routing resource graph device model, performing delay analysis at the end of each PathFinder iteration, and a final post-routing timing analysis.

Table III shows our results with and without our engineering enhancements described in Section III. The first column of the table gives the name of each circuit. The next eight columns present the runtime needed to route each circuit with a given number of processes (processor cores). The 2×4 columns corresponds to using two Core 2 Quad processors across a network—included for comparison purposes only and not necessarily indicative of scalability. We measured an approximately 5× latency difference when sending a nonblocking message across a network (between processors) rather than between cores in a processor. This leads to an approximately 5% runtime penalty when some messages have to travel across a network. We reinforce that for a given number of processes, our router produces the same routing results from run-to-run—it is deterministic. The third last row of the table provides the geometric mean runtime across all circuits for a particular number of processes. The second last row gives the speedup relative to the sequential (nonparallel) baseline VPR PathFinder implementation. The last row gives the speedup relative to sequential VPR with our engineering enhancements.

Relative to baseline VPR, the left-half of Table III shows that we achieve 1.44×, 2.30×, and 2.85× speedups with 2, 4, and 2 × 4 cores, respectively. Relative to the enhanced sequential VPR, the right-half of the table shows speedups of 1.31×, 2.03×, and 2.32×, with 2, 4, and 2 × 4 cores,

TABLE III
RUNTIMES VERSUS THE SEQUENTIAL BASELINE WITH AND WITHOUT ENGINEERING ENHANCEMENTS

Benchmark	No. of Cores (N)							
	Baseline VPR				VPR with Engineering Enhancements			
	1	2	4	2 × 4	1	2	4	2 × 4
cf_cordic_v_18_18_18	7.96	5.53	3.91	4.00	2.66	2.24	1.43	1.51
cf_fir_24_16_16	38.72	27.17	19.04	12.32	11.43	8.69	6.32	5.08
clma	22.95	14.55	9.33	8.08	7.54	7.97	3.64	2.75
des_perf	8.99	6.72	3.17	2.69	2.23	1.55	0.92	0.98
ex1010	13.49	8.68	5.65	3.43	5.06	3.21	1.90	1.62
frisc	10.21	6.25	4.76	3.64	2.53	1.95	1.54	1.33
mac2	25.86	18.94	14.51	10.21	6.00	4.80	3.78	3.14
paj_raygentop_ hierarchy_no_mem	21.16	15.29	9.91	10.88	8.64	7.61	4.87	5.35
pdc	24.15	13.45	7.12	6.12	7.65	5.31	3.34	2.53
rs_decoder_2	12.61	8.41	4.99	3.88	3.99	2.88	1.93	1.62
s38417	7.83	6.14	3.21	3.17	2.17	1.48	1.09	1.10
spla	11.99	7.63	4.90	3.28	4.06	3.12	1.83	1.62
sv_chip0	14.58	11.78	6.08	6.91	4.24	2.99	1.99	2.10
sv_chip1	144.55	115.83	67.84	49.11	40.50	30.46	19.71	14.02
sv_chip2	475.21	345.94	257.01	165.82	137.44	101.82	63.88	42.92
Geomean	22.01	15.28	9.55	7.72	6.64	5.05	3.27	2.86
Speedup versus baseline 1 core	1.00×	1.44×	2.30×	2.85×	3.31×	4.35×	6.73×	7.70×
Speedup versus enhanced 1 core					1.00×	1.31×	2.03×	2.32×

respectively. Clearly, the benefits of parallelization are less pronounced when applied in tandem with the engineering enhancements. In the next sections, we analyze where runtime is being spent in our parallel implementation, and then describe improvements to our initial parallelization approach.

A. Profiling

In this section, we categorize the different phases of the serial and parallel PathFinder algorithms to better analyze and improve the overall runtime. The runtime of sequential VPR routing can be split into three categories. The first category includes the runtime associated with reading the placement and netlist files from disk, allocating and loading the routing resource graph in memory, performing a final timing analysis, and writing the final results to disk. We do not include this runtime in our analysis. The second category is the route all signals function described in Section II. The third category includes the portion of runtime spent updating delay and slack information at the end of each PathFinder iteration. We separate categories 2 and 3 because in this paper, we do not attempt to parallelize category 3, which represents approximately 7% of runtime in our enhanced sequential algorithm and approximately 2% of runtime for the baseline VPR. We include only category 2 runtime in our final speedup figures.

Although the maximum speedup of a parallel algorithm implementation over its serial equivalent is $N\times$, where N is the number of processors, speedups close to this maximum are rare for algorithms that are not embarrassingly parallel. Several factors prevent us from reaching this maximum. In our parallel implementation of PathFinder, the category 2 runtime mentioned above incurs its own overhead that is not present in the serial implementation. With N total processors, for a processor i, we use $\text{route\_time}_{i/N}$ to represent the total time spent routing signals by the processor, and we use $\text{route\_time}_{i/N,j}$ to represent the time processor i spends routing signals in iteration j of PathFinder. We use $\text{wait\_time}_{i/N,j}$ to represent the time processor i spends waiting for updates from other processors in PathFinder iteration j. Using these runtime metrics, we attempted to breakdown runtime further as follows.

1) *Ideal Time:* We first define the *ideal* time to be that corresponding to a perfect parallelization without any stalls as follows:

$$t_{\text{ideal},i/N} = \text{route\_time}_{1/1}/N. \tag{1}$$

The ideal time is the sequential time divided by N (the number of processors).

2) *Stall Time:* Our parallel PathFinder implementation uses blocking receive messages to get updates about signals being routed in other processors. If a processor issues a blocking receive call but the corresponding update has not yet been sent, its execution will stall. This is due to inaccuracy in our work estimates. We define stall time as the time penalty incurred by processor stalling that could have been avoided had all receive messages been delayed to the end of an iteration. This is an important distinction because it distinguishes stall time from load imbalance time. Alternatively, the stall time can also be viewed as the time penalty incurred from processor stalling given perfect load balancing. Stall time is estimated as follows:

$$t_{\text{stall},N} = \sum_{j=0}^{I}(\min_{0\le i<N} \text{wait\_time}_{i/N,j}) \tag{2}$$

where I is the total number of PathFinder iterations and $t_{\text{stall},N}$ refers to the stall time across all processors when using N total processors.

The stall time is the wait time due to factors *other* than a load imbalance. Since the processor with the least amount of work experiences no delay due to load imbalance, all of its wait time can be attributed to stall time, hence the `min` in (2) above.

3) *Load Imbalance:* At the beginning of each PathFinder iteration, the signals are partitioned into N sets, such that the sum of the work estimates in each set is approximately equal. Each of these sets is then assigned to a processor. If the work estimates are inaccurate and the sum of the actual amount of work performed by one processor is more than that performed by another, there will be a load imbalance. For a PathFinder iteration, the time penalty incurred as a result of a load imbalance is approximated as the maximum route_time minus the average route_time as follows:

$$t_{\text{imbalance},N} = \max_{0 \le i < N} \text{route\_time}_{i/N} - \frac{1}{N} \times \sum_{i=0}^{N} (\text{route\_time}_{i/N}) \tag{3}$$

where $t_{\text{imbalance},N}$ refers to the imbalance time across all processors when using N total processors.

4) *Slower Convergence:* It is possible that stale congestion information in a processor causes it to make suboptimal routing decisions, leading to slower resolution of over-utilized routing resources. As a result, the number of PathFinder iterations may increase and in general, slower convergence will lead PathFinder to do more work. Let work_sum_N represent the total amount of work done when N processors are used, computed as the sum of the total number of routing resource nodes explored for all signals over all PathFinder iterations. The runtime penalty incurred due to the increased work is estimated as follows:

$$t_{\text{converge}} = \left(\frac{\text{work\_sum}_N}{\text{work\_sum}_1} - 1\right) \times \text{run\_time}_{1/1} \tag{4}$$

where $\text{run\_time}_{1/1}$ represents the sequential runtime.

5) *Serial Time:* Near the end of the routing process, the number of cores being used is reduced to improve convergence. The benefit of doing so can outweigh the cost of using fewer cores. The decision of when to reduce the number of used cores can have a large impact on overall runtime. Reducing core count too soon will waste runtime, as spending even a modest portion of the algorithm in serial can quickly reduce the resultant speedup. Conversely, reducing core count too late will waste time if out-of-date congestion information is negatively impacting convergence. Consider an N-core PathFinder run where the number of processors is reduced toward the end of the routing process to aid convergence, and let I_{final} be the set of (final) PathFinder iterations where fewer than N processors are used (i.e., at least one processor has been made inactive). t_{serial} is the estimated runtime penalty that results from using fewer than the total number of available cores, and we estimate it as follows:

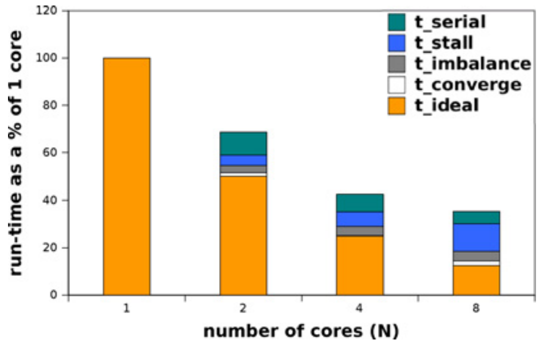

Fig. 8. Runtime contributions for initial parallelization of baseline VPR.

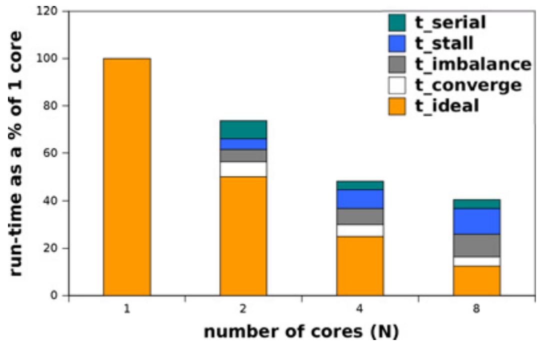

Fig. 9. Runtime contributions for initial parallelization of VPR with engineering enhancements.

$$t_{\text{serial},N} = \sum_{j \in I_{\text{final}}} \left(\max_{0 \le i < N} (\text{route\_time}_{i/N,j}) - t_{\text{ideal},N,j} \right) \tag{5}$$

where $t_{\text{ideal},N,j}$ represents the ideal time for iteration j of PathFinder had all N processors been kept active.

6) *Profiling Results:* Figs. 8 and 9 show the runtime contributions of t_{stall}, $t_{\text{imbalance}}$, t_{converge}, and t_{serial} with and without our engineering enhancements to the baseline sequential VPR routing algorithm. It should be noted that there are other effects not taken into account that make the total of the runtime contributions in these figures an underestimate by as much as 4%. Nevertheless, they are useful for determining the sources of nonideality.

A notable difference in Fig. 9 compared to Fig. 8 is the significant increase in t_{converge}. It is possible that this could be improved by reducing the number of used processors at an earlier PathFinder iteration, though this would likely result in an increase in t_{serial}.

Another notable difference is the increase in $t_{\text{imbalance}}$ as a percentage of total runtime. This increase stems from the difficulty of estimating how much work it will take to route a signal, when it is possible that the signal will not be ripped up and re-routed. Even if there is congestion on a signal's route at the beginning of an iteration (when work estimates are made), when it comes time to route the signal, that congestion may have been resolved. In such a case, the work estimate is a very poor proxy for the actual time needed to process that signal, which is zero (as the signal is not re-routed).

VI. Parallel Routing Improvements

Improvements to our parallel routing techniques are needed because of the difficulty of parallelizing a PathFinder implementation that incorporates the engineering enhancements of Section III. We offer two methods in this section that aim to improve stall time.

A. Enhancements to the First PathFinder Iteration

In VPR's implementation of PathFinder, the first iteration is used to discover the best-case delay for each signal, ignoring congestion. This means that the order in which the signals are routed does not matter, since congestion does not play a role in which route is chosen for each signal.[4] We can take advantage of this property through a more flexible allocation of signals to processors. Instead of deciding at the beginning of a PathFinder iteration which signals will be routed by which processor, we keep a pool of signals in processor 0. The other processors then issue requests to processor 0 for signals to route. Processor 0 frequently polls for these requests, to ensure that there is very little stall time.

B. Hierarchical Geographic Routing

In VPR's PathFinder implementation, not all signals have the potential to use the same routing resources. VPR computes a bounding box for each signal, outside of which routing resources cannot be used [18]. This bounding box is the smallest bounding box that contains all pins of the signal plus a buffer in each direction, which by default is equal to three FPGA tiles. If two signals have nonintersecting bounding boxes, they cannot use any of the same routing resources, and we say that these signals are *geographically independent*.

1) *Recursive Bipartitioning Technique:* We developed a recursive geographic bipartitioning technique that takes advantage of geographical independence between signals. The way in which geographic bipartitioning is applied recursively is best visualized using a slicing tree, shown in Fig. 10 for eight processors. Each node in the slicing tree holds a signal set that is geographically independent from the signals sets in all other nodes at the same level. The processors responsible for routing a node's signal set are labeled.

We first describe the top level of recursion, which corresponds to the top level of the slicing tree. At the top level, the FPGA is split into two regions. We call these the left and right regions, though they could also be the top and bottom regions. A signal with a bounding box that resides entirely within one region is geographically independent from a signal with a bounding box that resides entirely within a different region. The signals in the left region form the left_signal_set, while the signals in the right region form the right_signal_set. The signals with bounding boxes that overlap both regions form the crossing_signal_set. Because no geographic independence can be guaranteed between signals in the crossing_signal_set, all processors must receive updates about these signals when they are routed. For this reason, this set is assigned to the

[4]By default, the VPR routing cost function in the first iteration places a small weight on congestion. In order to make the first iteration entirely *congestion agnostic*, we altered the cost function to completely ignore congestion.

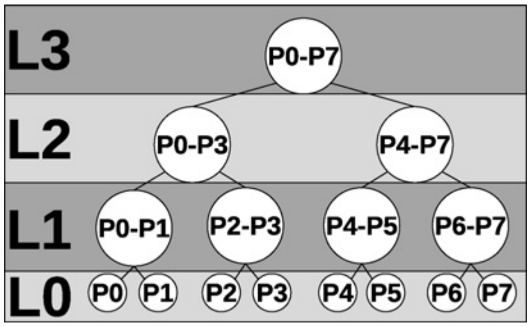

Fig. 10. Slicing tree for our recursive bipartitioning algorithm.

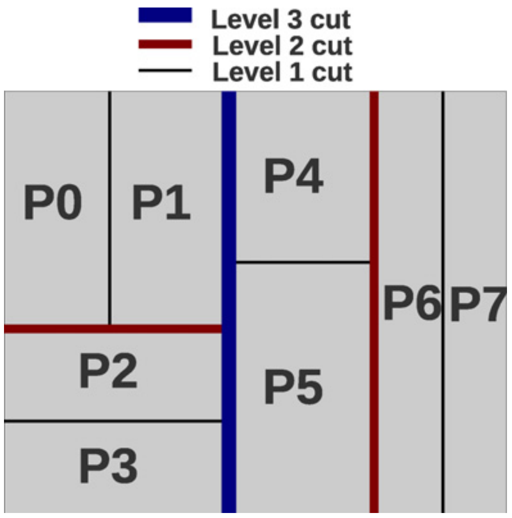

Fig. 11. Example of how the recursive bipartitioning could split the FPGA area between eight processors.

top level node in the slicing tree, labeled "P0–P7." This level of the slicing tree is now fully processed. If there were only two processors, the recursive bipartitioning process would be complete.

The left and right regions can now both themselves be bipartitioned, further splitting the signals in the left_signal_set and the right_signal_set into two parts. To do this, we call the recursive bipartitioning function first on the left_signal_set, then on the right_signal_set, specifying the region of the FPGA that corresponds to each signal set, and that the signals should be split between $N/2$ processors. The base case is reached when a signal set is split between only two processors. When this occurs, the left_signal_set is assigned to the node in the slicing tree corresponding to one of the two processors, and the right_signal_set is assigned to the node in the slicing tree corresponding to the other processor. Fig. 11 shows a possible result of the recursive-bipartitioning when using eight cores. This figure shows that regions may be cut vertically or horizontally.

2) *Routing in Phases:* Once the signals are assigned to nodes in the slicing tree, they must be routed. The routing begins by processing the node at the top level of the slicing tree (level 3). The signal set at this node is split between all processors such that the sum of the work estimates in each processor is approximately even. The signals are then routed,

TABLE IV
RUNTIMES OF IMPROVED PARALLEL ROUTING (WITH GEOGRAPHIC PARTITIONING) VERSUS THE SEQUENTIAL BASELINES WITH AND WITHOUT ENGINEERING ENHANCEMENTS

Benchmark	No. of Cores (N)							
	Baseline				Enhanced			
	1	2	4	2 × 4	1	2	4	2 × 4
cf_cordic_v_18_18_18	7.96	4.78	3.27	3.33	2.66	1.71	1.19	1.12
cf_fir_24_16_16	38.72	26.78	17.17	11.44	11.43	7.37	4.66	3.19
clma	22.95	16.36	10.15	7.25	7.54	4.89	3.35	2.32
des_perf	8.99	4.64	2.79	2.50	2.23	1.43	0.90	0.79
ex1010	13.49	8.60	5.80	3.71	5.06	2.95	1.75	1.37
frisc	10.21	6.97	5.84	5.13	2.53	1.79	1.21	1.02
mac2	25.86	18.10	17.52	11.57	6.00	4.34	3.20	2.30
paj_raygentop_ hierarchy_no_mem	21.16	15.48	15.37	9.26	8.64	5.36	3.25	3.88
pdc	24.15	16.19	7.30	5.89	7.65	5.00	3.00	3.04
rs_decoder_2	12.61	7.70	4.51	3.74	3.99	3.07	1.54	1.26
s38417	7.83	5.05	4.25	2.98	2.17	1.56	1.15	1.05
spla	11.99	7.26	5.01	4.16	4.07	2.76	1.79	1.60
sv_chip0	14.58	8.62	5.86	5.87	4.24	2.93	1.93	1.69
sv_chip1	144.55	105.24	62.59	55.02	40.50	25.95	17.67	9.34
sv_chip2	475.21	341.48	264.61	218.66	137.45	78.41	56.77	40.76
Geomean	22.01	14.38	9.99	7.86	6.64	4.38	2.85	2.34
Speedup versus baseline 1 core	1.00×	1.53×	2.20×	2.80×	3.31×	5.02×	7.71×	9.42×
Speedup versus enhanced 1 core					1.00	1.51	2.33	2.84

sending updates as explained in Section IV. All processors must finish routing their signals before continuing to the next routing phase (next level in the slicing tree). This ensures that each processor has up to date congestion information in its assigned region at the beginning of the next phase.

The routing process continues by routing signals at next level of the slicing tree (level 2), dividing signals at each node between the processors assigned to that node. In the slicing tree example, the node labeled "P0–P3" at level 2 contains signals that are split between processors 0 to 3 and are then routed, while the node labeled "P4–P7" contains signals that are split between processors 4 to 7 and are then routed. At this level, no communication is necessary between processor sets 0–3 and 4–7. Processors 0–3 must finish routing all their assigned signals at this level before any of these processors may continue to the next phase of routing. The same is true of processors 4–7.

This routing continues in phases until the leaf nodes at level 0 of the routing tree have been processed. Updates about geographically independent signals that were not sent/received during the routing process are now sent/received. This ensures that each processor has an up-to-date view of the congestion across the entire FPGA before beginning the next PathFinder iteration.

Fig. 12 shows the communication between processors that occurs at each phase in the routing process when using four processors. Each phase is labeled with the level of the slicing tree that holds the signals routed at this phase.

3) *Choosing a Cut:* There are three factors that we consider that determine the quality of a cut that bipartitions a region of the FPGA. In our description of these factors, we use the terms work_left and work_right to refer to the sum of work estimates in the left_signal_set and the right_signal_set, respectively.

TABLE V
SINGLE CORE VERSUS 4-CORE QUALITY OF RESULTS, BOTH WITH ENGINEERING ENHANCEMENTS

Benchmark	No. of Cores (N)			
	Total Wirelength		Critical Path Delay	
	1	4	1	4
cf_cordic_v_18_18_18	38 729	38 880	6.05E-09	5.95E-09
cf_fir_24_16_16	39 867	39 833	1.28E-08	1.28E-08
clma	77 362	77 674	1.26E-08	1.26E-08
des_perf	45 600	45 808	6.18E-09	6.18E-09
ex1010	46 762	46 991	9.60E-09	9.60E-09
frisc	28 837	29 373	1.09E-08	1.09E-08
mac2	84 876	85 219	3.08E-08	3.07E-08
paj_raygentop_ hierarchy_no_mem	38 651	38 692	1.11E-08	1.11E-08
pdc	57 613	58 833	1.06E-08	9.59E-09
rs_decoder_2	22 557	22 878	1.93E-08	2.04E-08
s38417	34 033	34 245	7.05E-09	7.05E-09
spla	36 649	37 415	9.14E-09	8.92E-09
sv_chip0	80 739	80 309	4.46E-09	4.46E-09
sv_chip1	156 664	156 889	1.34E-08	1.33E-08
sv_chip2	510 158	510 916	3.18E-08	3.19E-08
Geomean	57 557	57 940	1.12E-08	1.11E-08
Versus to 1 core		1.01		0.99

a) The first factor to consider is the load imbalance in the crossing_signal_set that results from having to split it between processors during the routing process. It may not be possible to split the signals in the crossing_signal_set in a way that results in approximately even amounts of

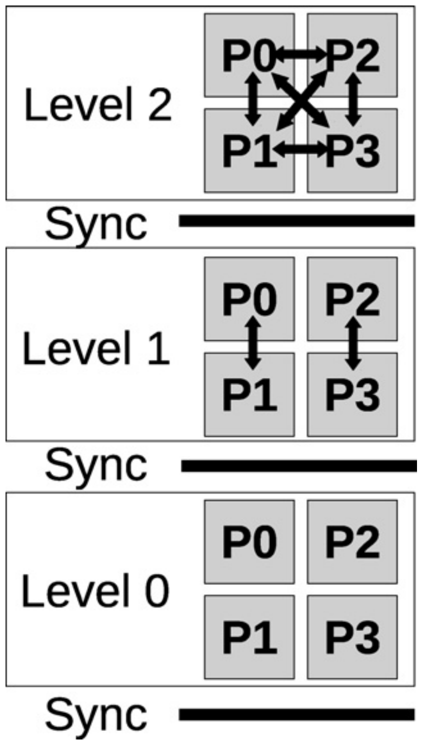

Fig. 12. Communication between four processors during routing when using hierarchical geographic routing.

work in each processor. This can be the case if there are many processors or if there are signals with very high work estimates (since a single signal is not split between processors). In such a case, it is beneficial to move additional signals from the left and right signal sets to the crossing_signal_set. When the crossing_signal_set is partitioned, these extra signals will be assigned to processors that have very little work to do, thereby improving load balancing. We measure the load imbalance in the crossing_signal_set by temporarily partitioning the signals in this set between N temporary processors. We move signals from the left or right signal set (whichever has more work) to the temporary processor with the least amount of work until the maximum work of the temporary processors divided by the average work of the temporary processors is less than 1.05.

b) The second factor to consider is that work_left and work_right should be approximately equal. This is important because it reduces the load imbalance at the end of each routing phase. We can make work_left and work_right approximately equal by moving signals that cause an imbalance from the left or right signal sets to the crossing_signal_set. Signals are moved from the left or right signal sets until work_left is approximately equal to work_right. We refer to the average of the now approximately equal work_left and work_right as as work_balanced.

c) At later routing phases, there are fewer processors communicating updates, so there is less opportunity for stall time. Choosing cuts such that more work is done at later routing phases is beneficial to stall time. By that reasoning, it is important to maximize work_balanced, since we want to assign as many signals as possible to lower levels in the slicing trees.

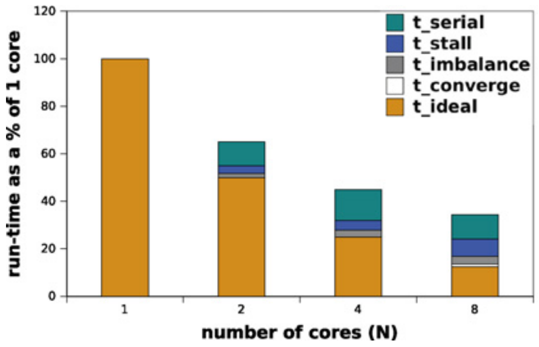

Fig. 13. Runtime contributions for improved parallelization of baseline VPR.

We use a greedy algorithm to choose a vertical or horizontal cut that maximizes work_balanced after having adjusted for load imbalance in the crossing_signal_set and between the left_signal_set and the right_signal_set. Maximizing work_balanced leads to more signals being routed at later routing stages, when communication between processors is reduced.

VII. Experimental Study

We now evaluate our improved approach to parallel routing (with geographic partitioning) using the same methodology as described in Section V. Table IV shows the runtime (in seconds) as a function of the number of VPR instances (processes) for parallelizing the baseline VPR, as well as VPR with our engineering enhancements.

Without our enhancements to the sequential algorithms, we observe speedups of 1.53×, 2.20×, and 2.80× with 2, 4, and 2 × 4 processor cores, respectively. Since these speedups are similar to the preliminary results reported in Section V, it is interesting to compare Fig. 13, which shows the runtime contributions of t_{stall}, $t_{\text{imbalance}}$, t_{converge}, and t_{serial} after our parallel routing improvements, with Fig. 8, which shows the runtime contributions before the improvements. Although using our hierarchical geographic partitioning technique did significantly reduce t_{stall}, there was an approximately equal increase to t_{serial}. We believe the reason for this is that geographic partitioning increases the likelihood of high-fanout bounding-box signals being routed concurrently. Such signals are not "local" to any partition, and must be routed in the first phase of routing, when all processors are communicating. Consequently, the congestion associated with such signals is more often out-of-date between processors, making congestion difficult to resolve, increasing serial time.

With our engineering enhancements to the baseline VPR (Section III), we see speedups of 1.51×, 2.33×, and 2.84×, with 2, 4, and 2 × 4 processor cores, respectively. When comparing Fig. 14, which shows the runtime contributions of t_{stall}, $t_{\text{imbalance}}$, t_{converge}, and t_{serial} after our improvements, with Fig. 9, it is clear that stall time is significantly reduced by the geographic partitioning technique.

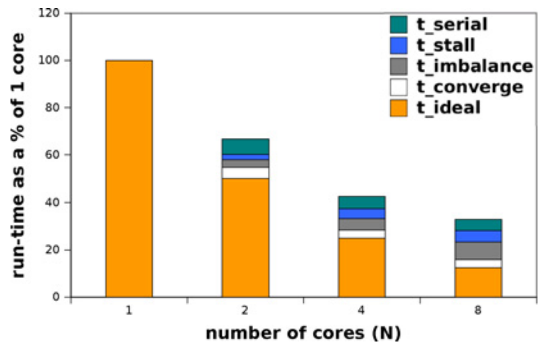

Fig. 14. Runtime contributions for improved parallelization of VPR with engineering enhancements.

Finally, combining our sequential and parallel improvements and comparing the runtimes to baseline VPR (second last row of the table), we observe speedups of 5.02×, 7.71×, and 9.42× using 2, 4, and 2×4 processors, respectively. We consider the results encouraging and we believe they should keenly interest FPGA vendors and users. For comparison, we note that Altera reported a speedup of 2.2× using four cores in their recent parallel placer work [1].

A. Impact on Quality of Results

In our parallel routing approach, the VPR instances executing concurrently operate with slightly stale congestion information, and consequently, it is conceivable that quality-of-result could be adversely impacted. We studied the impact on quality-of-result using two metrics: 1) the total number of used FPGA wire segments after routing, and 2) the post-routing critical path delay. The number of wire segments is a proxy for the total capacitance of the routing for all design signals.

We found the impact of parallelization on quality-of-result to be very small. The average change to both the number of wire segments and to the critical path delay was less than 1%. Table V shows the critical path delay and the total wirelength that results from running with four cores with our parallel improvements, as compared to one core. Critical path delay actually improved slightly in the parallel version, which we attribute to algorithmic noise.

VIII. Conclusion

Parallel computing is a promising avenue for reducing the runtime of FPGA CAD tools. In this paper, we presented new techniques for improving the performance of parallel routing for FPGAs as well as described two engineering enhancements to improve the runtime of sequential negotiated congestion FPGA routing. We presented a method of partitioning signals using a hierarchical geographic approach such that communication requirements between processors is minimized. Additionally, we showed how engineering enhancements to the sequential routing algorithm can lead to large speedups. We demonstrated that our parallel routing techniques work well for parallelizing a more realistic, aggressive, sequential routing algorithm. Results show that parallelization alone provides about 2.3× speedup using four cores, and that parallelization combined with the engineering enhancements provides over 7× speedup over a popular baseline FPGA routing implementation (VPR). The parallel router produces deterministic/repeatable results, with no considerable impact to quality-of-result (performance or wirelength).

Future Work: As future work, we believe that new techniques are necessary to push parallel FPGA routing to the many-core era. The main barrier to continued scalability is the presence of high-fanout signals that span a large portion of the chip. We see two potential methods to extract parallelism from these signals. The first is to use more fine-grained techniques, as we did in [2]. Unfortunately, those techniques only provided modest speedups, so improvements would have to be made. The second potential method is to break these signals into smaller parts, either in the routing stage of the CAD flow or at an earlier stage. As related future work, the routing resource graph could be split between processors, necessitating a more strict geographic partitioning, but allowing memory use to scale moderately as the number of processors increases.

References

[1] A. Ludwin, V. Betz, and K. Padalia, "High-quality, deterministic parallel placement for FPGAs on commodity hardware," in *Proc. ACM/SIGDA Int. Symp. FPGAs*, 2008, pp. 14–23.
[2] M. Gort and J. H. Anderson, "Deterministic multi-core parallel routing for FPGAs," in *Proc. Int. 7th Conf. Field Program. Technol.*, 2010, pp. 78–86.
[3] *Open MPI: Open Source High Performance Computing*. (2010) [Online]. Available: http://www.open-mpi.org
[4] W. Gropp, E. Lusk, and R. Thakur, *Using MPI-2*. Cambridge, MA: MIT Press, 1999.
[5] J. Luu, I. Kuon, P. Jamieson, T. Campbell, A. Ye, M. Fang, and J. Rose, "VPR 5.0: FPGA CAD and architecture exploration tools with single-driver routing, heterogeneity and process scaling," in *Proc. ACM/SIGDA Int. 7th Symp. FPGAs*, 2009, pp. 133–142.
[6] L. McMurchie and C. Ebeling, "PathFinder: A negotiation-based performance-driven router for FPGAs," in *Proc. ACM/SIGDA Int. Symp. FPGAs*, 1995, pp. 111–117.
[7] S. Gupta, J. H. Anderson, L. Farragher, and Q. Wang, "CAD techniques for power optimization in Virtex-5 FPGAs," in *Proc. IEEE Custom Integr. Circuits Conf.*, Sep. 2007, pp. 85–88.
[8] R. Fung, V. Betz, and W. Chow, "Simultaneous short-path and long-path timing optimization for FPGAs," in *Proc. IEEE Int. Conf. Comput. Aided Design*, Nov. 2004, pp. 838–845.
[9] C. Lee, "An algorithm for path connections and its applications," *IRE Trans. Electron. Comput.*, vol. EC-10, no. 2, pp. 364–365, 1961.
[10] P. K. Chan and M. D. F. Schlag, "Acceleration of an FPGA router," in *Proc. IEEE Symp. FPGA-Based Custom Comput. Mach.*, Apr. 1997, pp. 175–181.
[11] L. Cabral, J. Aude, and N. Maculan, "TDR: A distributed-memory parallel routing algorithm for FPGAs," in *Proc. Int. 7th Conf. Field Program. Logic Applicat.*, 2002, pp. 263–270.
[12] *XC4000 FPGA Data Sheet*, Xilinx, Inc., San Jose, CA, 1999.
[13] *Stratix-III FPGA Family Data Sheet*, Altera Corporation, San Jose, CA, 2008.
[14] *Virtex-5 FPGA Data Sheet*, Xilinx, Inc., San Jose, CA, 2007.
[15] U. Meyer and P. Sanders, "Parallel shortest path for arbitrary graphs," in *Proc. Int. 7th Euro-Par Conf. Parallel Process.*, 2000, pp. 461–470.
[16] E. Cohen, "Using selective path-doubling for parallel shortest-path computations," *J. Algorithms*, vol. 22, no. 1, pp. 30–56, 1997.
[17] J. R. Driscoll, H. N. Gabow, R. Shrairman, and R. E. Tarjan, "Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation," *Commun. ACM*, vol. 31, no. 11, pp. 1343–1354, 1988.
[18] J. Swartz, V. Betz, and J. Rose, "A fast routability-driven router for FPGAs," in *Proc. ACM/SIGDA Int. Symp. FPGAs*, 1998, pp. 140–149.

[19] V. Betz, J. Rose, and A. Marquardt, *Architecture and CAD for Deep-Submicron FPGAs*. Boston, MA: Kluwer, 1999.

[20] P. Chan and M. Schlag, "New parallelization and convergence results for NC: A negotiation-based FPGA router," in *Proc. ACM/SIGDA Int. Symp. FPGAs*, 2000, pp. 165–174.

[21] B. L. Synthesis and V. Group. (2007). *ABC: A System for Sequential Synthesis and Verification*. Release 70930 [Online]. Available: http://www.eecs.berkeley.edu/∼alanmi/abc

[22] A. Marquardt, V. Betz, and J. Rose, "Using cluster based logic blocks and timing-driven packing to improve FPGA speed and density," in *Proc. ACM/SIGDA Int. Symp. FPGAs*, 1999, pp. 37–46.

Marcel Gort received the B.A.Sc. degree from the University of Western Ontario, London, ON, Canada, in 2007, and the M.A.Sc. degree from the University of British Columbia, Vancouver, BC, Canada, in 2009, both in computer engineering. Currently, he is pursuing the Ph.D. degree from the University of Toronto, Toronto, ON, Canada, working on fast and scalable computer-aided design algorithms for field-programmable gate arrays (FPGAs) as well as FPGA architectures amenable to these algorithms.

He interned at the IBM Toronto Software Laboratory, Markham, ON, Canada, working with the Compiler Group, while an undergraduate student.

Jason H. Anderson (S'96–M'05) received the B.S. degree in computer engineering from the University of Manitoba, Winnipeg, MB, Canada, in 1995, and the M.A.Sc. and Ph.D. degrees in electrical and computer engineering from the University of Toronto (U of T), Toronto, ON, Canada, in 1997 and 2005, respectively.

He is currently an Assistant Professor with the Department of Electrical and Computer Engineering (ECE), U of T. In 1997, he joined the Field-Programmable Gate Array (FPGA) Implementation Tools Group at Xilinx, Inc., San Jose, CA. From 2005 to 2008, he managed groups at Xilinx focused on strategic research and development projects. He became a Principal Engineer at Xilinx in 2007. He joined the ECE Department at U of T in 2008. He has authored numerous papers in refereed conferences and journals, and holds over 20 issued U.S. patents. His current research interests include all aspects of computer-aided design and architecture for FPGAs.

Dr. Anderson was a recipient of the Ross Freeman Award for Technical Innovation, the highest innovation award given by Xilinx, for his contributions to the Xilinx placer technology in 2000. Since joining U of T as a Faculty Member, he has thrice received awards for excellence in undergraduate teaching, in 2009–2011. He serves on the technical program committees of various conferences, including the ACM International Symposium on Field Programmable Gate Arrays and the IEEE International Conference on Field Programmable Technology.