Distributed-Memory Parallel Routing for Field-Programmable Gate Arrays

Pak K. Chan, Martine D.F. Schlag
Department of Computer Engineering
University of California, Santa Cruz, California 95064 USA

Carl Ebeling and Larry McMurchie
Department of Computer Science Engineering
University of Washington, Seattle, Washington 98195 USA

Abstract

The problems of placement and routing are without doubt the most time-consuming part of the process of automatically synthesizing and configuring circuits for field-programmable gate arrays (FPGAs). FPGAs offer the ability to quickly reconfigure circuits to support rapid prototyping, emulation or configurable computing, but the time to perform placement and routing, which can take many hours, has become a serious bottleneck. This problem is addressed here by showing that the negotiation-based routing paradigm, which has been applied successfully in several FPGA routers, can be parallelized to achieve increased performance without any significant decrease in the quality of the results. In this paper, we report several new findings related to the negotiation-based routing paradigm. We examine in-depth the convergence of the negotiation-based routing algorithm. We illustrate that the negotiation-based algorithm can be parallelized. Finally, we demonstrate that a negotiation-based parallel FPGA router can perform well in terms of delay and speedup with practical FPGA circuits.

1 Introduction

The negotiation-based routing paradigm has been used successfully in a number of FPGA routers. This paradigm was originally developed by McMurchie and Ebeling for architecture-customizable FPGA routing [1], and has been applied to the routing of a number of FPGA architectures. Accelerations of this paradigm include using an aggressive A* search [2, 3], and its parallelization, reported in [4]. Despite these advances there are still unsettled questions about the convergence of the negotiation-based routing method and it's performance in terms of both routing success and delay optimization on real silicon. In this paper, we report several new findings related to the negotiation-based routing paradigm. First, we provide some proofs and counter-example of the convergence of the negotiation-based algorithm. Second, we demonstrate that the negotiation-based algorithm can be parallelized. Third, we demonstrate that a negotiation-based parallel FPGA router can perform well with practical FPGA circuits. For simplicity, we refer to the negotiation-based router as NC (or *Pathfinder* as in [1]).

This paper is organized as follows. We briefly review the negotiation-based router NC in Section 2. In Section 3, we outline the negotiation paradigm and model it as a conflict resolution method. Convergence properties of the paradigm are studied, and results are used to guide the development of a parallel negotiation-based routing algorithm in Section 4. The parallel routing algorithm has been implemented to route Xilinx 4000 FPGA designs with good results. Validation of the parallel implementation is established in Section 5 through a direct comparison with a commercial FPGA router. Finally, future directions are discussed in Section 5.4.

2 NC: a negotiation-based router

The architecture of the FPGA is represented by an architecture graph, $G_A = (N, E)$ which is a directed graph. A node $n \in N$ represents a routing resource such as a wire or terminal, and an edge $e \in E$ represents a switch or feasible connection between two nodes. Each signal consists of one node of G_A, the signal's source, along with a subset of nodes corresponding to its sinks.

Routing a signal involves assigning routing resources such that all the sinks are reachable from the source. When routing a set of signals sequentially, the order in which the signals are routed may be critical since some routing resources needed by a signal may be occupied by signals that are routed earlier. The principal idea of the negotiation paradigm (see NC pseudo-code in Figure 1) is to permit unlimited sharing of routing resources initially and then repeatedly reroute signals until no resources are shared. By assigning congestion costs to the shared routing resources, and increasing these costs with each iteration through the signals, the routing algorithm encourages alternative routes to be explored until all the conflicts are resolved.

The cost of a routing resource n has three terms

$$c_n = (d_n + h_n) \times (p_n + 1) \tag{1}$$

The base cost is d_n which can be used to reflect the delay of the routing resource. The 1st-order congestion term p_n is the number of signals that are currently sharing the routing resource. The 2nd-order congestion (history) term h_n grows monotonically with each iteration in which the routing resource is shared. (In fact h_n increases by a fixed amount each time a signal is rerouted through an already occupied node.)

Minimizing the critical path in the Pathfinder algorithm is achieved by creating a two-term cost function. The first term is the congestion cost for a node. The second term is the delay through a node. Coefficients on each term are related to the criticality of the signal being routed. If the signal belongs to a path that has almost as much delay as the critical path, the delay term is more important. Otherwise the congestion term is dominant.

To introduce delay into algorithm NC, we redefine the cost of using node n when routing a signal from n_s to t_{ij} as

$$C_n = A_{ij} d_n + (1 - A_{ij}) c_n \tag{2}$$

where c_n is defined in Equation (1) and A_{ij} is the slack ratio

$$A_{ij} = D_{ij} / D_{max} \tag{3}$$

where D_{ij} is the delay of the longest delay path containing the edge (n_-, t_{ij}) and D_{max} is the maximum delay over all paths, i.e. the critical path delay. Thus $0 < A_{ij} < 1$.

The first term of Equation (2) is the delay-sensitive term, while the second term is congestion-based. Equations (2) and (3) are the keys to providing the appropriate mix of minimum-cost and minimum-delay trees. If a particular source-sink pair lies on the critical path, then $A_{ij} = 1$ and the cost it sees for using node n is simply the delay term; hence a minimum-delay route will be used and congestion will be ignored. If a source-sink pair belongs to a path whose delay is much smaller than the critical path, its A_{ij} will be small and the congestion term will dominate, resulting in a route which avoids congestion at the expense of extra delay.

To accommodate delay, the NC algorithm is changed as follows. First the A_{ij} are initialized to 1. Thus during the first iteration the global router finds the minimum-delay route for every signal. The A_{ij} are recomputed each subsequent iteration. Second, the sinks are routed in decreasing A_{ij} order. Third, the priority queue Q is initialized to RT_i at cost $A_{ij}d_j$. The net effect of this initialization is that nodes that are already in the partial routing tree will have only a delay component. We shall refer to these modifications as the NCD algorithm, or simply *PathFinder*, in the remainder of this paper.

The global router completes when no more shared resources exist. Note that by recalculating the A_{ij}, we have kept a tight reign on the critical path. Over the course of iterations, the critical path increases only to the extent required to resolve congestion. This approach is fundamentally different from other schemes such as [5] and [6] which attempt to resolve congestion first, then reduce delay by rerouting critical nets.

Further, in our previous work [7] we have shown that if h_n is bounded by d_n, then algorithm NCD guarantees a worst case path delay equal to the minimum delay route of the critical path. That is, in this situation algorithm NCD achieves the fastest implementation allowed by the placement. In practice, h_n is allowed to increase gradually until a complete route is found. For very congested circuits, h_n will exceed d_n and performance will suffer.

We study the convergence properties of the negotiation paradigm, for the purpose of developing and parameterizing a parallel FPGA router.

3 Correctness of the Negotiation Paradigm

The negotiation paradigm used in NC is modeled as a conflict resolution method for bipartite matching. A routing problem is modeled by a graph or hypergraph matching problem. There are two sets of vertices in this graph,

$$G_M = (X \cup Y, E)$$

and the edges (or hyperedges) connect one vertex in X with one (or more) vertices in Y. Here X represents the signals to be routed and Y is the set of routing resources. The edges of a vertex $x \in X$ represent the subsets of resources that correspond to routing trees for the signal associated with x. A solution is a subset of $|X|$ edges, exactly one edge for each $x \in X$, such that for each $y \in Y$ no more than one of these edges includes y. The former condition ensures that each signal is assigned a routing tree, while the latter condition ensures that there are no shared routing resources (conflicts).

To model a routing problem as a hypergraph matching problem, a hyperedge should be generated for each possible routing tree of a signal. The number of different paths (or routing trees) for a signal can easily be exponential (in the size of the architecture graph). Fortunately, the negotiation method does not require the explicit enumeration of all hyperedges; it is sufficient only to be able to identify the cheapest hyperedge (subset of routing resources) for a given signal. In NC this amounts to a shortest-path calculation.

Figure 2 gives an example of a routing problem and its associated hypergraph matching problem. There are three possible routes for signal S_1, $\{A, B, C, D\}$, $\{E, F, G, H\}$ and $\{A, B, G, H\}$ in the architecture graph in Figure 2(a) and these correspond to the three hyperedges of node S_1 in the graph in Figure 2(b).[1]

The routing example in Figure 2(a) yields the hypergraph matching problem in Figure 2(b) which the negotiation-based method (NC) does not solve.[2] This example is a simplified version of a classic routing problem which cannot be solved by a simple obstacle avoidance method for any ordering of the nets. The assignments and the costs for each iteration are presented in Table I. The top row gives the iteration number and the hyperedges selected in that iteration for (S_1, S_2, S_3) are given in the second row. In the remaining rows, the costs at the start of the iteration and the resulting assignments are given. Note that the same hyperedges are selected in iterations 1 and 11.

In spite of this result for hypergraphs, it is still useful to examine the restricted case of graphs in order to determine which variants of the negotiation-based method are more effective. In practice we do not expect nor have we encountered cases in which the interactions between the signals are as complicated as in Figure 2. The hypergraph matching problem is a more general version of the 3-dimensional matching problem and hence it is NP-complete. In the remainder of this section we prove that the negotiation-based method does converge on graphs under different uses of the first-order and second-order congestion and with or without delaying the second-order congestion updates to the end of the iterations. There are clearly better algorithms for solving the bipartite graph matching problem (in polynomial time), However these methods are not practical for our routing application since they require the enumeration of all edges.

Given a bipartite graph $G_M = (X \cup Y, E)$ with $|X| \leq |Y|$, we wish to find a matching which assigns an edge to each vertex in X. The pseudo-code for the negotiation-based algorithm **NCG** is given in Figure 3. The algorithm constructs an assignment $a : X \rightarrow Y$ of X vertices to Y vertices. The number of X vertices currently assigned to a Y vertex is given by $n(y)$ which captures the first-order congestion information. When $n(y) > 1$, y is conflicted and when $n(y) = 0$, y is open. A solution has been found when $a()$ is 1-1; all y have $n(y) \leq 1$. An initial assignment is given and each vertex y in Y has a base cost $cost(y)$ which is incremented in between iterations if y is conflicted at that point. In NC this cost would initially be d_y and as the algorithm progresses it would be $d_y + h_y$. Here $n(y)$ represents the first-order congestion information while $cost(y)$ represents the

[1] Only minimal routes are represented. For example, the route $\{A, B, C, D\}$ for signal S_2 is not represented since $\{A, B\}$ is also a route for S_2.

[2] A more complicated version of this problem will yield a hypergraph which cannot be solved regardless of the use of immediate history cost update with any combination of the parameters considered in the section.

second order. In this version the costs are only updated in between iterations. The effects of immediately updating the costs after an assignment is made, are discussed in Section 3.3.

During each iteration (lines 3–9 in Figure 3) **NCG** uses one of two routines **SelectCheapest** or **SelectOpen** to determine which Y vertex to assign to a given vertex in X. If Boolean parameter *ReassignAll* is **FALSE**, vertices of X assigned to currently unconflicted Y vertices will be skipped, not reassigned. When *ReassignAll* is **FALSE** this corresponds to the **RouteSharedOnly** option of NC where only those signals that currently share nodes with other signals are ripped up and rerouted. We assume that the vertices of X are ordered $X = \{x_1, x_2, \ldots, x_{|X|}\}$.

The two routines in Figure 4 which determine the assignment for a given X vertex do not use Equation (1) to select a Y vertex. To facilitate the proofs the cost function for assigning vertices has been simplified. However, the only important factors in determining convergence are whether the first-order and second-order congestion information are used (either in deciding to reroute signals or selecting routes) and that the cost of resources that are conflicted keep increasing with each iteration.

SelectOpen will always select an open vertex ($n(y) = 0$) over a used vertex ($n(y) \geq 1$) and uses $cost(y)$ to choose among the open vertices or occupied vertices (if there are no open ones). **SelectCheapest** selects a Y vertex only on the basis of $cost(y)$. Thus **SelectCheapest** is not using the first-order congestion information. In either case, it is assumed that ties are broken consistently. That is, there is an underlying order on the neighbors of each vertex used to break ties. In our discussion, two Y vertices adjacent to an X vertex never have the same cost since the tie breaking order determines the cheapest.

If **NCG** terminates, then the final mapping $a()$ is a solution. **NCG** is said to converge if it always terminates when a solution exists. Table II summarizes the results in this paper on the convergence behavior of **NCG** on graphs. Its convergence depends on the selection routine and the use of *ReassignAll*.

It is not difficult to construct an example in which **NCG** with *ReassignAll* and **SelectCheapest** does not terminate even though a solution exists. Figure 5 illustrates a simple example. The problem is that vertices a and b are both assigned based on information only from the previous iteration. They have no information about the current location of the other vertex. If **SelectOpen** were used instead of **SelectCheapest** one of the two vertices (whichever is processed last) would remain at it's current location because it is open. We shall see that the use of **SelectOpen** in place of **SelectCheapest** is enough to make **NCG** with *ReassignAll* = **TRUE** converge.

In **NCG** the vertices of X are ordered and this is the order in which the vertices of X are examined in each iteration. Although the order of the X vertices can be arbitrary, for the algorithm to converge, it is critical that the X vertices to be examined in the same order in all iterations. Lemma 2 will make use of the property that the vertices in X are always processed in the same order in the main **for-**loop of **NCG**. The example in Figure 6 shows that varying the order can keep **NCG** from converging even though there is a solution. In this example the vertices of X are processed in the order a, b, c in the first and then in every even numbered iteration while the order c, b, a is used in the other iterations. (The position of c in the order is irrelevant since it has only one edge.) As a result a is never moved because it is always processed before a conflict occurs at e and in the next iteration the conflict at e is removed because b is processed before a. So although the cost of vertex e increases, a will never be assigned to vertex d which is necessary in the solution.

To show that **NCG** converges we show that no solution exists if it does not terminate.

Definition 1 A vertex in Y currently with $n(y) = 0$ is said to be *open* and otherwise *used*. A vertex in Y currently with $n(y) > 1$ is said to be *conflicted*.

During the course of the algorithm a vertex can change its status (open, used and/or conflicted). The next two definitions are properties of vertices which do not vary.

Definition 2 A vertex of Y is said to be *profoundly conflicted* if it is conflicted at the end of an infinite number of iterations.

Since all conflicted vertices (vertices with $n(y) > 1$) at the end of an iteration have their costs increased, the cost of a profoundly conflicted vertex is unbounded.

Definition 3 A vertex $y \in Y$ is said to be *stable* if after some finite number of iterations $a^{-1}(y)$ remains constant and $n(y) \leq 1$. That is, either no X vertex is ever again assigned to y or the vertex x is assigned to y from that point on and no other vertex is ever assigned to y from that point on; y becomes *fixed*. Otherwise the vertex is said to be *unstable*.

By definition if **NCG** terminates then no vertex is profoundly conflicted and all vertices are stable. So these two definitions are only useful when **NCG** does not terminate. Clearly a stable vertex is not profoundly conflicted although it might be conflicted a finite number of times before its assignment becomes fixed. It is not immediate that an unstable vertex is profoundly conflicted since a vertex could temporarily have $n(y) > 1$ during an iteration but be back to $n(y) = 1$ by the end of that iteration. However, we shall show that an unstable vertex must always be conflicted again at the end of some later iteration. We let C be the set of all profoundly conflicted vertices and S be the set of all stable vertices.

We prove that **NCG** converges with *ReassignAll* = **FALSE** in Section 3.1 and that it converges with *ReassignAll* = **TRUE** using **SelectOpen** in Section 3.2.

3.1 Convergence on Graphs without ReassignAll

In **NCG** with *ReassignAll* = **FALSE**, a vertex y never changes its status from used to open. This is because only vertices with $n(y) > 1$ can lose their assignees and only one at a time. Since used vertices can never become open, if the algorithm does not terminate, the number of open vertices stops decreasing at some point. The open vertices at this point become fixed and hence are stable. The first step in the convergence proof is to show that all Y vertices are either stable or profoundly conflicted.

Lemma 1 In **NCG** with $ReassignAll$ = **FALSE** using either **SelectCheapest** or **SelectOpen**, if a vertex $y \in Y$ is not profoundly conflicted, then it is stable.

Proof: By contradiction. Assume that there are unstable vertices which are not profoundly conflicted: $U = Y - (C \cup S)$ is not empty. Let t be an upper bound on the last iteration after which any vertex in U is never conflicted at the end of an iteration, and all vertices in S have become fixed. Since vertices in C have costs which are unbounded while vertices in U have fixed costs after iteration t, there is an iteration t' after which the cost of any vertex in C will exceed the cost of any vertex in $Y - C$. At this point the algorithm will never reassign a vertex x with $a(x) = y \in U$ to a vertex of C since C vertices are never open after iteration t and their costs will always exceed the costs of U vertices. After iteration t no more than one vertex can be reassigned to a given $y \in U$ within an iteration, since otherwise y would remain conflicted at the end of the iteration. So if within an iteration x_1 is assigned to vertex $y_1 \in U$, then later in the same iteration the vertex previously assigned to y_1 must be moved to another $y_2 \in U$ and the vertex previously assigned to y_2 must be moved to $y_3 \in U$ and so on. Eventually this chain must end, and it can only end if a second x is assigned to some vertex in $y_k \in U$. The other vertex assigned to y_k at the start of the iteration will not be reassigned. However, this means that y_k will remain conflicted at the end of the iteration which contradicts the assumption that no vertex in U will have such a conflict after iteration t. Hence it is not possible for unstable vertices not to be profoundly conflicted. $\square$

By Lemma 1 all Y vertices are either profoundly conflicted or stable. Let T be the iteration after which all stable vertices become fixed (no longer change their assignment) and the number of open vertices no longer decreases. Let $a_t(x)$ denote the vertex assigned to x at the end of iteration t. Then for stable vertices y we know that

$$\text{for } t \geq T,\ a_t^{-1}(y) = a_T^{-1}(y).$$

Let $X_S = \{x | x \in a_T^{-1}(y) \text{ for some } y \in S\}$, the set of vertices (permanently) assigned to S vertices after iteration T. Let $X_C = X - X_S$ be the other vertices of X, the profoundly conflicted vertices. No vertex in X_C is ever assigned to a vertex in S after iteration T, so they must all be assigned to vertices in C by Lemma 1.

Lemma 2 If **NCG** with $ReassignAll$ = **FALSE** using either **SelectCheapest** or **SelectOpen** does not terminate, then there is no edge between any $x \in X_C$ and $y \in S$.

Proof: By contradiction. Suppose there is an edge between $x \in X_C$ and $y \in S$. Since $x \in X_C$, it is always assigned to a vertex in C after iteration T. Some number of iterations after T, say at iteration T' the cost of any vertex in C will exceed the cost of any vertex in S because the cost of vertices in C are unbounded while the costs of vertices in S are fixed after iteration T. Suppose $a(x) = y \in C$ at iteration T'. Since y is in C at some point another vertex x' will be reassigned to y. Let x' be the first such vertex. x is still assigned to y at this point since the algorithm will not move x until $n(y) > 1$. There are two cases to consider depending on the ordering of x and x'.
Case 1. x is first. Then y will still have both x and x' assigned to it at the start of the next iteration. Since x is processed first, x' will still be assigned to y and hence $n(y) > 1$ when x is

processed. Since x has an edge to a vertex in S and all vertices in S have smaller cost than C vertices, x will be reassigned to a vertex in S.
Case 2. x' is first. In this case x' will be moved to y and later in the same iteration x will be processed. Since $n(y) > 1$ when x is processed, x will be reassigned to a vertex in S.

Since both cases lead to the reassignment of x to a vertex in S after iteration T which is a contradiction. □

Theorem 1 If **NCG** with *ReassignAll* = **FALSE** using either **SelectCheapest** or **SelectOpen** does not terminate, then the bipartite graph $(X \cup Y, E)$ has no matching which contains an edge for each $x \in X$.

Proof: By Lemma 2 there is no edge from X_C to S. This means that all of the vertices of Y incident on vertices of X_C are in C: $N(X_C) \subseteq C$. After iteration T a profoundly conflicted vertex is never open. All vertices in X_C are assigned to C and there is always one vertex of C with two or more vertices of X_C assigned to it. It follows that $|X_C| > |C|$. Then $|X_C| > |C| \geq |N(X_C)|$ which means that the bipartite graph $(X \cup Y, E)$ has no matching with an edge for each vertex in X_C and hence X. □

3.2 Convergence on Graphs with ReassignAll

In **NCG** with *ReassignAll* = **FALSE**, a vertex y never changes its status from used to open is because only vertices with $n(y) = 1$ are reassigned. This is still the case when *ReassignAll* = **TRUE** (all vertices of X are processed) but it now requires a proof and an assumption on how vertices are initially assigned.

Definition 4 An assignment $a() : X \to Y$ is *minimal* if for each vertex $x \in X$ none of the open neighbors of x are cheaper than $a(x)$.

An initial minimal assignment can be constructed by assigning each X vertex to its cheapest neighbor (regardless of its occupancy). This is what happens in *Pathfinder*.

Lemma 3 If **NCG** with *ReassignAll* = **TRUE** using **SelectOpen** starts with a minimal assignment then no vertex in Y ever changes its status from used to open.

Proof: By contradiction. Suppose there is a vertex $y \in Y$ which changes its status from used to open at some point during the course of the algorithm. Let y be the first vertex to become open. This means that in some iteration t we have $a(x) = y$ and $n(y) = 1$ when vertex x is processed and after x is processed $a(x) = y'$ and $n(y) = 0$. In order for x to switch from y to y' we must have $n(y') = 0$ before x is processed and $cost(y') < cost(y)$. Since y is the first vertex to become open, y' must have been open in the initial assignment and remained open up till this point.

Case 1: x switched from y to y' in the first iteration ($t = 1$). In order for the switch to occur in the first iteration, the base cost function would have to satisfy $cost(y) > cost(y')$. However this contradicts the assumption that the initial assignment is minimal since y' is open in the initial assignment.
Case 2: x switched from y to y' in the second or later iteration ($t > 1$). Since x was assigned to y rather than y' in iteration $t-1$ it must have been the case that $cost(y) < cost(y')$ in iteration $t-1$. Since $cost(y) > cost(y')$ in iteration t, the cost of y must have increased at the end of iteration $t-1$ which means that y was conflicted at the end of iteration $t-1$. We know x was one of the vertices assigned to y at the end of iteration $t-1$. Let x' be any other vertex assigned to y at the end of iteration $t-1$. Since x is the last one to be removed from y in iteration t, x' must precede x in the processing order. However, then x' was already assigned to y in iteration $t-1$ at the time x was processed. Under **SelectOpen** x is assigned to y rather than to y' only if y' was not open in iteration $t-1$. This contradicts the fact that y' has been open since the start of the algorithm.

Since either case leads to a contradiction, it follows that no vertex can become open if **NCG** uses **SelectOpen** and starts with a minimal assignment. □

After establishing that vertices do not become open, the convergence of **NCG** can be shown in exactly the same manner for $ReassignAll$ = **TRUE** as for $ReassignAll$ = **FALSE**. Lemma 2 and Theorem 1 hold.

3.3 Immediate Update of History Costs

The negotiated-congestion based method employed by NC updates the history costs of the signals after each signal is routed rather than waiting until the end of the iteration. As a result the cost of a Y vertex increases by the number of signals assigned to that vertex within an iteration minus 1, rather than just by 1 if the vertex remains conflicted. Not only do the second-order congestion (history) costs rise more rapidly in proportion to the demand for a Y vertex, but also the cost changes affect other assignments within the same iteration. Experimentally immediate update of the history costs appears to decrease the number of iterations required for finding a solution.

In showing that using immediate update of the history costs still results in convergence, we modify the definition of profoundly conflicted to include vertices which are conflicted as a result of an infinite number of assignments (rather than only at iteration boundaries). These are now the vertices that have unbounded costs during the course of the algorithm. In this case it is much easier to see that all unstable vertices are profoundly conflicted. A vertex $y \in Y$ cannot change it's assignment without either becoming open (impossible with $ReassignAll$ = **FALSE** or by Lemma 3 for $ReassignAll$ = **TRUE**) or becoming conflicted. The proofs for Lemma 2 and Theorem 1 can then be adapted to cover this case.

3.4 Implications for NC

Although the example in Figure 2 demonstrates that NC will not always converge, the convergence properties of NCG on graphs identify the following factors which affect NC's convergence.

1. The signals should be examined in a fixed order in each iteration.
2. The second-order congestion costs may either be updated immediately after a signal is rerouted or only at the end of the iteration.
3. The second-order congestion costs updates may be proportional to the number of signals occupying a node, or simply constant.
4. The first-order congestion information must be updated within an iteration and may be used either to select assignments, determine which signals should be rerouted, or both.

4 Parallel Pathfinder: a timing-driven parallel FPGA router

There have been many attempts to parallelize routers. The book of Banerjee [8] gives an excellent survey of parallel algorithms for detailed and global routers for VLSI IC layouts. Representative works are [9, 10, 11]. Some algorithms are specifically designed for shared memory multiprocessors, and some must be executed on distributed memory MIMD machines. The development of our parallel FPGA router is motivated by the growing place-and-route demand in high-capacity field-programmable gate arrays and the increasing availability of clusters of workstations. An FPGA user has the burden of performing placement and routing in the field. A parallel FPGA router that performs under a low-cost ubiquitous distributed-memory multiprocessing environment will find wider application than a router which requires more specialized hardware. The parallel FPGA router should also be scalable and the quality of the result should not degrade considerably with the number of processors. We discuss some important issues related to a parallel routing algorithm in the following sections. Namely,

1. What makes an FPGA router parallelizable, and what is the granularity of parallelism?
2. What is the processor interconnection architecture, memory model and synchronization method?
3. How would the parallel router scale in terms of speedup and quality of the results?
4. Would the parallel router converge?

The first question is determined by whether functional or data parallelism (or both) will be exploited in the router. The kernel of any router employs some form of shortest-path routing algorithm. A functional-parallel approach to parallelizing a router would require the decomposition of the kernel of the router into concurrent fine-grain tasks. A data-parallel approach to parallelizing a router would involve decomposing the design to be routed into subsets. A typical large FPGA design has more than thousands of signals, so routing could be trivially parallelized if separate signals could be routed in parallel. If this is feasible, the degree of parallelism in principle will be limited only by the number of signals. The difficulty is that the route taken by each signal depends on knowledge of the other signals' routes since the routing resources cannot be shared. Routing signals in parallel was the approach used in *Locusroute*, a global router for standard cell [12]. *Locusroute* distributes the signal nets among several processors using a shared memory where mutual exclusion is maintained. It sacrifices the router's performance in terms of routing tracks to allow multiprocessing. In global routing the exact locations of the routes are not determined; the goal is only to specify the regions between obstacles (channels) through which a signal will be routed; the exact locations of routes are determined afterwards by a detailed router. However, in FPGAs the routing resources are discrete and fixed. Channels cannot be expanded to accommodate additional wires and having a sufficient number of tracks does not guarantee that a channel can be routed because of the nature of the connections between tracks and logic block terminals.

Since the routing of separate signals is tightly coupled in an FPGA, it might appear that a net-parallel (data-parallel) approach to routing FPGAs would not be possible since knowledge of other signal locations is necessary to find a feasible route. This is the case in a typical maze router which uses rip-up and reroute to resolve conflicts. However in *PathFinder*, there is no restriction on the number of signals which can occupy a resource simultaneously during routing, instead the cost of using a congested resource is the mechanism used to resolve resource conflicts. In the next section, we exploit this built-in conflict-resolving mechanism to decouple the routing of separate signals to achieve a parallelization of the PathFinder algorithm.

4.1 Parallelizing the Pathfinder Algorithm

In the serial Pathfinder (NC) routing algorithm, the interaction among different signals is captured by the first and second-order congestion costs of the routing resources that the signals are sharing. Given that we are assuming a distributed-memory multiprocessing environment, the congestion costs must be decentralized. There will be a certain degree of uncertainty regarding the value of the congestion costs. In order to reduce the uncertainty, the foremost concerns in parallelizing the algorithm are *how* and *when* will the *updating* of the congestion costs take place.

The results in Section 3 serve as important guidelines. First, the signals must be examined in a fixed order. Second, the updating of second-order congestion cost is not crucial, there is some flexibility with respect to when the updating will occur. Third, the first-order congestion costs must eventually be accurate.

Imagine a parallel routing scheme in which each processor is assigned a disjoint subset of signals to route concurrently. Each processor has a private local memory, and each processor is connected to a network. The routing algorithm for a processor is given in Figure 7. To route a signal, a processor would require knowledge of congestion costs of the routing resources that the route is attempting. The first-order congestion cost of a routing resource can be calculated easily if the number of signals sharing the resource is known.

So we take the approach that each processor tallies the first and second-order congestion costs of all the routing resources (nodes). Each processor keeps its local version of the congestion cost data structure. The advantage of this approach lies in the absence of any mutual exclusion mechanism to maintain the integrity of the data structure. It is possible that the value of the congestion cost of a particular routing resource may not be consistent among different processors.

The pseudo-code of the parallel routing algorithm is given in Figure 7. Referring to lines 11 to 12, a processor is routing a signal N_i that it is assigned to. Each processor is required to broadcast messages to announce the *changes* in the delays and routing resources whenever a signal is rerouted. The first-order and second-order congestion costs are updated as soon as the signal is re-routed and the routing trees are rebuilt. For routing resources used by signals that are not assigned to the processor, the first-order congestion costs are updated as soon as messages are received in line 14, but routing trees are not reconstructed.

The second-order congestion costs for the routing resources used by the set of signals that are not assigned to the processor are not updated immediately but are deferred later till line 17. It is necessary because the second-order congestion costs must be increased gradually and monotonically per iteration. If the second-order congestion costs were updated in line 13, it is possible that the second-order congestion cost of a routing resource would be increased excessively if several processors were using the same routing resource.

Although the first-order congestion cost is updated as soon as the message is received, the parallel routing algorithm may not converge due to message latency. Imagine a scenario in which two processors are each routing one signal. As the two signals are competing for the same routing resources, it is possible that each processor would "see" the same cost landscape as it finds the best routes for its signal. The same situation has also been depicted in Figure 5. Due to message latency or merely concurrency, the two signals would oscillate between routing iterations, since each processor only knows where the other processor's signal was in the last iteration, but not the current one.

So it is necessary for the parallel routing algorithm to restrain its parallelism by collapsing to *single mode* when the number of shared resources is less than a certain value (typical value is 5). That is, having one processor that takes over and completes the routing of the last few signals. We have observed in all the experiments described in Section 5.3 that Parallel Pathfinder converges successfully. Details will be provided in the implementation section.

4.2 Synchronizing the processors

The previous subsection described a parallel routing scheme in which each processor is assigned a disjoint subset of signals to route concurrently. Each processor has a private local memory and each processor is connected to a network. Processors communicate with each other by sending and receiving messages. The network serves as the medium for transferring messages. There is no guarantee regarding the order or timing of the arrival of the messages to a processor. This is a source of indeterminism in the parallel router. Moreover, processors are not tightly synchronized at every iteration in order to avoid processor idling. All processors start at iteration $k = 1$. Each processor decides individually to advance to the next iteration k. Let k_i be the current iteration count of processor i. There is one designated processor that keeps track of the current iteration count of all processors. A running average $\tilde{k}$ of all iteration counts is computed and broadcast. Since the second-order congestion costs of the routing resources depend on k, it is important to set limits on how far the processors can be ahead of or behind each other. So, it is maintained that $k_i - \tilde{k} < B$. Typically, we use $B = 2$. If a processor is too far ahead, it will perform no routing task and just listen to the network until the other processors catch up.

4.3 Delay-Driven routing

The key requirement in delay-driven routing in serial PathFinder is to compute the value of the longest path of the entire circuit, D_{max}, at each iteration [7].

However, in Parallel PathFinder, processor i only has a subset of the signals to route. The delay of all of it's signals must be known in order to accurately compute the critical path of the circuit. Hence, the delays of the rest of the signals are provided by other processors via messages, as indicated in line 11 of Figure 7. Due to message latency, the computation of the delay of the critical path of the entire circuit at iteration k_i by processor i, D_{max,k_i}, can be imprecise. Moreover, each processor may not be synchronized at the same iteration count. If each processor informs others about its current D_{max,k_i}, a good approximant of D_{max} that can be computed is $D_{max} \approx \max_{k_i} \; D_{max,k_i}$. This will be used during the current iteration for delay-driven routing, which is another source of indeterminism in the parallel router. There are no explicit delay constraints for delay routing in Parallel Pathfinder.

4.4 Net Ordering and Load Balancing

The distribution of signals among different processors affects the performance, load balancing, and consequently speedup. We order the signal nets according to increasing order of fanouts which serves the following purposes. First, the results that we have derived in Section 3 suggest a fixed ordering of the signals. Recall that in PathFinder all the signals are routed with their minimum achievable path delays during the first iteration. In subsequent iterations, the signals that have shared routing resources will be ripped up and re-routed. Since low-fanout signals are less likely to be on the critical paths and have more alternatives, it is more reasonable to re-route the low-fanout signals than the others. It is also more efficient to rip up low-fanout signals. Secondly, since the routing time of a signal depends on its fanout, the netlist is preprocessed to achieve load balancing among P processors. The netlist is sorted according to fanouts of the signals, then the sorted netlist is distributed evenly among P processors such that each processor has roughly the same number of sinks. Each processor will process a subset of the circuit in increasing order of fanouts.

5 Implementation

To evaluate the performance of the Parallel Pathfinder algorithm we modified the serial Pathfinder implementation to run on a network of workstations using `UNIX` sockets for interprocessor communication [13]. Our implementation of the Parallel Pathfinder algorithm is written in `C++`. When evoked, multiple copies of the PathFinder program run concurrently using sockets to send and receive messages among the parallel programs. We discuss the details of our implementation in Section 5.2.

Parallel PathFinder was implemented beyond a proof-of-concept for our distributed-memory routing algorithm; much effort has been put in to validating the speedup and the delay-driven capability of the parallel router. We decided that the best way to validate our router was to use commercial tools as the yardsticks for routing and performance validations on actual designs rather than artificial benchmarks. This will be discussed in Section 5.1.

5.1 Validation

We target Parallel Pathfinder to route circuits on a real FPGA architecture. The requirements are: first and foremost, we must verify that our router generates correct routes. Secondly, we must measure the performance of the routed circuits. Third, we use real circuits in our validation instead of artificial benchmark circuits.

The yardstick that we use is a very mature commercial Xilinx routing program `PPR` (version 5.2.1) and its supporting tools `XACT` and `Xdelay`. `PPR` is Xilinx placement and routing program for the XC4000 family FPGAs, and `PPR`'s delay-driven routing algorithm is well-documented in [6]. `XACT` is a graphic editor, it has a design-rule checker that can be used to validate the routes generated by a router. `Xdelay` is a static delay analyzer that can be used to measure the performance of a placed and routed design.

We briefly describe the circuits that are used in our experiments. `ISA-x0` is an ISA bus controller, this circuit uses multiple clock sources. `Xor-neural` contains "hidden units" of a digital neural network that implements the XOR function [14]. This design has hard macros, and part of the placement was done by hand. `Pci-core` is Xilinx (HighGate Design) PCI logic core. The placement of this circuit is largely done by hand using relative location attributes. This circuit also contains delay constraint (time-group) instructions for `PPR`. `Pri-Queue` is a 16-element systolic priority queue. This circuit is floor-planned by hand using relative location attributes. `Sat-r2` is part of a reconfigurable Satisfiability (SAT) solver. It contains about 350 implication logic modules [15]. This circuit has 4 very high-fanout signals that broadcast to all the logic modules. `S15850` is an MCNC benchmark circuit. This circuit has 900 CLBs and occupies 87% of a XC4025 part. The rest are MCNC combinational benchmark circuits

5.1.1 From FPGA architecture to routing graph

All of the Xilinx design tools `PPR`, `XACT`, and `Xdelay` read and write Xilinx LCA format. The LCA format is not in the public domain and it holds critical information regarding the Xilinx XC4000 architecture. In order to facilitate a direct comparison, our parallel router must be capable of reading and writing out LCA format. This is a tedious task, we carefully studied the XC4000 architecture, and observed the correspondence between the routing resources and the LCA file format. We represent the XC4000 architecture by 22 different tiles according to their functions or geographic locations within the XC4000 FPGA die. We translate all the physical routing resources into an architecture graph to be processed by Parallel Pathfinder. Hence all the routing features (such as global long lines, double-length lines, switch matrices, feedthrough, tri-state buffers, pin swapping and fast carry) in the XC4000 architecture are represented. The architecture graph for a particular package type is generated on-the-fly by our router. When routing is completed, the router translates all the routes back to Xilinx LCA format. A physical wire is represented as a node and a Programmable Interconnection Point (PIP) is represented as two directed edges in the architecture graph, respectively. The Configurable Logic Block (CLB) tile constitutes the majority of the routing resources. The input pins of a CLB are represented as nodes and the LUT is represented as a fanout-less node (*sink*) in the architecture graph. By the same token, the output pin `Y` is represented by a fanin-less node (*source*). A route starts from a source and terminates at a sink. For a signal that must be realized with a tristate net and buffers, it has multiple sources. A tristate buffer is chosen arbitrarily as the source of the signal.

5.1.2 Delay model

In PathFinder, the basic physical routing resources are switches, wires and multiplexers. Switches and wires are represented as *architecture edges* and *nodes* in the architecture graph. Associated with each node n in the architecture is a constant delay d_n. Each wire also has a constant delay.

$$d_n = c_0 + c_1 \times fanin + c_2 \times fanout \tag{4}$$

Unlike the Xilinx tools which use uniformly the Elmore Resistor-Capacitor delay model [16], PathFinder uses a block-oriented delay model in which delay of a source-sink route is the sum of the delays of the routing resources along the route: $d_{route} = \sum_{resource\ from\ source\ to\ sink} d_i$ where d_i is the delay of a routing resource.

Moreover, a *logic path* consists of a sequence of blocks and routes, the delay of the logic path is the sum of the routing delays and block delays along the path.

$$d_{path} = \sum_{route\ along\ path} (d_{route} + d_{block})$$

Figure 8(a) illustrates the routing of a signal, the route consists of 2 PIPS and 3 wires. The delay of this route is $2d_{PIP} + 3d_{WIRE}$. Figure 8(b) illustrates the routing of a signal using 2 PIPS, 4 wires, and a switch box.

We extract the delays of these paths in `XACT` and use them to fit our delay model. This is rather ad hoc. In our final delay model, the delay of a PIP is 5 units, the delay of a wire is 1 unit and the delay of a switch is 5 units. The c_1 and c_2 coefficients in (4) are small constants in the range of 0.01. All buffers, restoring or non-restoring, have the same delay. This is a possible source of inaccuracy in our delay model.

5.2 Interprocessor communication

The current implementation of Parallel Pathfinder uses Unix `socket` for interprocess communication under the client/server model. Sockets are developed for the `C` language, so adapting the serial Pathfinder to Parallel Pathfinder requires changes to only the non-kernel components of the program. In essence, the following `C` statements are inserted to facilitate interprocessor communications.

```
write(socket id, message, message length)
read(socket id,  message, message length)
```

There are three different types of messages. Two are for insertion and deletion of the use of a particular routing resource to or from the current signal. The third type corresponds to a delay change for a signal with respect to one of its sinks which is needed for path delay calculation. Each processor has its unique socket identifier. Message broadcasting is implemented by unicasting copies of the message to each destination, one at a time. Multicasting will be more efficient, as it reduces the overhead on the network and the sender [17].

5.3 Results

In this section we present experiments which evaluate the benefit of our parallel router. A very mature and good commercial router `PPR` is used as the yardstick. We evaluate both the serial PathFinder and Parallel PathFinder. The results are reported in terms of routing time and performance of the routed designs. The readers are reminded that the serial and parallel PathFinder are not developed specifically for a single FPGA architecture. So no effort has been put into fine tuning the C codes of the serial and parallel routers to accommodate the Xilinx architecture.

First, we perform experiments on the 5 circuits described earlier in Section 5.1 with both `PPR` and the *serial* PathFinder. All the experiments were conducted using 143MHz UltraSparc workstations each with 64 Mbytes of random access memory. The memory requirement for each copy of the Pathfinder program is 32 Mbytes for an XC4013 part, and is 50 Mbytes for an XC4025 part.

`PPR` was evoked using the default parameters. The serial router completely routed all designs using the *route share* option without using the A* search option.[3] We report both routing time and design performance in Table III. The results indicate that *serial* PathFinder is capable of producing competitive performance with the XC4000 FPGA architecture. The performance-critical circuit `Pci-core` deserves some attention since a lot of effort has been put into optimizing its placement. We are glad to see that our router did as well as `PPR`.

Next, we evaluate our parallel router using the serial router as the reference. Speedup of the parallel router will be measured with respect to the serial version. As in the serial version, the parallel router completely routed all the designs using the *route share* option without the A* search option. Input to the parallel router consists of 1) the LCA file of a design that has been placed by `PPR`, and 2) a configuration file with the names of the workstations on the network to be used for parallel routing.

We should also mention that in the current implementation of Parallel Pathfinder, not only is routing of signals done in parallel, several postprocessing steps such as checking of illegal routes and conversion from the routing information to Xilinx LCA file format are also computed in parallel.

Output of our parallel router is an LCA file, ready to be examined for performance by `Xdelay` and validated by `XACT` to ensure zero design rule errors.

Due to the intrinsic indeterminism in the parallel router, we run the same design 10 times and collect

[3] Pathfinder has an architecture independent A* search option [7]. In order to use the A* search option, a file containing estimates of the minimum delay from any node to any possible sink node must be generated ahead of time. This file is about 32 Megabytes for the XC4013 part types. So just reading this file will take a few minutes across the network. Also due to the long lines in the XC4000 architecture, these estimates are not useful.

the minimum, maximum, average, standard deviations to deem the experiments to be statistically significant. Figure 9 shows the (average) speedup of the parallel router for up to 5 processors. The circuit `sat_r2` has many small modules, so it has the best speedup performance, while the circuit `xor_neural`'s speedup saturates very early at about 2 processors. Quite noticeable is that speedup of 2.5 is attainable using 3 processors.

Figure 10 plots the performance of the routed designs produced by the parallel router. The performance is measured by Xilinx delay analyzer, `Xdelay`. The performance of the routed design is along the vertical axis, and the horizontal axis is the number of processors. The ideal performance curve would be a horizontal straight line regardless of the number of processors. Due to the external indeterminism of the network environment, the performance varies from run to run. The average performances of the routed designs along with the maximum and minimum are reported. Figure 10 reveals that the average performance doesn't degrade significantly with increasing number of processors. This is a desirable feature. As a matter of fact, the performance may even exceed the serial Pathfinder. We attribute this feature to the robustness of the timing-driven routing algorithm, which is controlled by the delay of the critical path D_{max}; since this is the key information that prescribes the performance of a routed design. We have also conducted some control experiments that deliberately removed the transmission of D_{max,k_i} to other processors; this results in large variations in performance of the routed designs.

5.4 Enhancements and Future work

In this section, we discuss schemes that can improve the parallel router further. These schemes require resources that we do not possess, therefore none of these schemes have been implemented.

Predictable route: Since each processor maintains its local version of congestion costs, the updating of the costs relies on interprocessor messages. Consequently, the routes generated by a processor depend on the order and timing of messages that it receives. Different routes may be produced depending on the physical configuration of the network, network traffic, and workstation loads. All of these factors affect the ordering of the messages. The message-order variation can be minimized by running the parallel router within a controllable network environment. It is not uncommon to find network configurations using switch media supporting a cluster of workstations. There is a certain degree of isolation with switched networks. Network traffic within the switched network is localized. In this case, the performance of *Parallel Pathfinder* is determined by the activity of the workstations within the cluster.

More speedup: The current implementation of Parallel Pathfinder uses Unix socket for interprocess communication under the client/server model. Broadcasting of changes in routes is implemented by uni-casting copies of the changes to each destination, one at a time. Multicasting will be more efficient, as it reduces the overhead on the network and the sender [17].

The processors running Parallel Pathfinder send to and receive a lot of messages from each other. Message passing using the standard Unix socket is slow. It is conceivable that a faster message-passing scheme would result in faster routing time. There are faster message passing schemes such as Fast Message (FM) Myrinet [18] in which a message can be delivered in microseconds across the network.

Also, larger speedup can be obtained by combining coarse-grain and instruction-level parallelism. The kernel of this negotiation-based router (its path search core) has instruction-level parallelism [4]. This parallelism enables parallel memory access and processing of data within a single instruction cycle. By combining coarse-grain and instruction-level parallelisms, the speedup is multiplicative.

6 Conclusions

We have examined in-depth the convergence of the negotiation-based routing algorithm by modeling it as a conflict resolution method for bipartite matching. We have presented results on the convergence of the negotiation-based conflict resolution method for graphs, and shown that it will not always converge for hypergraphs. Nevertheless, our examination provides useful guidelines for the development of a parallel negotiation-based routing algorithm. We have built a parallel negotiation-based router for the Xilinx XC4000 series FPGA. Experiments have demonstrated that our parallel router performs very well with practical circuits. The parallelization method that we have described is applicable to any negotiation-based algorithms. We are exploring this concept for parallel placement.

References

[1] 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*, (Monterey, California, USA), pp. 111–117, Feb. 1995.

[2] J. S. Swartz, V. Betz, and J. Rose, "A fast routability-driven router for FPGAs," in *Proceedings of International ACM/SIGDA Symposium on Field-Programmable Gate Arrays*, (Monterey, California, USA), pp. 140–149, Feb. 1998.

[3] R. Tessier, "Negotiated A* routing for FPGAs," in *Proceedings of the Fifth Canadian Workshop on Field-Programmable Devices*, (Quebec, Canada), June 1998.

[4] P. K. Chan and M. Schlag, "Acceleration of an FPGA router," in *1997 IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 175–181, 1997.

[5] S. Brown, J. Rose, and Z. G. Vranesic, "A detailed router for Field-Programmable Gate Arrays," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. CAD-11, pp. 620–628, May 1992.

[6] J. Frankle, "Iterative and adaptive slack allocation for performance-driven layout and FPGA routing," in *ACM IEEE* 29^{th} *Design Automation Conference Proceedings*, (Anaheim, California), pp. 536–542, June 1992.

[7] C. Ebeling, L. McMurchie, S. Hauck, and S. Burns, "Placement and routing tools for the Triptych FPGA," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 3, pp. 473–481, Dec. 1995.

[8] P. Banerjee, *Parallel Algorithms for VLSI Computer-Aided Design*. Prentice Hall, 1994.

[9] M. Martonosi and A. Gupta, "Tradeoffs in message passing and shared memory implementations of a standard cell router," in *Proceedings of International Conference on Parallel Processing (ICPP)*, Aug. 1989.

[10] Y. Won and S. Sahni, "Maze routing on a hypercube multiprocessor computer," in *Proceedings of International Conference on Parallel Processing (ICPP)*, Aug. 1987.

[11] Z. Xing, J. Chandy, and P. Banerjee, "Parallel global routing for standard cells," in *Proceedings of International Parallel Processing Symposium (IPPS-97)*, Apr. 1997.

[12] J. Rose, "Parallel global routing for standard cells," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 9, pp. 1085–1095, Oct. 1990.

[13] W. R. Stevens, *UNIX Network Programming.* Prentice Hall, 1990.

[14] A. T. Ferrucci, M. H. Martin, T. Geocaris, M. Schlag, and P. K. Chan, "ACME: A Field-Programmable Gate Arrays implementation of a self-adapting and scalable connectionist network," in *Proceedings of the* 2^{nd} *International ACM/SIGDA Workshop on Field-Programmable Gate Arrays*, (Berkeley, California, USA), Feb. 1994.

[15] P. K. Chan, M. Boyd, S. Goren, K. Klenk, V. Kodavati, R. Kundu, M. Margolese, J. Sun, K. Suzuki, E. Thorne, X. Wang, J. Xu, and M. Zhu, "Reducing compilation time of Zhong's FPGA-based SAT solver," in *1999 IEEE Workshop on FPGAs for Custom Computing Machines*, 1999.

[16] XILINX*: The Programmable Gate Array Data Book.* 2100 Logic Drive, San Jose, CA 95124, 1993.

[17] K. C. Miller, *Multicast networking and applications.* Addison-Wesley, 1999.

[18] S. Pakin, M. Lauria, M. Buchanan, K. Hane, L. Giannini, J. Prusakova, and A. Chien, *Fast Message (FM) 2.0.* University of Illinois at Urbana-Champaign, 1304 W. Springfield, Urbana, IL 61801, 1996.

Pak K. Chan received the B.Sc. degree in Electronics from the Chinese University of Hong Kong, and the M.S. and Ph.D. degrees in Computer Science from the University of California, Los Angeles in 1984 and 1987. During the summer of 1986, he worked at the IBM T.J. Watson Research Center. Presently he is an Associate Professor of Computer Engineering at the University of California, Santa Cruz. Pak K. Chan has served as associate editor of IEEE Transactions on Computers and General Chair for the International Symposium on FPGAs. His research interests include computer arithmetic, algorithms, timing analysis, circuit theory, and Field-Programmable Gate Arrays.

Martine Schlag received the B.A. degree in mathematics, the M.S. and Ph.D. degrees in computer science from the University of California at Los Angeles in 1978, 1982 and 1986 respectively. She is currently Professor of Computer Engineering at the University of California, Santa Cruz. Her interests are in physical design algorithms for integrated circuits and her recent work has focussed on tools for FPGA design and FPGA-based accelerators. In 1987 she received an NSF Presidential Young Investigator Award. She served on the editorial board of the IEEE Transactions on Computers from 1992 to 1997 and as Program Chair for FPGA'2001.

Carl Ebeling received his B.S. in Physics from Wheaton College in 1971, his M.S. in Computer Science from Southern Illinois University, Carbondale, in 1976, and his Ph.D. from Carnegie-Mellon University in 1986. He joined the faculty of the University of Washington in 1986 where he is currently professor of Computer Science and Engineering. He has been actively involved in developing undergraduate and graduate hardware courses and establishing a new computer engineering hardware design laboratory. In 1987 he received an NSF Presidential Young Investigator Award and in 1995 received the University of Washington Distinguished Teaching Award.

His primary research interests are in the area of VLSI architectures, configurable computing, and computer-aided design of complex digital systems. His current research project is RaPiD, a reconfigurable architecture for high-performance embedded image and signal processing. He has served as program co-chair for the Conference on Advanced Research in VLSI and program chair for the International Symposium on FPGAs. In a former life he designed the Hitech chess machine and wrote the Gemini layout-to-schematic comparison program.

Larry McMurchie received his Ph.D. in Physical Chemistry from the University Washington (Seattle, Washington) in 1977. While a graduate student and a post-doc, he developed an algorithm for evaluating integrals over multi-center Gaussians and employed it in the calculation of molecular spectra.

In 1982, he joined the research staff of the Dept. of Computer Science at the University of Washington and began work in VLSI architectures and CAD. While in this position, he developed tools for specifying and routing Field Programmable Gate Array (FPGA) architectures.

In 1998, he moved to a research staff position with the Dept. of Electical Engineering also at the University of Washington. His current interests include high-performance logic families and molecular computing.

List of Tables

List of Figures

```
NC(netlist, FPGA architecture, RouteAll/RouteSharedOnly)
while there are shared routing resources
  foreach signal S_i
  if RouteAll or S_i has shared nodes
  then rip up S_i
      RT ← n_s (source node of S_i)
      while there are sinks of S_i unconnected to n_s
          P_j ← find-shortest-path(RT, n_s)
          add P_j to RT
      endwhile
      update 1st and 2nd order costs of RT nodes
  endfor
endwhile

find-shortest-path(RT, n)
foreach node n 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
        foreach 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
return this path
```

Fig. 1. The NC Algorithm.

	0	1	2	3	4	5	6	7	8	9	10	11	12
	-	(1,4,8)	(3,5,9)	(1,6,7)	(3,4,8)	(1,5,7)	(3,4,8)	(1,5,9)	(3,6,7)	(1,4,8)	(3,5,7)	(1,4,8)	(3,5,9)
A	S_2	0 S_2	0 S_1	1	1 S_1,S_2	2	2 S_1,S_2	3	3 S_1	3 S_2	3 S_1	4 S_2	4 S_1
B		0	0 S_1,S_2	1	1 S_1	1 S_2	1 S_1	2 S_2	2 S_1	3	3 S_1,S_2	4	4 S_1,S_2
C		0	0 S_1	0 S_2	0 S_1	1	1 S_1	1	1 S_1,S_2	2	2 S_1	2	2 S_1
D		0	0 S_1	0 S_2	0 S_1	1	1 S_1	1	1 S_1,S_2	2	2 S_1	2	2 S_1
E	S_1	0 S_1	0 S_3	0 S_1	1	1 S_1	1	1 S_1,S_3	2	2 S_1	2	2 S_1	2 S_3
F	S_1	0 S_1	0 S_3	0 S_1	1	1 S_1	1	1 S_1,S_3	2	2 S_1	2	2 S_1	2 S_3
G	S_1	0 S_1,S_3	1	1 S_1	1 S_3	1 S_1	2 S_3	2 S_1	3	3 S_1,S_3	4	4 S_1,S_3	5
H	S_1,S_3	0 S_1	1	1 S_1,S_3	2	2 S_1,S_3	3	3 S_1	3 S_3	3 S_1	4 S_3	4 S_1	5

Table I. The behavior of the algorithm NC on the problem in Figure 2. The assignments and costs of Y vertices A,B,C,D,E,F,G,H are given for each iteration. The costs at the start of the iteration and resulting the assignments at the end of the iteration are given. The edges selected in each iteration are given in the second row. The assignments in iterations 1 and 11 are repeated and the sequence of assignments repeats every 10 iterations since each signal's edge costs increase by the same amount (except for edge 2 which unfortunately increases by more and is never selected).

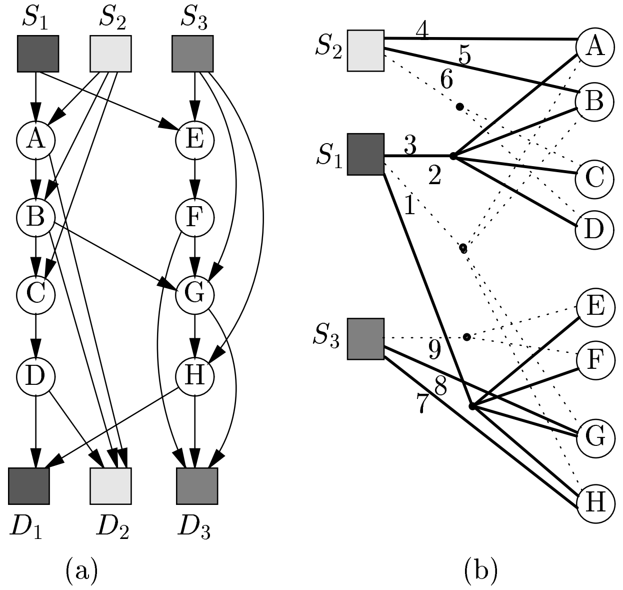


Fig. 2. A routing problem (a) which yields a hypergraph matching problem (b) in which the negotiation-based method using immediate history cost update fails to find a solution. The solution is given by the three dotted hyperedges: 2,6,9.

```
NCG(a(), cost(), Select(), ReassignAll)
while ∃y ∈ Y with n(y) > 1
    for i = 1 to |X|
        if ReassignALL or n(a(x_i)) > 1 then
            n(a(x_i)) ← n(a(x_i)) − 1
            a(x_i) ← Select(x_i)
            n(a(x_i)) ← n(a(x_i)) + 1
        endif
    endfor
    for each y ∈ Y with n(y) > 1
        cost(y) ← cost(y) + 1
    endfor
endwhile
```

Fig. 3. The pseudo-code for **NCG**.

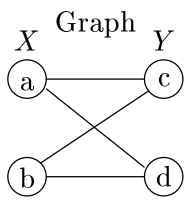
NCG with	**SelectOpen**	**SelectCheapest**
ReassignAll = **TRUE**	Converges	Doesn't converge
ReassignAll = **FALSE**	Converges	Converges

Table II. Summary of convergence behavior of **NCG** with all four combinations of parameters.

```
SelectCheapest(x)
y_new ← a(x)
for each y adjacent to x
    if cost(y) < cost(y_new)
    then y_new ← y
endfor
return(y_new)
```

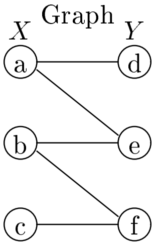
```
SelectOpen(x)
FoundOpen ← FALSE
y_new ← a(x)
for each y adjacent to x
    if n(y) = 0
        FoundOpen ← TRUE
        if FoundOpen = FALSE or cost(y) < cost(y_new)
          then y_new ← y
    else if FoundOpen = FALSE and cost(y) < cost(y_new)
        then y_new ← y
endfor
return(y_new)
```

Fig. 4. The pseudo-code for **SelectCheapest** and **SelectOpen** routines which are used to select a Y vertex to assign to a given $x \in X$. **NCG** is called with one of these two routines as an argument.



Iteration	1		2		3		4		5	
	cost()	$a^{-1}()$	cost()	$a^{-1}()$	cost()	$a^{-1}()$	cost()	$a^{-1}()$	cost()	$a^{-1}()$
c	0	a,b	1	-	1	a,b	2	-	2	a,b
d	0	-	0	a,b	1	-	1	a,b	2	-

Fig. 5. An example in which **NCG** with *ReassignAll* = **TRUE** and **SelectCheapest** does not terminate even though a solution exists. The graph is given on the left. Column i has the cost and assignment of the Y vertices in iteration i after processing all X vertices but before updating the costs.



Iteration	1		2		3		4		5	
	cost()	$a^{-1}()$	cost()	$a^{-1}()$	cost()	$a^{-1}()$	cost()	$a^{-1}()$	cost()	$a^{-1}()$
d	0	-	0	-	0	-	0	-	0	-
e	0	a	0	a,b	1	a	1	a,b	2	a
f	0	b,c	1	c	1	b,c	2	c	2	b,c
Order of X	$< a, b, c >$		$< a, b, c >$		$< c, b, a >$		$< a, b, c >$		$< c, b, a >$	

Fig. 6. An example in which varying the order in which X vertices are examines keeps **NCG** with *ReassignAll* = **FALSE** from finding the solution. The graph is given on the left. Column i has the cost and assignment of the Y vertices in iteration i after processing all X vertices but before updating the costs.

```
Parallel Pathfinder(netlist, FPGA architecture, RouteShareOnly)
Route all signals belong to my processor, ignore shared costs
Communicate with other processors to update node costs

while there are shared routing resources
    foreach signal S_i owned by me that has shared resources
        rip up S_i
        RT ← n_s source node of S_i
        while there are sinks of S_i which are not routed
            P_j ← find-shortest-path(RT, n_s)
            add P_j to RT
        endwhile
        Broadcast changes of delays along signal to others
        Broadcast changes of route to other processors
        Update 1st- and 2nd-order costs of nodes used by N_i
        Listen to other processors and update 1st-order costs
    endfor
    foreach signal S_j not assigned to my processor
        Update 2nd-order congestion cost of nodes used by S_j
    endfor
endwhile
```

Fig. 7. Parallel Pathfinder algorithm: action of one processor.

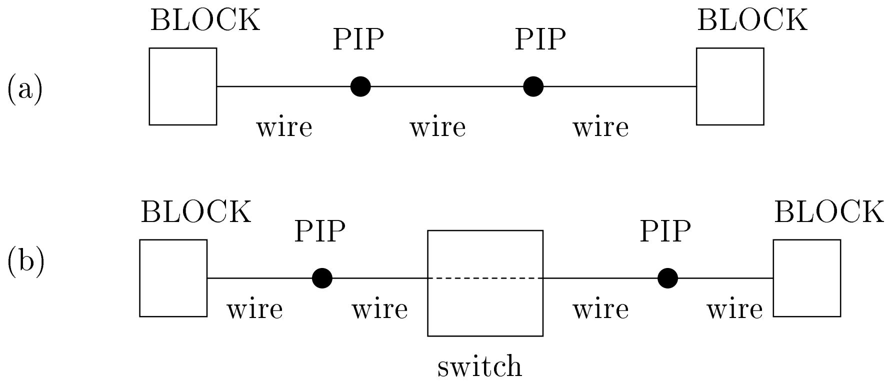


Fig. 8. Wires, pips, switches, and blocks in the XC4000 FPGA.

Design	FPGA device	CLB%	nets	PPR		Serial Pathfinder	
				Performance	route time	Performance	route time
isa_x0	XC4003-4	40%	118	27.7MHz	50.2	27.7MHz	35.7
xor_neural	XC4010-4	38%	357	15.5MHz	200	15.4MHz	191
pci_core	XC4013-4	37%	466	21.3MHz	349	21.6MHz	480
pri_queue	XC4013-4	78%	794	9.0MHz	339	10.0MHz	641
sat_r2	XC4013-5	100%	1183	16.9MHz	1008	15.7MHz	1644
s15850	XC4025-5	87%	1542	4.3MHz	1578	3.7MHz	2184
alu4	XC4005-4	98%	276	190ns	90	165ns	408
apex4	XC4010-5	100%	1235	231ns	7183	235ns	5266
misex3	XC4005-5	79%	244	148ns	66	138ns	348
seq	XC4008-5	100%	929	403ns	377	317ns	2560

Table III. Comparison of *serial* pathfinder and `PPR` with different designs. Speed (Performance) of designs are computed using `Xdelay` and are all in Mega Hertz except for the combinational circuits: `Alu4`, `apex4`, `misex3`, and `seq`. Routing times are in elapsed time in seconds.

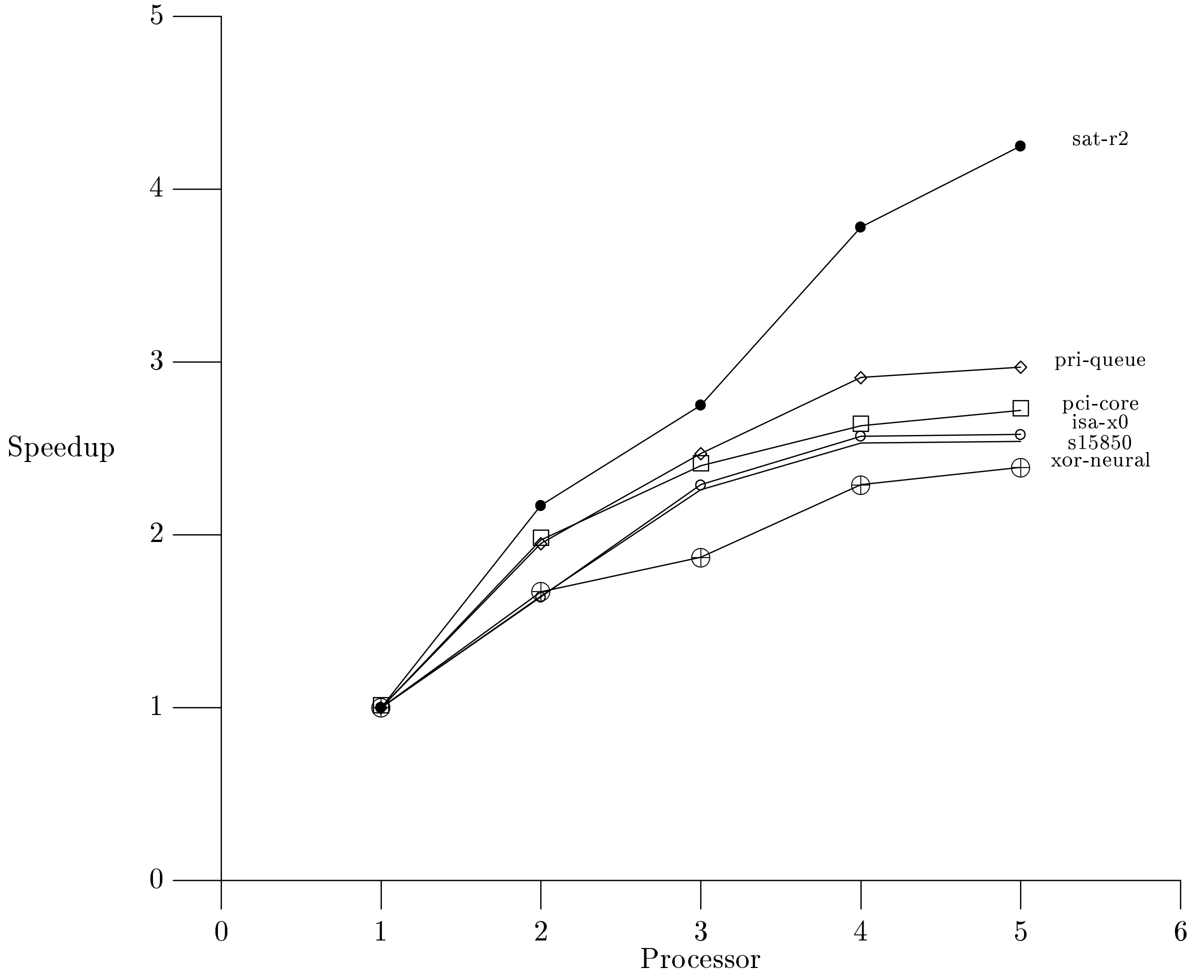


Fig. 9. Speedup of the Parallel Router.

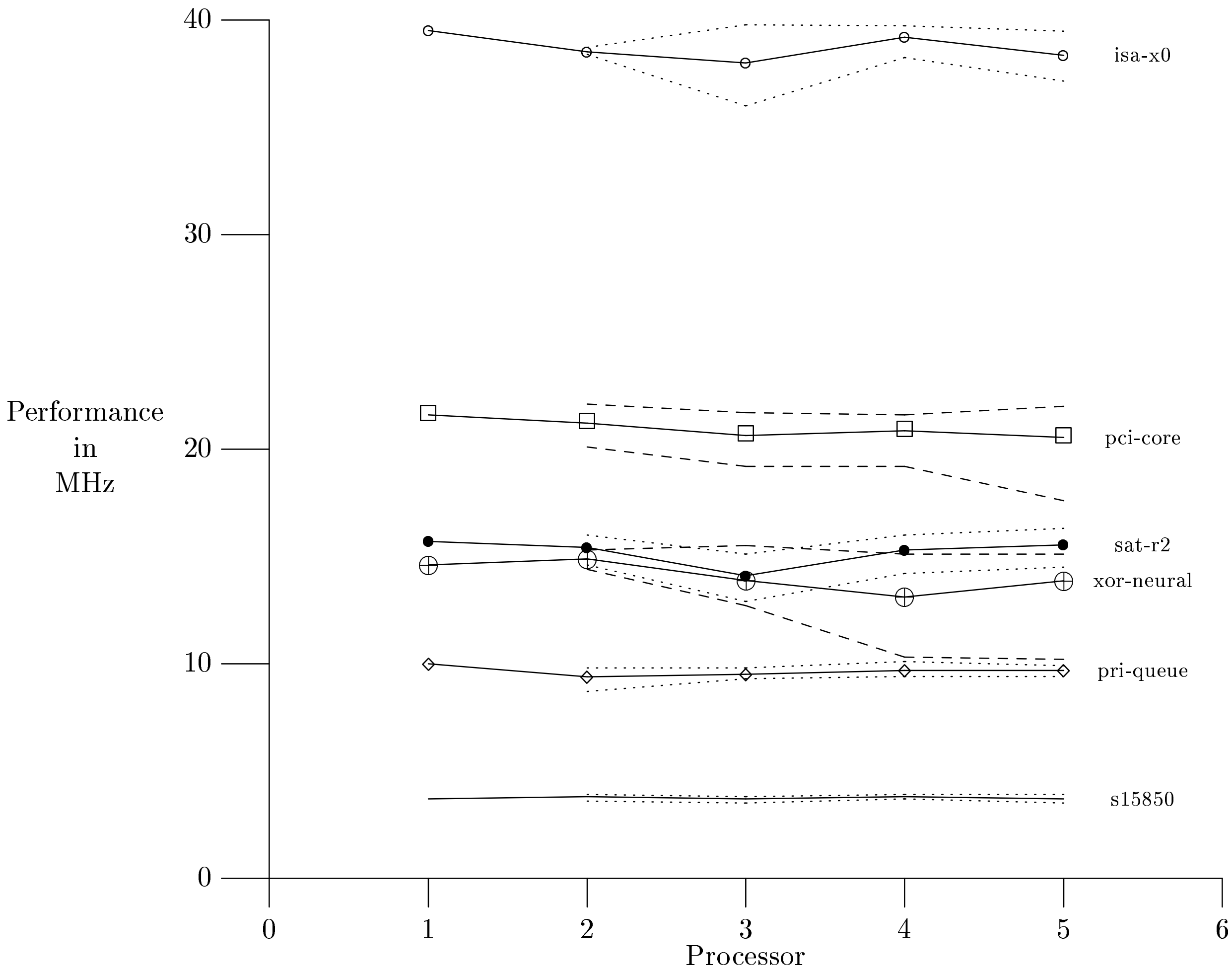


Fig. 10. Performance of the routed designs. Solid lines are average performance. Dotted and dashed lines are minimum or maximum performance.