ACCELERATION OF AN FPGA ROUTER

Pak K. Chan and Martine D.F. Schlag

Department of Computer Engineering
University of California, Santa Cruz, California, 95064 USA

ABSTRACT

In this article we describe our experience and progress in accelerating an FPGA router. Placement and routing is undoubtly the most time-consuming process in automatic chip design or configuring programmable logic devices as reconfigurable computing elements. Our goal is to accelerate routing of FPGAs by 10 fold with a combination of workstation clusters *and* hardware acceleration. Coarse-grain parallelism is exploited by having several processors route separate groups of nets in parallel. A hardware accelerator is presented which exploits the fine-grain parallelism in routing individual nets.

1 INTRODUCTION

A serious problem in the use of Field-Programmable Gate Arrays as elements in the reconfigurable computing paradigm is the amount of time required to place and route FPGAs. Placement and routing of a single FPGA design may take hours for the denser devices even with high performance workstations. This problem is particularly serious for reconfigurable computing platforms that use multiple FPGAs. In this paper we examine methods for accelerating an FPGA router in a distributed computing environment of workstations which have attached FPGA coprocessors or multiple-FPGA boards.

Almost every major microprocessor improves its performance by approximately 4 times every 3 years. On the other hand, although custom computing is rapidly evolving, it is still in its infancy; FPGAs are not fast enough to compete with a microprocessor executing the same algorithm. A hybrid approach which can take advantage of advances in both microprocessor and FPGA technologies to accelerate the routing is indicated.

Communication latency between the workstations is on the order of milliseconds in a local area network environment. To achieve good speedup in accelerating an FPGA router with a cluster of uniprocessor workstations, the coarse-grain parallelism in the FPGA router must be exploited. In Section 3 we present a method which provides a 4X speedup with 10 processors. We anticipate that another 2X acceleration is achievable by implementing parts of the FPGA router in hardware as described in Section 4. Altogether, we can obtain 8X speedup using 10 uniprocessor workstations. With the advent of multiprocessor workstations which have lower communication latency, we would expect further speedups.

The primary activity of any router is the search for paths for signal nets within a *target graph* representing the routing resources. Hence, routing acceleration has focussed primarily on implementing at least some of the search in hardware. In maze routing, the target graph is a grid from which some vertices and edges have been removed to represent obstacles [1, 4, 3].

Previous approaches to routing acceleration have exploited the regularity of the grid and parallelism in the propagation of possible routes from a source [6, 7]. An exception is *Locusroute* which is a global router for standard cell [5]. *Locusroute* distributes the signal nets among several processors using a *shared memory* and sacrifices the router's performance in terms of routing tracks to allow multiprocessing. That is, the routing tracks required increased by roughly 10%.

Unlike standard cell and mask programmable gate array technologies, FPGAs have fixed and scarce routing resources. Although FPGAs often consist of logic and routing resources arranged in a grid at a high level, the routing resources are far from grid-like and may vary substantially in cost/delay (not unit cost).

2 An FPGA routing algorithm

We have been experimenting with a (serial) FPGA router *PathFinder*, which is applicable to various FPGA architectures. *PathFinder* is a negotiation-based router for FPGAs [2]. Figure 1 contains a sketch of the original *Pathfinder* algorithm used to find routes for the nets. In each iteration, *PathFinder* routes signals one at the time. In the first iteration, signals are allowed to share routing resources. In subsequent iterations signals must negotiate with other signals to determine which one needs the shared resource the most. A completely routed FPGA design is obtained when there are zero shared routing resources. The negotiation paradigm is based on the adjustment of costs attached to the routing resources to resolve congestion conflicts. In *Pathfinder*, the cost of a routing resource n is equal to

$$(c_n + h_n) \times p_n \qquad (1)$$

where c_n is the basic cost (usually the delay), h_n depends on the history of congestion of the resource,

```
Serial pathfinder(netlist, FPGA architecture)
    while there are shared routing resources
        for each net N_i
            RT ← source node of N_i
            while there are sinks of N_i which are not connected to the source
                P_j ← findpath(RT)
                add P_j to RT
            endwhile
            update cost of nodes based on congestion history
        endfor
    rip up nets if they have shared resources
    endwhile


findpath(RT)
begin
    for each node in RT
        enqueue n onto Q with key 0
    endfor
    while a new sink has not been found
        dequeue node, m, with lowest key from Q
        if m was not previously dequeued
            for each neighbor n of m
                enqueue n on Q with cost of n plus key of m
            endfor
    endwhile
    backtrace from the found sink till a node of RT is reached
        and return this path
end
```

Figure 1: The Negotiated Congestion Algorithm of Pathfinder.

```
Parallel pathfinder(netlist, FPGA architecture)
    while there are shared routing resources
        for each net N_i
            RT ← source node of N_i
            while there are sinks of N_i which are not connected to the source
                P_j ← findpath(RT)
                add P_j to RT
            endwhile
        endfor
    update cost of nodes based on congestion history
    rip up nets if they have shared routing resources
    endwhile
```

Figure 2: The Parallel Pathfinder.

and p_n reflects the current congestion, the number of signals currently sharing the resource.

In each iteration except the first, the cost of resources which are claimed by multiple nets are increased and nets which share nodes with other nets are ripped up and routed again. Notice that the node costs and congestions are updated within the inner loop of the algorithm. Figure 3 shows the execution profile of the serial *Pathfinder* algorithm routing a small design. The routine `findpath` performs a Dijkstra type search of the target graph to find a path from a partially routed net (RT) to its next closest sink. In this search a priority queue is maintained containing each node that has been encountered sorted by the cost of its path from RT. During each iteration, the execution time is proportional to the number of enqueue operations, and the number of dequeue operations is fairly constant. The congestion (number of shared resources) crests during the first several iterations and diminishes as the iterations progress. The algorithm terminates when there are no shared routing resources.

The routing algorithm in *Pathfinder* is particularly suited to FPGAs which have limited routing re-

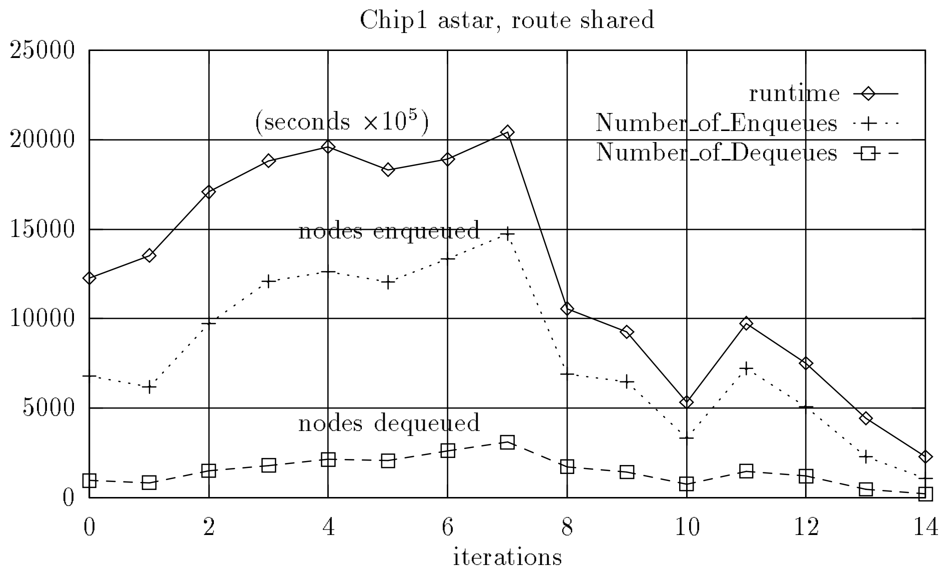

Figure 3: Execution profile of the serial *Pathfinder* routing algorithm.

router	Design	unrouted nets	routing time in minutes
Pathfinder	alu4	0	33
APR	alu4	15	41
Pathfinder	s1A	0	26
APR	s1A	5	28

Table 1: Comparisons between *Pathfinder* and Xilinx *apr* routers on the XC3090PC84 FPGA.

sources. Its authors have demonstrated the strength of this router by a direct comparison with the Xilinx router APR [2]. We reinforce this point by showing the results of both routers on two pathological designs for XC3090s in Table 1. In essence, *PathFinder* is able to complete two difficult-to-route designs while the vendor's router cannot. Table 1 also reveals that *PathFinder* is very demanding computation-wise. For a large design (e.g. Xilinx XC3090) typically *PathFinder* takes 40 to 100 iterations to converge (with the Astar search option). For a XC3090 design, each iteration takes about 30 seconds of CPU time on an Ultra-SPARC 143 MHz machine. So it is a worthwhile effort to speed it up. Since the node costs and congestion information are updated within the inner loop of the algorithm, *PathFinder* does not lend itself to a direct parallel implementation. In the next section we shall consider modifications to this algorithm to parallelize it.

3 Coarse-Grain Parallelization of the Pathfinder Algorithm

The original *PathFinder* algorithm is tightly coupled. We modified it as follows to obtain a loosely-coupled parallel routing algorithm that can be executed in a distributed memory environment. First we modified the node-cost updating scheme so that node costs are updated *outside* the inner loop of the algorithm, as shown in Figure 2.

In the first iteration, *Pathfinder* routes all of the nets without consideration of the resources used by other nets. In subsequent iterations the number of nets currently sharing a routing resource affects the cost of using that resource. The second modification allows *Pathfinder* to communicate changes in the congestion information (shared resources) with other *Pathfinder* processes using UNIX sockets. The nets are partitioned among processors and each processor runs this modified version of *Pathfinder* on its subset of the nets. Notice that in this parallel routing algorithm, there is no requirement to "lock" down the router's data structures; each processor keeps its own copy of the complete data structures for the entire FPGA. However, since the congestion information may no longer be current, a few more iterations are required than in the serial version. Figures 4 and 5 illustrate the speedup attained by using 10 workstations and 5 workstations respectively.

3.1 Convergence and load balancing of the parallel routing algorithm

The number of enqueue operations decreases as time progresses as seen earlier from Figure 3 which shows the execution profile of the serial *Pathfinder* routing algorithm. Therefore it is not beneficial to use a constant number of processors to route the signals. So, we reduce the number of processors available accordingly as iterations progress. Also, in our parallel implementation of *PathFinder*, node costs are updated *outside* the inner loop. Even though the congestion information is not as current in the serial version, the parallel *PathFinder* is still able to complete the routing of both difficult-to-route designs from Table 1. However, it generally takes a few

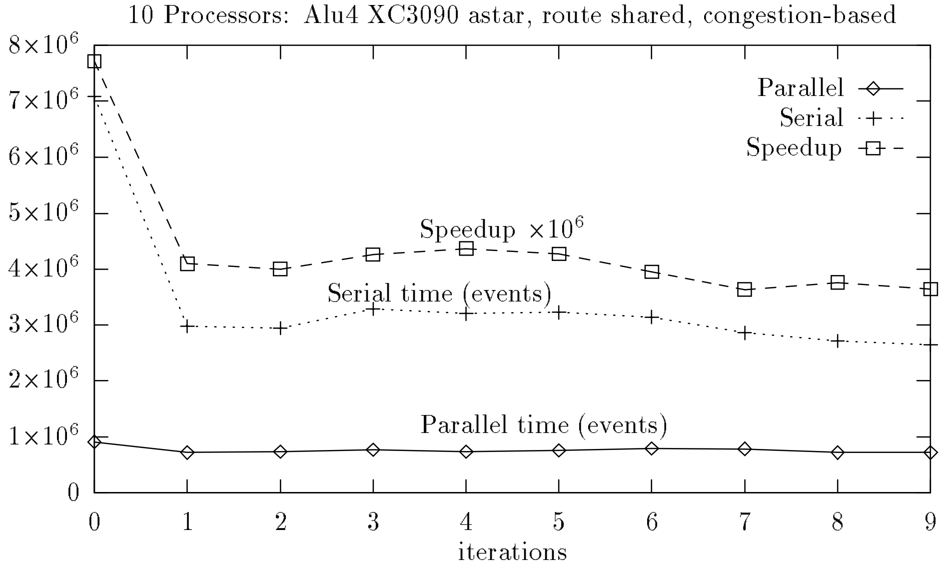

Figure 4: Speedup: using 10 processors in terms of the number of queue operations (events).

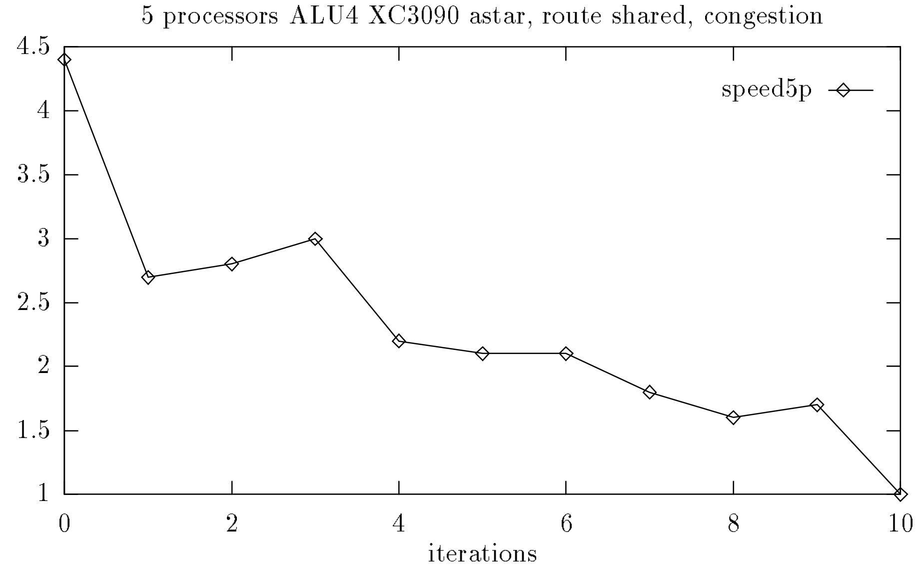

Figure 5: Speedup: using 5 processors.

more iterations for the parallel routing algorithm to converge. Again, it should be noted that execution time is not directly proportional to the number of iterations. The number of queue operations is the determining factor.

To optimize the speedup that can be attained from the parallel routing algorithm, the load must be balanced among the processors/workstations. Roughly, the processing time is proportional to the number shared resources left. As a preprocessing step, we sort the netlist in a lexicographical order, as illustrated in Figure 6. This results in signal nets that would potentially be sharing routing resources being adjacent to each other and hence they are very likely to be assigned to the same processor.

3.2 Global synchronization

The parallel router requires the changes in congestion information to be communicated as each signal's routing is completed. This communication overhead can be reduced with the addition of a smart network coprocessor to assist the microprocessor, as shown in Figure 7. This architecture facilitates the distributed updating of the node costs. In this environment,

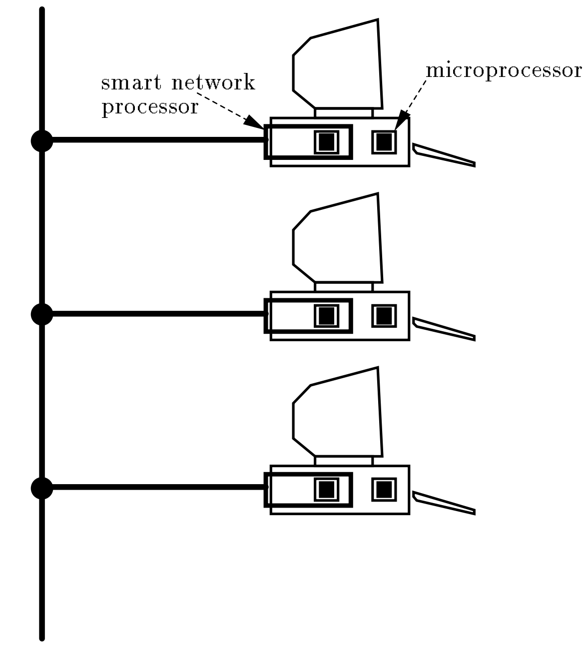

Figure 7: Basic architecture: using smart coprocessors to hide network latency.

	real version	simulated version	speedup
s1423	735.84	699.63	1.059
s386d	55.71	50.53	1.103
s1423 (with -astar)	43.63	39.6	1.102
s386d (with -astar)	18.36	15.25	1.204

Table 2: Time in seconds for the original version of *Pathfinder* and the modified version with pre-computed results of `dequeue` operations. The times are elapsed times for only the routing algorithm, not including reading inputs and writing results.

```
Addnet net1 AA.X AB.A BB.A
Addnet net2 BA.X AB.B BB.B
Addnet net3 CA.X AB.C BB.C

Addnet net4 AB.X P6.O
Addnet net5 BB.X P5.O

Addnet net6 P13.I AA.B BA.B CA.A
Addnet net7 P12.I AA.A BA.A CA.B
```

Figure 6: A lexicographically ordered netlist.

the coprocessor broadcasts the changes in resource congestion as soon as the signal's route is completed. Each coprocessor also intercepts this broadcasted information and computes and updates the costs and congestions of the nodes locally.

4 Fine-Grain Parallelism in a routing algorithm

We discuss the possibility of fine-grain parallelism in the router. Although the queue operations are the most numerous, they are themselves fairly efficiently implemented in software. In the context

	real version	simulated version	speedup
s1423	824.73	63.20	13.05
s386d	25.96	19.79	1.31
s382	175.03	10.9	16.06
s1423 (with -astar)	51.77	29.78	1.74
s386d (with -astar)	21.16	22.30	0.949
s382 (with -astar)	18.88	9.09	2.08

Table 3: Time in seconds for the original version of *Pathfinder* and the modified version with pre-computed results of `findpath` operations.

of configurable computing, an interesting question is how much speedup can be realized by eliminating the memory overhead required for handling the large number of queue operations? To estimate this speedup, *Pathfinder* was modified to record the results of all of the dequeue operations for two benchmarks. Another version was created to simulate the existence of a board implementing the queue operations. This version of *Pathfinder* first reads into memory all of the dequeue results and provides them when needed. The original queue operations were removed from the simulated version as well as the overhead in setting up the queue. Table 2 gives the running times of the "simulated" version versus the original code. As predicted from the profiling results the improvement was only in the 5 to 20% range, this

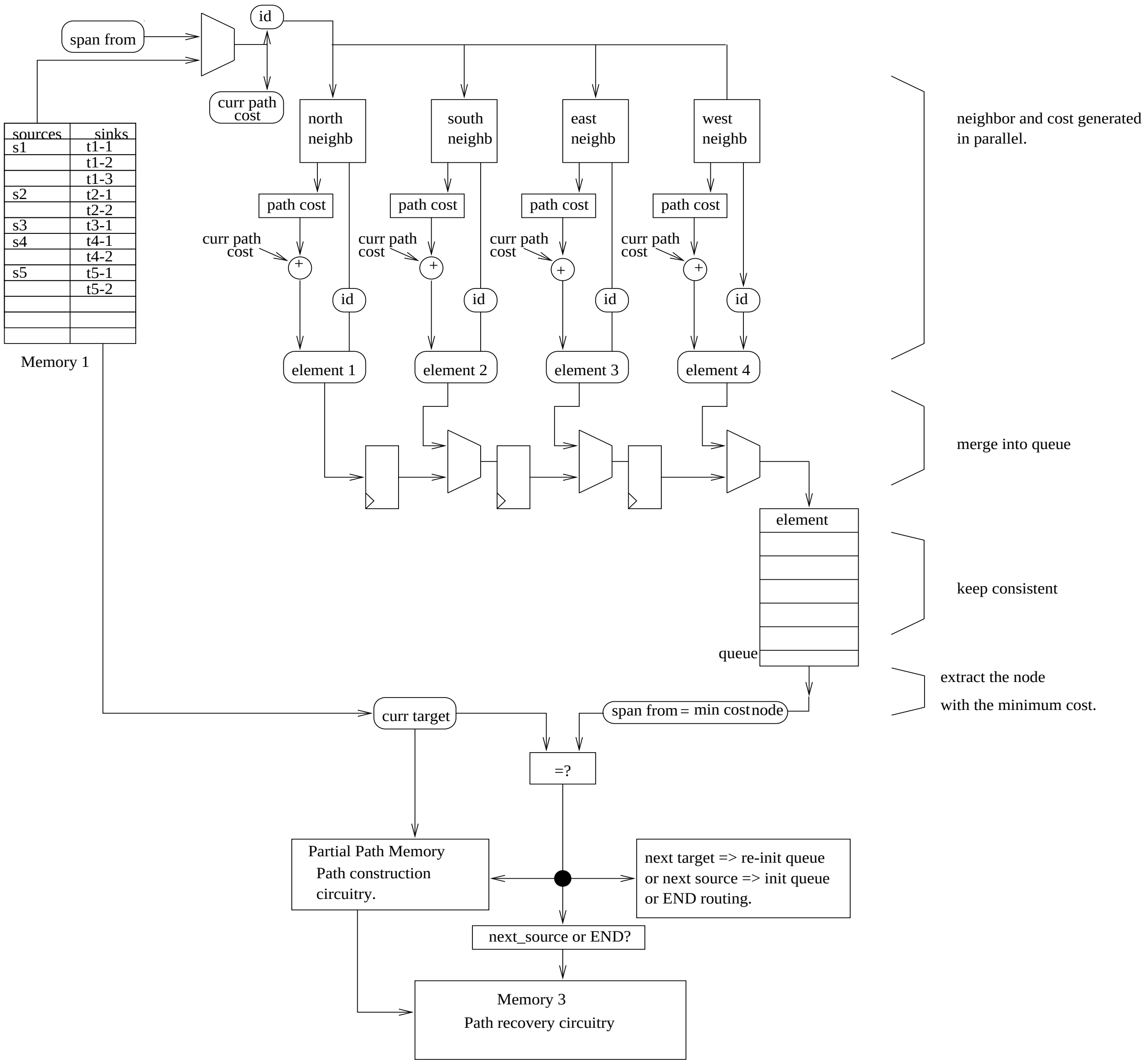

Figure 8: Basic architecture of a hardware router.

implies that it is not beneficial to implement *just* the priority queue in hardware.

However, from our experiments, it appears that the routing effort is primarily concentrated in expanding nodes and calculating costs. We next estimated in a similar manner the potential speedup if the results of `findpath` were known. In this experiment, the simulated version of `findpath` reads results from disk and not from memory, so there is some overhead in retrieving the previously recorded result. Since UNIX I/O is buffered, the actual I/O time is less than the disk access latency. Our measurement shows that each I/O is approximately 0.15 ms. The results are reported in Table 3. The potential speedup can be dramatic as observed for **s1423** and **s382**. Note that in the case of **s386d_astar**, the modified version was actually slower, probably due to disk accesses.

Unfortunately, implementing the entire search for a path in hardware is not as straightforward as in the case of maze routing, since

- the target graph is not simply a subgraph of a grid and,
- the costs of the routing resources vary to reflect delay and congestion penalties.

Fortunately, there is still some regularity in the FPGA target graph and the costs of the resources (nodes) do not vary within an iteration. We propose to explore a hardware accelerator, in which the hardware will perform the node expansion in parallel, and the costs of the nodes will be stored in on-board memories. As shown in Figure 8, the hardware accelerator has several components: a priority queue, a neighbor generator, path reconstruction circuitry, and some local memories. The local memories will be updated after each iteration by a host processor.

The priority queue is a systolic array that sorts.

Given a node (routing resource), the "neighbor generator" returns the node addresses of its neighbors. The "regularity" of an FPGA resource graph enables a fairly efficient implementation of the neighbor generator in hardware. Not only is this more efficient in hardware, but also by having several "neighbor generators" operate in parallel, we can expand the multiple neighbors of a node in parallel. The cost of the neighbors are obtained from the local memories. These neighbors are then enqueued with their respective keys.

In detail, each neighbor generator has a table with the cost of each node in the circuit, and the cost tables are all identical. Each table stores the cost of all the nodes (routing resources) in the FPGA architecture. Note that the cost tables are not drawn in Figure 8. The host computer initializes memory bank 1 with signals and target IDs, and initializes the cost tables with the cost of each routing resource. During each routing iteration, a signal is selected from memory 1 and routed. Each neighbor generator receives the current routing resource used by the signal and produces the IDs and intrinsic cost of its neighboring node. The path cost of each neighbor is accumulated. Since the neighbors can be generated in parallel, their associated path costs can be computed in parallel. The costs are inserted into the priority queue. The minimum path cost is extracted from the priority queue, and the associate node is chosen as the next routing resource of the signal. The ID of this node is maintained in the partial path memory to enable the host to reconstruct the routing paths.

The logic design of the priority queue has been implemented and tested using the Xilinx XC4013E family FPGAs. An efficient implementation of the priority queue in hardware which allows multiple simultaneous enqueue operations is still under investigation.

5 Summary

In summary, we have investigated the acceleration of an FPGA router by parallelizing it both at the coarse-grain and the fine-grain levels.

- We restructure the serial Pathfinder routing algorithm to a parallel routing algorithm so that it can be executed by a cluster of uniprocessor workstations. Our parallel routing algorithm is operational; it allows signal nets to be routed independently.

- We have proposed an implementation of a hardware accelerator which performs the search for a route for a given net by expanding neighbors in parallel. Although this part is not completed, it has been throughoutly investigated and implementation is underway.

6 Acknowledgments

The authors have benefited from the Pathfinder C code written by Larry McMurchie and Carl Ebeling. The authors acknowledge the support from Xilinx Corporation and the University of California, MICRO research program.

References

[1] C. Y. Lee. An Algorithm for Path Connections and Its Applications. *IRE Transactions on Electronic Computers*, pages 346–365, Sept. 1961.

[2] L. McMurchie and C. Ebeling. Pathfinder: a negotiation-based performance-driven router for FPGAs. In *Proceedings of 3^{rd} International ACM/SIGDA Symposium on Field-Programmable Gate Arrays*, pages 111–117, Monterey, California, USA, Feb. 1995.

[3] K. Mikami and K. Tabuchi. A computer program for optimal routing of printed circuit board connections. In *IFIPS*, pages 1475–1478, 1968.

[4] M. Palczewski. Plane parallel A$*$ maze router and its application to FPGAs. In *ACM IEEE 29^{th} Design Automation Conference Proceedings*, pages 691–697, Anaheim, California, June 1992.

[5] J. Rose. Parallel global routing for standard cells. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 9(10):1085–1095, Oct. 1990.

[6] R. A. Rutenbar and D. E. Atkins. Systolic routing hardware: Performance evaluation and optimization processor. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 7(3):397–409, Mar. 1988.

[7] T. Watanabe, H. Kitazawa, and Y. Sugiyama. A parallel adaptable routing algorithm and its implementation on a two-dimensional array processor. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 6(2):241–250, Mar. 1987.