Incremental Reconfiguration of Multi-FPGA Systems*

K. K. Lee[1] and D. F. Wong[2]

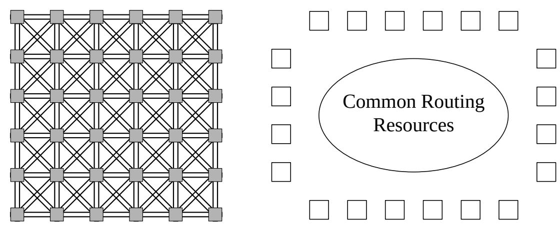
[1]Synopsys, Inc., Mountain View, CA
[2]Department of Computer Sciences, University of Texas at Austin, Austin, TX

ABSTRACT

In reconfigurable computing, circuits implemented on multi-FPGA systems have to be incrementally modified. Since reconfiguring an FPGA is time-consuming, the time for reconfiguration depends on the number of FPGAs to be reconfigured. Our objective is to reduce the number of such FPGAs. In this paper, we consider the specific problem of incrementally reconfiguring a multi-FPGA system that utilizes the direct interconnection architecture, where routing connections between FPGAs are to neighbors that are near. This problem can be divided into a net addition problem and a net deletion problem. We show that the net addition problem is a generalization of the NP-complete Steiner tree problem. Our algorithm for this problem is based on an adaptation of the Klein-Ravi approximation algorithm for the node-weighted Steiner tree problem. As for the net deletion problem, we prove that it is NP-complete but the problem is solvable in polynomial time for tree topologies. Based on the algorithm for trees, we design an effective heuristic algorithm for the general net deletion problem. Finally, we present an algorithm for solving the incremental reconfiguration problem which handles both placement of new gates and inter-FPGA routing.

1. INTRODUCTION

Hardware solutions have been used to speed up time-consuming applications like circuit verification where even parallel software solutions have been found to be wanting. Reconfigurable computing on multi-FPGA systems becomes a natural choice in these type of applications. In such problems, the circuit is partitioned into clusters, where each cluster is implemented on an FPGA. The FPGAs are then connected through external routing resources. Moreover, the circuit has to be modified very often, as the algorithm progresses through stages. The changes occur throughout the entire cycle of the computation. Such modifications to existing circuit are similar to those due to ECO changes where design changes have to be made to existing designs. However, in such problems, incremental changes are required very frequently, not as a result of design changes, but as a result of computational requirements. Thus, reconfiguration is very important for enabling such applications. Unfortunately, these design changes are extremely difficult to implement and error prone due to a lack of automation tools and methodologies. Another consideration is that it can be very time-consuming. As an example, suppose there are 300 FPGAs in the system[3], and that it takes 30 minutes to configure each FPGA. This means that it takes 150 computer-hours to implement the entire system for a single stage of change. Now, suppose that we need to make 100 changes (i.e., there are 100 stages) to this system. Completely re-configuring this system implies that we need 15,000 computer-hours to implement all the changes. Even if a farm of 50 computers were to be used for configuring the FPGAs in parallel, we would still need almost two weeks for the whole task. Thus, it is not desirable to completely re-configure all the FPGAs after each change. It is clear that the smaller the number of FPGAs that need to be reconfigured, the faster the time for reconfiguration. This paper addresses the problem of minimizing the number of chips that are needed to be reconfigured after each incremental change.



(a) A direct interconnect architecture using the 8-way mesh.

(b) An indirect interconnection architecture. Connections between FPGAs are made through common routing resources.

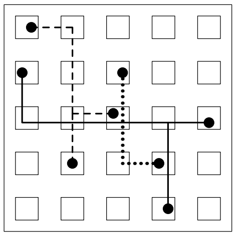
Figure 1: Two main interconnection architectures

There are two main types of routing architectures for multi-FPGA designs, namely *direct* and *indirect* connection style. In the direct interconnection architecture, dedicated routing resources are provided between FPGAs. As such, the connections are often local and limited, perhaps only to its immediate neighbors. A natural implementation is the mesh architecture. Figure 1(a) shows an example of the 8-way mesh, where the FPGAs are arranged in a grid and connect only to its 8 immediate neighbors. Several successful

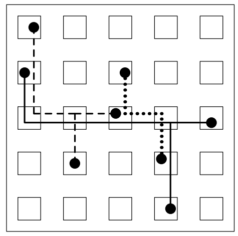
*This work was partially supported by the National Science Foundation under grant CCR-9912390.

[3]A multi-FPGA system with 300 FPGA is reasonable. For example, Axis' Xcite-2000 system has 240 FPGAs.

FPGA'02, February 24-26, 2002, Monterey, California, USA.



(a) 14 FPGAs need to be reconfigured



(b) 11 FPGAs need to be reconfigured

Figure 2: An example of the Net Addition Problem for multi-FPGA systems. Three new nets are to be connected and each FPGA that lies on the routing tree of a net has to be reconfigured. By carefully sharing intermediate FPGAs, the number of FPGAs that requires reconfiguration can be reduced.

multi-FPGA systems using the mesh style have been reported in [3, 10, 7, 15, 17, 20]. The original Quickturn emulation system [21] and the IKOS system [8] are examples of commercial multi-FPGA systems that utilize this architecture [19].

In the indirect architecture, the FPGAs are connected to a pool of field programmable interconnect devices (FPIDs), and connections between FPGAs are made by programming the interconnect resources. Figure 1(b) shows an example of such a system. The *partial crossbar* is one of the more popular methods for this implementation. The Quickturn Mercury system [16] is an example of such a multi-FPGA system.

There are some tradeoffs involved between the two main architectures. The direct method is simpler, faster and easily scaled, but needs to use routing resources both between and *within* FPGAs in order to complete connections. It is particularly suited for computations that exhibit *locality effect*, where the code changes occur very slowly over time. On the other hand, an indirect-style interconnection architecture does not use routing resources within FPGAs except for FPGAs affected by logic changes. However, due to pin limitations of FPIDs, FPIDs cannot scale as the number of logic modules increases in the multi-FPGA system.

In this paper, we introduce the reconfiguration problem for direct style multi-FPGA systems. The underlying system already has a circuit implemented on it, and we are required to reconfigure the system by deleting some nets and adding new nets, subject to capacity constraints on each of the FPGAs and the connections between FPGAs. This corresponds to the reconfiguration changes for one stage of reconfigurable computing. Our objective is to reduce the number of FPGAs that need to be reconfigured in order to reduce FPGA reconfiguration time, since each reconfiguration of an FPGA involves re-placement and re-routing of circuits. There are two subproblems to be solved, which we call the *Net Addition Problem* (NAP) and the *Net Deletion Problem* (NDP).

In the NAP, we are given a set of terminals of nets to be placed and routed. We want to connect each net using the dedicated routing resources between FPGAs and *within* FPGAs. Our objective is to minimize the total number of FPGAs the nets pass through, since each of these FPGAs has to be reconfigured. In Figure 2, we show an example where three nets are to be connected. The multi-FPGA architecture is the 4-way mesh style (the connections are not shown in the diagram). Figure 2(a) shows a routing solution using the shortest path algorithm which requires that a total of 14 FPGAs be reconfigured. However, a better solution is given in Figure 2(b), where the number of FPGAs to be reconfigured is 11.

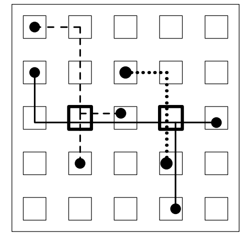


Figure 3: An example of the Net Deletion Problem for multi-FPGA systems, where three routes are shown. By reconfiguring the FPGAs shown in bold, all the three nets shown can be broken.

In the NDP, we are given the routing tree of a set of nets. We want to determine a minimum set of FPGAs such that each net is broken by some FPGA in the set (when reconfigured). It is often not necessary to reconfigure all the FPGAs in these routings because a break in a path causes the path to be electrically disconnected. The rest of the FPGAs that the nets passes through and the LUTs associated with the ports of the LUTs can be marked for reconfiguration later in order to save on the time for reconfiguration. The set of FPGAs chosen has to break every path in each net. Thus each net becomes ("lazily") deleted when the set of FPGAs are reconfigured to disconnect the nets. Figure 3 shows an example. By reconfiguring the FPGAs shown in bold, every net is now broken.

We show that the net addition problem is a generalization of the NP-complete Steiner tree problem. Our algorithm for this problem is based on a modification of the Klein-Ravi approximation algorithm for the node-weighted Steiner tree problem. As for the net deletion problem, we prove that it is NP-complete but the problem is solvable in polynomial time for tree architectures. Based on the algorithm for trees, we design an effective heuristic algorithm for the general net deletion problem. Finally, we present an algorithm for solving the incremental reconfiguration problem which handles both placement of new gates and inter-FPGA routing for the direct style interconnection architecture.

The rest of this paper is organized as follows. Section 3 introduces some definitions and terminologies used in this paper. Section 4 introduces the Net Addition Problem. This problem is NP-complete, and we give an algorithm for determining the FPGAs to be reconfigured after placement and routing. Section 5 introduces the Net Deletion Problem. We show that this problem is also NP-complete, but show that deleting a net is polynomial-time solvable. We give an algorithm to determine a small set of FPGAs to reconfigure in order to delete all these nets. In Section 6, we show how to handle these two problems together. Since there are no existing test cases for this problems, we explain how the experiments are setup in Section 7. We show the results of our algorithm and that based on shortest-path. Finally, in Section 8, we give some conclusions and some extensions.

2. RELATED RESEARCH

Tessier in [19] dealt with the problem of incremental compilation in logic emulation. Their solution involves using a multi-way partitioning algorithm to incrementally partition the circuit to minimize the interconnections of the new circuit. Each partition is then implemented on an FPGA and incrementally routed. However, their

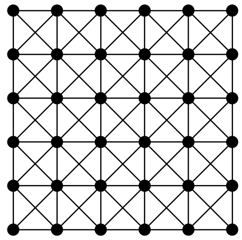


Figure 4: Graph representation of the system in Figure 1(a). Each vertex represent an FPGA. Weights on vertices model the logic capacity and weights on edges model connectivity capacity.

two test cases used only 16 FPGAs in each case, and are therefore quite small compared to the system we are considering, which has hundreds of FPGAs. Moreover, we are not aware of any previous publication that deals with the specific incremental reconfiguration problem studied in this paper. In the problem considered in this paper, the amount of leftover capacity (from the circuit implemented from the previous stage) on the FPGAs are non-uniform and the multi-way partitioning algorithm will have to dynamically decide on each partition capacity to use for the partitioning algorithm and optimally determine which of the FPGAs to use. The number of potential partitions is in the order of hundreds for reasonably sized multi-FPGA systems.

There have been some related research on routing and on different multi-FPGA architectures. Previous research on routing and architecture of multi-FPGA systems deal with systems that are quite small, typically containing only dozens of FPGAs [11, 12, 13]. However, today's multi-FPGA systems are much bigger. For example, the Axis Xcite-2000 system consists of 10 FPGAs on a single board, and up to 24 such boards can be fitted into the slots of a SUN Ultra-SPARC machine, for a total of 240 FPGAs [1]. Also, none of the previous researches are specifically concerned with reducing the number of FPGAs to reconfigure.

3. PRELIMINARIES

In this section, we give some definitions used in the rest of this paper. A graph, $G = (V,E)$, for a direct interconnection multi-FPGA architecture is a graph-theoretic representation of the underlying interconnection in this system. The vertices or nodes V represent the FPGAs in the system and the edges E represent the dedicated interconnections between FPGAs. We also refer to the vertices of a graph G as $V(G)$ and its edges as $E(G)$. A capacity function $c : V \cup E \to R^+$ associates with each node a *node capacity* and each edge an *edge capacity*. The node capacities may be used to model the amount of logic resources (for example, the number of LUTs) the corresponding FPGA can accept comfortably, while the edge capacities can be used to model the number of connections that it can accept. Note that if techniques such as *virtual wire* [2] that multiplexes signals onto the wires are used, we can set edge capacities higher than the actual number of connecting wires, at the expense of longer emulation time. Figure 4 shows the device graph for the multi-FPGA system in the example of Figure 1(a). Typically, such mesh-styles also have *wraparound edges* to connect vertices on the two sides, but no wraparound edges between the top and bottom row. The wraparound edges are not shown in Figure 4. A graph $G' = (V',E')$ is a *subgraph* of a graph $G = (V,E)$ if $V' \subseteq V$ and $E' \subseteq E$.

The *length* of a path between two vertices is the number of edges in the path. The *cost* of the path is the total of node costs and edge costs used in the path. The *level* of a node in a rooted tree is the length from the root. We say that $u \leadsto v$, or that v is reachable from u, if and only if there is a path from node u to node v. The *lowest common ancestor* of two vertices u and v in a rooted tree (denoted $lca(u,v)$) is a vertex a such that $a \leadsto u$ and $a \leadsto v$ and such that the path length from a to u and a to v is the smallest possible (i.e., the length from the root to a is the *longest* possible). Note that the *lca* of any two vertices in a rooted tree is unique. Analogously, we define $lca(S)$, where $S \subseteq V$, to be the lowest vertex a such that $a \leadsto v$ for all $v \in S$. Again $lca(S)$ is unique in tree structures for a given set S.

A graph is *planar* if it can be drawn on a plane with no edges crossing. Such a drawing is known as a *planar embedding*. An embedding is *orthogonal* if the edges are drawn with only horizontal and vertical lines. A graph is a *grid graph* if the vertices are placed on a grid, and all edges of a node are to its neighbors on its immediate left, right, above or below in a geometric sense.

A *net* specifies the *source* and the *sinks* of a signal. A *netlist* is a collection of nets. To *route a net*, we specify the tree used to connect the source to every sink. Routing trees of different nets may share vertices, unlike in typical routing problems, since the vertices are FPGAs that can accommodate multiple nets. Even though a net may have more than one sink in an FPGA, this appears as only one sink in that FPGA in the reconfiguration problem because a signal entering the FPGA can be distributed *within* the FPGA. We refer to the set of such inter-FPGA netlists as *aggregated netlists*. Accordingly, the terms *two-terminal* and *multi-terminal* nets refer to the aggregated netlist, and not to the original netlist.

4. NET ADDITION PROBLEM

In this section, we describe the Net Addition Problem (NAP), and algorithms to solve this problem. We first give the definition of a classical graph problem,

DEFINITION 1. **(Node-Weighted Steiner Tree Problem) :** *Let $G = (V,E)$ be a graph with weights on each node v, w_v. Let $N \subseteq V$ be a subset of vertices of V. Find a tree $T_G(N)$ that connects all the vertices in N, using, optionally, some nets in $V - N$, and such that $w(T_G(N))$ is a minimum, where $w(T_G(N)) = \sum_{v \in V(T_G(N))} w_v$.*

The nodes in $T_G(N) - N$ are called *steiner points*. This problem is a generalization of the NP-complete *Steiner Tree Problem*, where the weights are on the edges, since we can always convert the standard problem to the node-weighted version by inserting a vertex into every edge and putting the weight of the edge onto the inserted vertex. Thus, the node-weighted problem is NP-complete. If we are only interested in minimizing the number of steiner points, this problem is known as the unweighted version. This problem is also NP-complete [6].

We now consider the first problem we have to handle during multi-FPGA reconfiguration. The problem is given in an abstract graph-theoretic formulation.

DEFINITION 2. **(Net Addition Problem) :** *Let $G = (V,E)$ be the device graph of a multi-FPGA system and let $c : V \cup E \to R^+$ be the capacity function of the nodes and edges. Let $N = \{N_1, \ldots, N_k\}$ be a set where N_i is a netlist for net i. Determine a* steiner forest, *i.e., a set of vertices $S = T_G(N_1) \cup T_G(N_2) \cup \ldots \cup T_G(N_k)$, such that $T_G(N_i)$ is a connecting tree of N_i for $1 \le i \le k$ and $w(S)$ is minimum, subject to node and edge capacity constraints.*

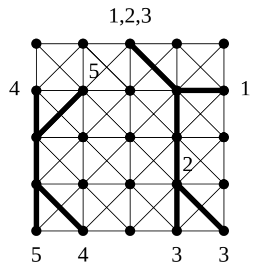


Figure 5: A rerouting example with 5 nets. The number represent the terminals of nets located inside the FPGA. A steiner forest, with sharing of vertices, is sought to connect all the terminals of each net. 13 FPGAs need to be reconfigured, of which 4 are steiner.

A connecting (steiner) tree $T_G(N_i)$ is a route of net i. Unlike usual routing problems where vertices must be *exclusively* used by nets, sharing of vertices between routes are allowed, since each vertex correspond to FPGAs. Figure 5 shows an example of NAP with five nets. The graph is an 8-way mesh, and the terminals of nets are labelled 1 through 5. Two distinct steiner trees with 13 vertices are used to connect the nets, with 4 vertices being steiner.

The regular Steiner Tree Problem (with edge weights) is a very well studied problem. However, there is much less study on the node-weighted Steiner tree problem, as it is harder than the standard problem. The Steiner Tree Problem can be approximated to within a constant factor of the optimal [22], but the node-weighted version cannot be approximated to less than a logarithmic factor unless $\tilde{P} \supseteq NP$, where $\tilde{P} = \text{DTIME}[n^{\text{polylog} n}]$ [14]. In [14], Klein and Ravi gave the first approximation algorithm for the node-weighted Steiner tree problem. It achieves an approximation ratio of $O(2\ln k)$, where k is the number of terminals to connect. Guha and Khuller then gave an algorithm that achieves an approximation ratio of $O(1.35\ln k)$ [6].

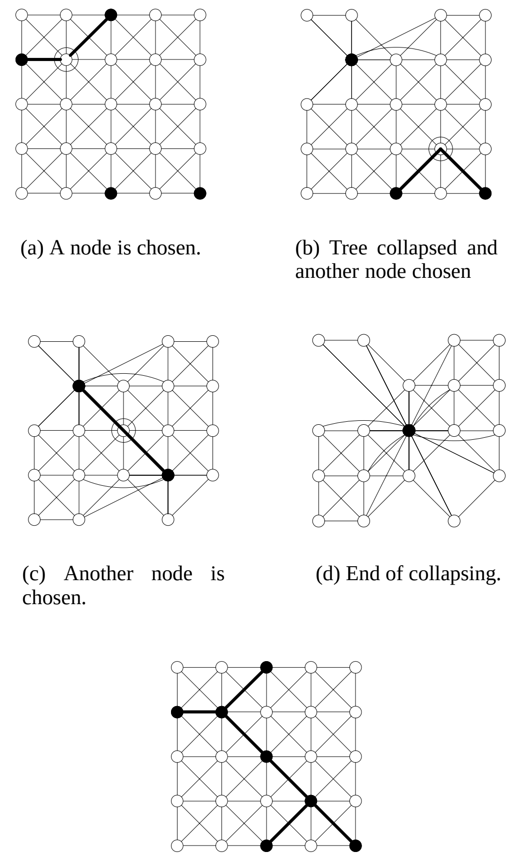
We now describe our algorithm `RouteOneNet` for routing a new net. Let T be the set of terminals in the net. Although the Guha-Khuller algorithm gives a slightly better theoretical bound on approximation, it requires repeated graph matching. This is not quite practical since we need to perform different placements and hence routing repeatedly. Our algorithm is a modification of the Klein-Ravi algorithm.

In each iteration, the algorithm scans through all the vertices of the tree and determines a node with small "average cost". As an example, consider the net in Figure 6(a). The node chosen is circled, together with the tree shown in thick lines. In the next step, the tree is collapsed into a terminal as shown in Figure 6(b). Another node is then chosen and another tree collapsed in Figure 6(c). The process is then repeated until one terminal is left. At this point, the tree is reconstructed as shown in Figure 6(e).

To choose the node and the tree to collapse, we compute the cost as follows : let t_j be the terminals of a net to be connected, and let $d(v,t_j)$ be the minimum weighted distance from v to t_j. Assume that the terminals are sorted in order of $d(v,t_j)$. This node has cost :

$$c_{vi} = \frac{w_v + \sum_{j=1}^{i} d(v,t_j)}{i} \text{ for } i = 2, \ldots, |T|.$$

We choose v and i with the smallest c_{vi} value. To ensure that v is not too far from the remaining terminals, we also compute the average distance, x_{vi} of the remaining terminals from the source s and also the average distance of these nodes from the node v, y_{vi}.



(a) A node is chosen.

(b) Tree collapsed and another node chosen

(c) Another node is chosen.

(d) End of collapsing.

(e) Tree reconstructed.

Figure 6: Illustration of algorithm

Then we pick the smaller of these two and add to $c_{vi} + d(s,v)$ and use this modified cost as our new cost, i.e.,

$$c'_{vi} = c_{vi} + \min\{x_{vi}, y_{vi}\} + d(s,v), \text{ for } i = 2, \ldots, |T|.$$

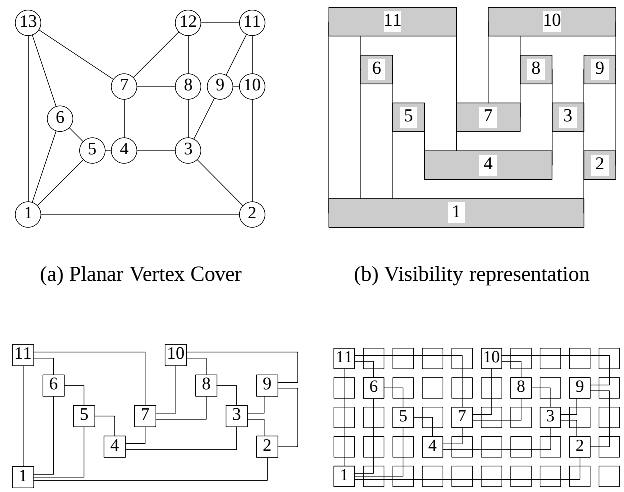
Our rationale is as follows. Since the remaining terminals are to be connected with this tree, we estimate the cost taken to connect the remaining terminals. If the remaining nodes are, on average, close to the source, then these should be connected to the source. Otherwise, the remaining nodes should be connected to v. Hence, our choice of quotient cost. Clearly, $(|T| - i).\min\{x_{vi}, y_{vi}\} + d(s,v)$ is an upper bound on the cost to connect the remaining trees. Note that this algorithm is not the same as maze routing. In maze routing, the least cost path is found iteratively as the algorithm considers one terminal at a time to connect to the partial tree constructed, starting with the source. This does not always produce a good steiner tree. In our algorithm, great care is taken to produce good steiner trees. In particular, the Klein-Ravi algorithm produces provably good trees.

We can perform routing of all new nets by iteratively routing one net at a time with `RouteOneNet`. Once a net is routed, the nodes that the net passes through will have its weight reduced by a certain amount. When routing subsequent nets, the nets will be encouraged to use those vertices that have already been used. In fact, no further cost is incurred by routing through such vertices, since a one time cost has already been paid for.

Algorithm `RouteOneNet` **(terminals** T**)**

1. **while** $|T| \geq 1$ **do**
   begin
2. compute $d(v,t)$ where $v \in V$ and $t \in T$
3. **for** each $v \in V$ **do**
   begin
4. order T into $t_1, \ldots t_{|T|}$ in non-increasing order of $d(v,t_i)$
5. compute c'_{vi} for $i = 2$ to $|T|$
6. pick i such that c'_{vi} is minimum,
   end
7. pick (v,i) pair such that c'_{vi} is minimum
8. collapse the tree formed by $v \rightsquigarrow t_1, \ldots, v \rightsquigarrow t_i$ into v' where $v \rightsquigarrow t_j$ is the path used to compute $d(v,t_j)$
9. $T = T - \{t_1, \ldots, t_i\} \cup \{v'\}$
   end
10. reconstruct routing tree

Figure 7: Algorithm `RouteOneNet`



(a) Planar Vertex Cover

(b) Visibility representation

(c) Orthogonalized representation

(d) Net Deletion Problem

Figure 8: Proof that NDP is NP-complete

5. NET DELETION PROBLEM

Another common problem in the multi-FPGA reconfiguration problem is the Net Deletion Problem. As illustrated in Figure 3, we are given a set of routes of nets to be deleted. We want to choose a small set of FPGAs to reconfigure, so that all connections between the terminals of each net are broken. This problem can be simply stated as follows :

DEFINITION 3. *Given a graph G representing the connectivity of the FPGAs, and a set R of net routings on this graph, find a minimum cardinality (or weighted) set of vertices whose removal disconnects all pairs of terminals of each net in R.*

This set is known as the *breaking set* of (G,R). We now show that this problem is NP-complete if the underlying graph is a grid graph. Since the grid graph is one of the simplest implementation of the underlying connectivity of multi-FPGA systems, this means that the problem on other more general and interesting graphs like mesh, planar, etc., are also NP-complete. However, we show that if the underlying graph is a tree, the problem is polynomial time solvable.

We give the following definition and then state a well-known result which will be needed in the proof of Theorem 2 :

DEFINITION 4. **(Vertex Cover Problem) :** *Given a graph $G = (V,E)$, find a minimum cardinality set $C \subseteq V$ such that for each edge $(u,v) \in E$, either $u \in C$ or $v \in C$.*

THEOREM 1. *Vertex Cover for planar graphs (PVC) is NP-complete [5].*

THEOREM 2. *The Net Deletion Problem over a grid graph is NP-complete.*

Proof: NDP is in NP since given a breaking set B for an NDP over a grid graph G, we can easily check in polynomial time if for each routing tree, $R_i \in R$, whether for every pair of terminals, u and v, there exists a vertex $w \in B$ such that w lies on the unique path between u and v in R_i.

We next show that NDP is NP-complete by reducing PVC to NDP over a grid graph. Given a PVC problem $G = (V,E)$, we construct an instance of the NDP problem (G',R'), where $G' = (V',E')$, by creating an orthogonal embedding of G such that the vertices are placed on a grid, and each edge becomes transformed to a path with orthogonal edges.

To do this, we first find the *visibility representation* of G. This is a geometric diagram where each vertex is mapped to a horizontal rectangle and each edge is mapped to a vertical line, only touching the two rectangles representing the vertices of its endpoints. Thus, no vertical line cut across any rectangle and no two vertical lines coincide. The algorithm of Kant [9] computes this representation in linear time using an area at most $O(|V|) \times O(|V|)$. Figure 8(a) shows a planar graph and Figure 8(b) shows a corresponding visibility representation. We create an orthogonal embedding of G by shrinking each horizontal rectangle into a point on the left side of each rectangle. For each vertical connection originally touching the rectangle, a horizontal segment is added to connect to the new point. This causes each path to have at most two bends. Figure 8(c) shows the shrinking. For example, consider the edge (2,9) in Figure 8(b). Since the rectangles 2 and 9 are shrunk, the edge (2,9) now becomes a "⊐" shaped *orthogonal path* as shown in Figure 8(c). This stage requires polynomial time since each path have length at most $O(3|V|)$ in the orthogonal embedding. Once this is done, we simply add additional vertices representing the FPGAs to fill the entire grid. At most $O(|V|^2)$ extra vertices are needed. Hence, the transformation takes polynomial time.

By the way we constructed the horizontal segments, all turns are always to the left, i.e., "⌉" or "⌋" shaped turns only. "∟" and "⌈" shaped turns are not possible. Neither can there be any crossing of paths, i.e., "+", since by definition of the visibility representation, no vertical edge can cross a rectangle. Also, whenever an orthogonal path turns left at an FPGA, other orthogonal paths passing through that vertex must be horizontal. It is also easy to see that all orthogonal paths connect to only two endpoints (i.e., all nets are two-terminal).

Let $f : V \to V'$ be a function that correlates a vertex in V with a vertex in V', such that $v' = f(v)$ if v' in the orthogonal embedding is obtained by shrinking the rectangle representing v in the visibility

```
Algorithm BreakRoute (route R)
1.  S = the vertices of route R
          sorted in reverse topological order
2.  T = terminals of R
3.  B := ∅
4.  for each vertex v in S do
    begin
5.      if v already marked for reconfiguration
        or is an lca of any two t1, t2 in T then
        begin
6.          B := B ∪ {v}
7.          T = T − {t | t ∈ T and t is a desc. of v}
        end
    end
```

Figure 9: Algorithm `BreakRoute`

representation. Conversely, $f^{-1}(v') = v$ if v' is the corresponding vertex of v in the orthogonal embedding, and $f^{-1}(v')$ is undefined if it was a vertex added at the last stage to fill the grid. Also, we define $f(S) = \{f(v) \mid v \in S\}$. For each edge $(u,v) \in E$, we define the corresponding orthogonal path in G' as $P_{f(u)f(v)}$. Let G' be the grid graph created above and let $R' = \{P_{f(u)f(v)} \mid (u,v) \in E\}$. Therefore, the instance of NDP we create is (G', R'). We now show that G has a vertex cover of size k if and only if (G', R') has a breaking set of size k.

Suppose G has a vertex cover C of size k. Then for every edge (u,v) in G, either $u \in C$ or $v \in C$. Let $C' = f(C)$. Then in G', for every path $P_{f(u)f(v)} \in R'$, either $f(u) \in C'$ or $f(v) \in C'$. Thus, the vertices in C' breaks all the paths in R'. Hence C' forms a breaking set of (G', R'), and $|C'| = |C|$.

Now suppose that B forms a breaking set of (G', R') of size k. We first show that there exists a breaking set B' such that for each vertex $v' \in B'$, $f^{-1}(v') \in V$. Suppose for some $v' \in B'$, $f^{-1}(v')$ is undefined, then v' lies completely within an orthogonal path P, i.e., v' is not at the end point. If v' lies on a horizontal segment, then we can follow P to the left until it hits an endpoint. Note that it is not possible for a concurrent path to leave this horizontal segment because all turnings are to the left. If v' lies on a vertical segment, then v' breaks only P since no vertical segment overlap. We now follow P down till it hits an endpoint of the path, or it turns left. Then the situation is the same as above. Let the endpoint reached be v''. In either case, it is clear that v'' breaks the path(s) it originally breaks (in fact, it may break more paths). Therefore, the set $B - \{v'\} \cup \{v''\}$ also forms a breaking set with the same cardinality. Applying the above repeatedly, we can find a breaking set B' such that for each vertex $v' \in B'$, $f^{-1}(v') \in V$, and $|B'| = |B| = k$.

Since B' is a breaking set of (G', R') such that $B' \subseteq f(V)$, then for each path $P_{u'v'} \in R'$, either $u' \in B'$ or $v' \in B'$. Let $C = \{f^{-1}(v') \mid v' \in B'\}$. Then, for each edge $(f^{-1}(u'), f^{-1}(v')) \in E$, either $f^{-1}(u') \in C$ or $f^{-1}(v') \in C$. Hence, C forms a vertex cover of G and $|C| = k$. $\square$

COROLLARY 2.1. *NDP is NP-complete over mesh and planar topologies.*

THEOREM 3. *NDP is polynomial-time solvable for a single net.*

Proof: Consider algorithm `BreakRoute` in Figure 9. Let B be the breaking set obtained from the algorithm and suppose B' is a breaking set such that $|B'| < |B|$. We arrange the elements of B and B' in the order in which the vertices are processed. Let x be the first vertex in this order that is in $B' - B$. Let x break the path between two remaining terminals t_1 and t_2, where t_1 and t_2 have the lowest *lca*. Clearly, $lca(t_1,t_2)$ breaks the path between t_1 and t_2 and that $lca(t_1,t_2) \in B$. Then either t_1 or t_2 is a descendent of x, otherwise x can never break the path between t_1 and t_2. Without loss of generality, assume that t_1 is a descendent of x. If $x = lca(t_1,t_2)$, then x will be found by the algorithm since it is the lowest *lca*, contradicting our assumption. Hence $x \neq lca(t_1,t_2)$. In this case, if $lca(t_1,t_2)$ is a descendent of x, then x cannot break the path between t_1 and t_2, so $lca(t_1,t_2)$ must be an ancestor of x. However, if we replace x with $lca(t_1,t_2)$, then $lca(t_1,t_2)$ breaks the path between t_1 and t_2, and we still get a breaking set with the same cardinality. Notice that $lca(t_1,t_2)$ is what will be found by the algorithm. Hence, if we perform the above replacement repeatedly, we get a breaking set that is the same as that found by the algorithm, yet with a smaller cardinality, leading to a contradiction. Hence, the algorithm gives us the optimal solution.

The algorithm can be made to run in linear time as follows : it traverses R in depth-first order. At the leaves (terminals), the number of descendent terminals to break, d, is set to 1. As it backups the tree, it collates d from its children. This is the number of terminals it needs to break at the node. If $d > 1$, then the node is an *lca* of two of the remaining terminals and can be marked for reconfiguration. d is then set to zero and passed to its parent. $\square$

The algorithm `BreakRoute` can "break" one net optimally. It can be generalized to the algorithm `Break` which can simultaneously "break" all nets if the union of all the nets is a tree. Optimality of `Break` will be presented in the proof of the next theorem. The algorithm `Break` is the basic procedure to solve the general NDP. We iteratively look for a maximal set of nets such that their union is a tree and apply Break to delete the nets in the set.

THEOREM 4. *NDP is polynomial-time solvable for tree topology.*

Proof: The proof is similar to that of the algorithm `BreakRoute`. Let B be the breaking set obtained from algorithm `Break` and suppose B' is a breaking set such that $|B'| < |B|$. We arrange the elements of B and B' in the reverse topological order in which the vertices are processed. Let x be the first vertex in this order that is in $B' - B$. Let x break the path between t_1 and t_2, where t_1 and t_2 are terminals belonging to route R_i and have the lowest *lca*. Without loss of generality, assume that t_1 is a descendent of x, but t_2 is not. Let y be the nearest ancestor of x that is an *lca* of any two remaining terminals of any net (note that y may or may not be $lca(t_1,t_2)$). Then clearly, $lca(t_1,t_2)$ is either y or an ancestor of y. If we replace x with y, then y breaks the path between t_1 and t_2, and we still get a breaking set with the same cardinality. Notice that y is what will be found by the algorithm. Hence, if we perform the above replacement repeatedly, we get a breaking set that is the same as that found by the algorithm, yet with a smaller cardinality, leading to a contradiction. Hence, the algorithm gives us the optimal solution.

This algorithm runs in polynomial time. It keeps track of the number of descendent terminals, $d[i]$, it needs to break for each net i. At the terminal of a net i, $d[i]$ is set to 1. As it backups the tree, the number is collated from its children. If $d[j] > 1$ for some j, then it is an *lca* of two remaining terminals of net j. This node is then marked for reconfiguration, and all $d[i]$ is set to zero. Since all the updates are local, it runs in polynomial time. $\square$

```
Algorithm Break(NDP (G,N))
1.  S = the vertices of G
          sorted in reverse topological order
2.  T_i = terminals of R_i
3.  B := ∅
4.  for each vertex v in S do
    begin
5.        if v already mark for reconfiguration
          or v = lca(x,y), where x,y ∈ T_i, for some i, then
          begin
6.              B := B ∪ {v}
7.              T_j = T_j − {t | t ∈ T_j and t is a desc. of v}
          end
    end
```

Figure 10: Algorithm `Break`

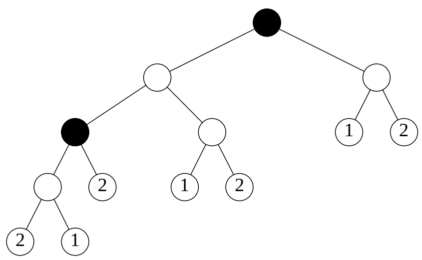


Figure 11: Illustration of Algorithm `Break`**. The dark vertices form a breaking set. There exist a dark vertex between any two pairs of terminals belonging to any net.**

6. INCREMENTAL RECONFIGURATION PROBLEM

The multi-FPGA incremental reconfiguration problem involves the modification of a circuit already implemented on the system. Abstractly, we are given the device graph G, a circuit H with some nodes already placed (F) and the rest to be placed and routed on G, and also a set of routes R to be disconnected. The objective is to determine a placement and routing of the new circuit such that the number of FPGAs to reconfigure is minimum, subject to node and edge capacity constraints. The set of circuit nodes F that have fixed positions are the connections to the existing circuit.

Clearly, the combined problem is also NP-complete. We solve this problem problem using simulated annealing. During each simulated annealing move, it solves the NAP followed by the NDP. This is because the NAP potentially uses more FPGAs in order to route, and this can provide FPGAs for nets to be broken without incurring additional FPGAs to reconfigure. The NAP is solved by routing each net using our algorithm `RouteOneNet`, reducing the weights of vertices used by another net. The NDP is solved by running Algorithm `Break` on the routes. The cost evaluated during the simulated annealing algorithm is an aggregate cost of the number of FPGAs to reconfigure (n), the total length of the routes (l) and the maximum overflow (f). The cost function used is

$$c = \alpha n + \beta l + \gamma f^2.$$

Since the capacity constraints model the ability of the FPGA software to reconfigure the FPGA, more overflow implies that the FPGA is harder to reconfigure, therefore we make it a quadratic term to allow for small overflows, but heavily penalize large overflows. In most FPGAs, a utilization below 70% implies that the FPGA should be quite easily placed and routed. Complete placement and routing becomes increasingly harder as utilization grows beyond 80%. The overflow can also be changed to $(f+k)^2$ for some big enough $k > 0$ to completely discourage overflow. Then as the utilization approaches f, the penalty increases. The coefficients are used to adjust the relative importance of the different variables. In our experiments, we used $\alpha = 0.99$, $\beta = 0.001$ and $\gamma = 0.009$.

7. EXPERIMENTAL RESULTS

To design some meaningful test cases, we use the suite of MCNC benchmark circuits as our starting point. Each circuit was treated as a new sub-circuit to be placed and routed on a multi-FPGA system with a circuit already implemented on it. Each node in the circuit represents a 4-LUT (the most popular logic block among commercial FPGA vendors) in order to get reasonably sized circuits for our experiments. We randomly generated a small set of fixed placement for the I/O nodes and some of the internal nodes. The fixed nodes formed about 30% of the total nodes, and the rest are determined by the algorithm. To simulate the fact that an existing circuit is already implemented on the multi-FPGA system, we randomly generate node capacities for each FPGA using the uniform distribution with a mean of 100 4-LUTs for most circuits and 200 4-LUTs for the larger circuits. The edge capacities are not taken into account (as in the case where virtual wires are used). However, it is a simple extension to take edge capacities into account in the case where virtual wires are not used. We assume that, for each FPGA a net passes through, it uses up some η unit of the FPGA resources. We used $\eta = 1$ in our experiments. Note that η can be non-uniform. The topology graph used is the 8-way mesh with horizontal wraparound.

For comparison purposes, we also implemented a method using shortest path as the algorithm for connecting nets. The nets are routed in turn and the cost of FPGAs that have already been used are set to zero. In other words, the algorithm will try to reuse FPGAs that already need to be reconfigured. The deletion algorithm is the same in both our steiner-tree based algorithm and the shortest path based algorithm. The results are summarized in Table 1. For example, for the circuit `Ex1` the entry using shortest path is 43/521/1, which means that it requires reconfiguration of 43 FPGAs, a total length of 521 units and a maximum overflow of 1. As can be seen, the average improvements in the number of FPGAs to be reconfigured is about 15%, while some improvements of 20-30% are also seen. Since reconfiguring each FPGA can take hours, this reduction can translate to significant savings in overall reconfiguration time.

8. CONCLUDING REMARKS

In this paper, we formulated the problem for reconfiguration in multi-FPGA systems that uses the direct connection architecture. Our objective is to reduce the number of FPGAs to reconfigure in order to reduce the time needed for such reconfiguration. We formulated the Net Addition Problem and the Net Deletion Problem for this architecture, and showed that the Net Addition Problem is a generalization of the NP-complete Steiner Tree Problem and is therefore NP-complete. We also proved that the Net Deletion Problem is NP-complete over grid graphs, mesh graphs and also planar graphs. However, we prove that it is polynomial-time solvable for tree architectures. We gave algorithms to solve both problems and

circuit	size $(\lvert V(H)\rvert, \lvert N\rvert, \lvert F\rvert)$	shortest path $(n/l/f)$	ours $(n/l/f)$	% imp. on n
Ex1	(107,105,38)	43/521/1	37/718/0	14.0
Ex2	(116,83,74)	35/570/0	30/378/0	14.3
Ex3	(179,153,59)	54/996/0	37/923/0	31.5
Ex4	(482,460,134)	85/3767/0	79/3628/0	7.1
Ex5	(161,153,47)	42/947/0	39/921/0	7.1
Ex6	(116,83,74)	43/507/0	38/339/0	11.6
Ex7	(779,682,302)	50/1821/0	43/1540/0	14.0
Ex8	(235,209,87)	56/1094/0	38/1056/0	32.1
Ex9	(208,202,67)	56/1397/0	41/1493/0	26.8
Ex10	(490,487,157)	72/4030/0	73/3703/0	-1.4
Ex11	(274,247,79)	57/1671/0	44/1802/0	22.8
Ex12	(340,501,170)	30/567/0	30/562/0	–
Ex13	(1073,1027,325)	59/8212/0	44/8570/0	25.4
Ex14	(1253,1061,482)	71/6959/0	49/7251/0	31.0
Ex15	(565,530,156)	38/1469/0	37/1383/0	2.7
Ex16	(821,683,311)	49/2057/0	41/1724/0	16.3
Ex18	(166,161,54)	47/1088/0	38/1315/0	19.1
Ex17	(443,340,243)	64/2959/0	47/3220/1	26.6
Ex19	(231,229,74)	35/744/0	36/548/0	-2.9
Ex20	(495,480,158)	36/1013/0	35/1238/0	2.8
Ex21	(503,487,158)	79/4572/0	80/4362/0	-1.3
Ex22	(123,112,45)	53/597/0	43/653/0	18.9
Ex23	(226,222,64)	57/1611/0	44/1602/0	22.8
Ex24	(100,79,46)	38/477/0	35/396/0	7.9
Ex25	(309,275,90)	54/2067/5	49/2130/0	9.3
avg	–	1303	1107	15.0

Table 1: Reconfiguration results for test cases. Each test case is a sub-circuit to be placed and routed on a multi-FPGA system with a circuit already implemented on it. Each node in this sub-circuit represents an LUT. The shortest-path based net addition algorithm is a simpler and reasonable alternative to `RouteOneNet`**. A set of nets is also randomly generated to simulate nets to be deleted.**

presented a simulated annealing approach that solves both problems in a reconfiguration setting. The number of FPGAs needed for reconfiguration in our approach is about 15% less than using a more direct shortest path approach.

As a future research, the problems investigated in this paper should also be investigated for the indirect connection architecture. As the architecture is very different than that considered in this paper, a totally different approach is probably necessary. Also, this problem should also be studied for hybrid architectures that combine good features of the different architectures.

9. REFERENCES

[1] Axis Corporation. "Xcite-2000 datasheet". http://www.axiscorp.com/products/Xcite2000.html, 1999.

[2] J. Babb, R. Tessier, M. Dahl, S. Hanono, D. Hoki, and A. Agarwal. "Logic Emulation with Virtual Wires". *IEEE Transactions on Computer Aided Design*, 16(6):609–626, June 1997.

[3] N. W. Bergmann and J. C. Mudge. "Comparing the Performance of FPGA-Based Custom Computers with General-Purpose Computers for DSP Applications". In *Proceedings FPGA Symposium*, pages 164–171, 1994.

[4] M. Chrobak and T. H. Payne. "A Linear-time Algorithm for Drawing a Planer Graph on a Grid". In *TR UCR-CS-90-2, Department of Math and Computer Sciences., University of California at Riverside*, 1990.

[5] M. R. Garey and D. S. Johnson. *"Computers and Intractability"*. W. H. Feeman & Company, 1979.

[6] S. Guha and S. Khuller. "Improved Methods for Approximating Node Weighted Steiner Trees and Connected Dominating Sets". *Information and Computation 150*, pages 57–74, 1999.

[7] S. Hauck, G. Borriello, and C. Ebeling. "Springbok: A Rapid-Prototyping System for Board-Level Designs". In *Proceedings FPGA Symposium*, pages 170–177, October 1994.

[8] Ikos Systems, Inc. "VirtuaLogic Emulation System Manual". Feb 1997.

[9] G. Kant. "Drawing Planar Graphs Using the Canonical Ordering". In *Proceedings Foundation of Computer Science*, 1992.

[10] T. Kean and I. Buchanan. "The Use of FPGAs in a Novel Computing Subsystem". In *Proceedings FPGA Symposium*, pages 60–66, 1992.

[11] M. A. S. Khalid and J. Rose. "The Effect of Fixed I/O Positioning on The Routability and Speed of FPGAs". In *Canadian Workshop on Field Programmable Devices*, pages 94–102, 1995.

[12] M. A. S. Khalid and J. Rose. "Experimental Evaluation of Mesh and Partial Crossbar Routing Architecture for Multi-FPGA Systems". In *International Workshop on Logic and Architecture Synthesis*, pages 119–127, December 1997.

[13] M. A. S. Khalid and J. Rose. "A Hybrid Complete-Graph Partial-Crossbar Routing Architecture for Multi-FPGA Systems". In *Proceedings FPGA Symposium*, pages 45–54, February 1998.

[14] P. Klein and R. Ravi. "A Nearly Best-possible Approximation Algorithm for Node-weighted Steiner Trees". In *Proceedings Integer Programming and Combinatorial Optimization*, pages 323–331, 1993.

[15] D. Matthew, B. Jonathan, R. Tessier, S. Hanono, D. Hoki, and A. Agarwal. "Emulation of the sparcle microprocessor with the MIT virtual wires emulation system". In *Proceedings FPGA Symposium*, pages 14–22, 1994.

[16] Quickturn. "Mercury Technology Backgrounder". http://www.quickturn.com/products/mercury_backgrounder.htm, 2000.

[17] P. Shaw and G. Milne. "A Highly Parallel FPGA-based Machine and its Formal Verification". *Lecture Notes in Comp. Sc. 705 - Field-Programmable Gate Arrays : Architectures and Tools for Rapid Prototyping*, pages 162–173, 1993.

[18] R. Tamassia, I. G. Tollis, and J. S. Vitter. "Lower Bounds for Planar Orthogonal Drawings of Graphs". *Information Processing Letters 39*, pages 35–40, 1991.

[19] R. Tessier. "Incremental Compilation of Logic Emulation". In *Proceedings Rapid System Prototyping*, pages 236–241, 1999.

[20] K. Yamada, H. Nakada, A. Tsutsui, and N. Ohta. "High-Speed Emulation of Communication Circuits on a Multiple-FPGA system". In *Proceedings FPGA Symposium*, 1994.

[21] J. Varghese, M. Butts, and J. Batcheller. "An Efficient Logic Emulation System". In *IEEE Transactions on VLSI Systems*, 1(2), Jun. 1993.

[22] A. Zelikovsky. "An 11/6-approximation algorithm for the network Steiner problem". *Algorithmica*, 9:463–470, 1993.