ARTICLES

Journal of Semiconductors
(2020) 41, 022405
doi: 10.1088/1674-4926/41/2/022405

A routing algorithm for FPGAs with time-multiplexed interconnects

Ruiqi Luo[1, 2, 3], Xiaolei Chen[4], and Yajun Ha[1, †]

[1]School of Information Science and Technology, ShanghaiTech University, Shanghai 201210, China

[2]Shanghai Institute of Microsystem and Information Technology, Chinese Academy of Sciences, Shanghai 200050, China

[3]University of Chinese Academy of Sciences, Beijing 100049, China

[4]Intel Singapore, Singapore 339510, Singapore

Abstract: Previous studies show that interconnects occupy a large portion of the timing budget and area in FPGAs. In this work, we propose a time-multiplexing technique on FPGA interconnects. In order to fully exploit this interconnect architecture, we propose a time-multiplexed routing algorithm that can actively identify qualified nets and schedule them to multiplexable wires. We validate the algorithm by using the router to implement 20 benchmark circuits to time-multiplexed FPGAs. We achieve a 38% smaller minimum channel width and 3.8% smaller circuit critical path delay compared with the state-of-the-art architecture router when a wire can be time-multiplexed six times in a cycle.

Key words: field programmable gate arrays; digital integrated circuits; routing algorithm design and analysis; digital integrated circuits

Citation: R Q Luo, X L Chen, and Y J Ha, A routing algorithm for FPGAs with time-multiplexed interconnects[J]. *J. Semicond.*, 2020, 41(2), 022405. http://doi.org/10.1088/1674-4926/41/2/022405

1. Introduction

Interconnect and logic resources can be seen as the two significant parts of FPGAs. Logic blocks are used to implement a user design. Interconnect resources are designed to achieve connections among logic blocks. In FPGAs, logic resources are always organized as arrays of blocks. Interconnect resources are routing switches and wires grouped into channels. FPGA interconnects (or interconnection network) can be thought of as a programmable network of signal paths among FPGA logic resource ports.

Both measurements and analyses indicate that the programmable interconnections contribute the most to the FPGA area, latency, and power consumption. In order to achieve high routability with reduced routing, FPGA vendors use the substantial on-chip area to route programmable switches and wires[1]. This is why interconnects out-weight logic in terms of power budget and area. FPGA interconnects are slow because they need to traverse a series of tracks connected by switches to route among different logic blocks. Optimizing the interconnect network is critical because it has a profound impact on the performance of FPGAs.

Previous work by Trimberger *et al.*[2] has illustrated that the utilization of logic resources in FPGA can be improved by time-multiplexing, which inspires us to apply time-multiplexing to FPGA interconnection networks and leads to our new architecture and routing algorithm that we will introduce in this paper. We notice that most wires are only used for a short period in a clock cycle. The delay of wire for a signal is only a small portion in a clock cycle. So we can better utilize the interconnect resources by time-multiplexing signals. It can also improve the FPGA performance at a small cost of the multiplexing circuitry complexity.

FPGAs with time-multiplexed interconnects require specialized electric design automation (EDA) algorithms and tools to support them. Existing algorithms and tools cannot be reused in their present forms because they are not multiplexing-aware. In this paper, we propose a time-multiplexing-aware routing algorithm. This algorithm is similar to VPR routing algorithm[3]. Unlike the VPR algorithm, our algorithm can add the time-multiplex feature to the algorithm and implement it. In order to achieve this feature, we also design TM switches which are described in an architecture file. We define a bitmap for signals in our design, which is a new technique and can add the scheduling capability to the VPR algorithm. This algorithm can be seen as timing-driven because it has a delay-sensitive term in its multiplexing-aware congestion cost function.

We also validate our proposed routing algorithm experimentally. We evaluate our TM-ARCH FPGAs by using our TM-router to replace the conventional VPR router in a standard FPGA EDA flow. We also use the VPR 5 router[4] as a comparison on the same set of benchmark circuits with conventional architecture. Experimental results reveal that we only use 65% average minimum channel width of conventional architectures on average and can achieve 3.8% smaller circuit critical path delay on average.

The rest of this paper is as follows. Section 2 introduces existing FPGA architectures and their EDA algorithms. Section 3 briefly introduces TM-ARCH architecture. Section 4 proposes our time-multiplexing-aware routing algorithm. Section 5 introduces what we have done to validate this algorithm, and

Correspondence to: Y J Ha, hayj@shanghaitech.edu.cn
Received 6 DECEMBER 2019; Revised 31 DECEMBER 2019.

also presents experimental results. Finally, Section 6 concludes this paper.

2. Related work

Previous work on time-multiplexed FPGA[2] has been done. Trimberger *et al.* propose both time-multiplexed interconnects and configurable logic blocks (CLBs) with the Xilinx XC4000E FPGA family. To facilitate mapping user designs into time-multiplexed architecture, a scheduling algorithm is presented[5] that is based on list scheduling. Lin *et al.* also apply time-multiplexing to Xilinx 4000 architecture FPGA[6], but they only time-multiplex routing resources. Even so, they still have achieved 30% reduction of channel density. Francis *et al.*[7] apply time-multiplexed interconnects to an Altera FPGA, which is similar to our architecture proposed in this paper. Only interconnects are time-multiplexed in ours, Lin's and Francis' work, while both logic blocks and interconnects are time-multiplexed in Trimberger's architecture. In terms of the supporting EDA flow, Trimberger's work can be considered as close to ours in both works, time-multiplexing is scheduled after technology mapping and before placement. In our work, time-multiplexing is scheduled during routing, while Francis' work does it after routing.

Shen *et al.* present a serial-equivalent static and dynamic parallel router[8]. This router provides a significant speedup in routing and can achieve the same result as the serial router. They also try box expansion and grid partitioning to speed up parallel FPGA routing[9]. Shen *et al.* use parallel recursive partitioning to accelerate FPGA routing that can achieve 7.06× speed up using 32 processers[10]. They propose a GPU-based parallel routing algorithm[11] and achieved 1.57× speedup improvement than VPR router. Based on PathFinder kernel, they develop Raparo, which is an angle based region partitioning[12]. Results show that it can provide 16× speedup when scaling to 32 processor cores. They also exploit strictly-ordered partitioning in parallelizing FPGA routing called Megrez and achieve 15.13× speed up on GPU without influence on quality[13].

Vercruyce *et al.* propose CRoute[14], which is a connection-based timing-driven router. This routing algorithm can reduce total wire-length and increase the maximum clock frequency at the same time. Wang *et al.* propose a new approach[15] based on the PathFinder algorithm. They only reroute illegal paths during each routing iteration and can reduce 68.5% routing runtime on average. Patil *et al.* use a heuristic method in routing for hybrid FPGA networks[16]. They schedule data streams efficiently to increase the bandwidth and then achieve 11% stream bandwidth improvement on five benchmark circuits. Chaplygin *et al.* propose an FPGA routing block optimization with a given number of trace signals[17]. Farooq *et al.* apply the time-multiplexing technology to multi-FPGA prototyping routing for more complicated designs[18]. Omam *et al.* use RRAM-based switches and decrease 56% path delay compared to CMOS base switches[19]. Chen *et al.* explore state-of-the-art research directions for FPGA placement and routing[20]. Huriaux *et al.* evaluate the routing of I/O signals of large applications through column interfaces in embedded FPGA fabrics[21]. Kashif *et al.* compare a completely connected graph and time-multiplexing nets on multi-FPGA system[22]. Results show that completely connected graphs can achieve higher performance and time-multiplexing can provide a better cost/performance ratio.

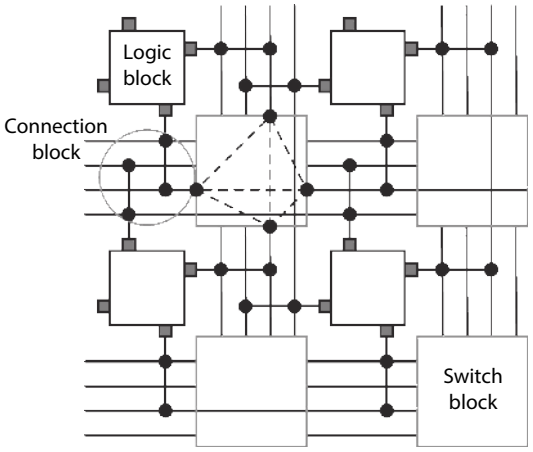


Fig. 1. Island-style architecture, which is the base of TM-ARCH with the time-multiplexed interconnects.

3. Target FPGA architecture

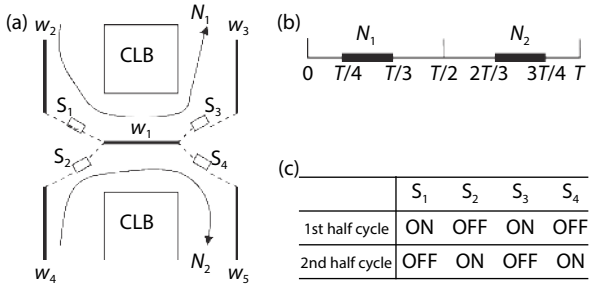
Fig. 1 illustrates an island-style FPGA, which is the base of our target time-multiplexed FPGA architecture. The only difference between this island-style architecture and ours is that all wires in our design can be time-multiplexed. In the conventional island-style FPGAs, vertical and horizontal routing channels surround logic blocks from all four sides containing multiple tracks. We define channel width as the total number of tracks in a channel. Every track has multiple wire segments. A switch block is arranged at every intersection of a vertical channel and a horizontal channel. Programmable switches are placed in switch blocks and connection blocks to implement configurable routing. Logic block pins are connected to routing channels through connection blocks.

3.1. Clock cycle and microcycle

In our new architecture, we divide a clock cycle into multiple microcycles. Different signals can occupy the same wire if this wire is multiplexable, and they can use the same wire at different microcycles in a clock cycle. We use a similar definition of microcycle and clock cycle as in Trimberger *et al.*'s work[2]. A clock cycle is constrained by the critical path delay. We defined M_c, which means a clock cycle has M_c microcycles.

3.2. Time-multiplexed wires

A route of nets is usually made up of many segments of wires. One segment of a wire can only be occupied between the time that the signal arrives and leaves in one clock cycle. In the TM-ARCH FPGAs, different signals can occupy the same segment of wires at different time if this wire segment can be multiplexed. Fig. 2(a) illustrates this. We first define that one clock cycle consists of two microcycles in this TM-ARCH architecture device. w_1 is multiplexable which is contained in both net N_1 and N_2. This condition is permitted because N_1 and N_2 can use the same wire segment at different microcycles in one clock cycle. N_1 occupies this wire at $[T/4, T/3]$, which is in the first microcycle. N_2 occupies the wire segment at $[2T/3, 3T/4]$ in the second microcycle. Fig. 2(b) describes the time intervals in which N_1 and N_2 use the wire. T means the period of a clock cycle.



	S_1	S_2	S_3	S_4
1st half cycle	ON	OFF	ON	OFF
2nd half cycle	OFF	ON	OFF	ON

Fig. 2. (a) Signals N_1 and N_2 occupy a wire at different time. (b) In time domain, N_1 and N_2 do not overlap. (c) TM switches' different states.

3.3. TM switch

A time-multiplexing switch (TM switch) is the key to implement time-multiplexing at the hardware level. Compared with conventional FPGAs, a TM switch has two more features: latching data and associating with multiple contexts.

3.3.1. Multiple contexts

In TM-ARCH, a TM switch can have at most M_c contexts. As time passes, the state of TM switch changes among the M_c contexts. This unique feature helps achieve the time-multiplexing of wires in our architecture. Fig. 2(c) shows an example of different states of TM switches (S_1, S_2, S_3, and S_4). This example can be time-multiplexed in the architecture shown in Fig. 2(a). In this condition, the TM switches can have two contexts, and change between them in one clock cycle.

3.3.2. Latching capability

A TM switch can latch the current logic value when it switches from on to off state. Fig. 2 illustrates the necessity of latching data. In the first microcycle, the state of TM switches S_1 and S_3 are on. This means the connection $w_2 - w_1 - w_3$ is maintained for the signal N_1. In the second microcycle, TM switches S_1 and S_3 are turned off, while S_2 and S_4 are turned on. Hence, the connection $w_4 - w_1 - w_5$ is turned on while $w_2 - w_1 - w_3$ is turned off for the signal N_2. In order to prevent w_3 from the floating state, the switch S_3 must to work as a driver. When it transits from on to off, S_3 will latch the current value, then drive the w_3 segment.

Francis' architecture[9] can provide the latching capability, but it is provided by wires which are different from ours. Our architecture does not have to apply latches at LUT inputs, because our TM switches can provide the latching function.

4. Time-multiplexing-aware timing-driven-routing algorithm

4.1. Problem formulation

$G(V, E)$ is a routing-resource graph which is used to illustrate routing resources and their connections in TM-ARCH. V is the set of vertices (or nodes) which correspond to CLB pins or wires. The set of edges E corresponds to switches. During the association between every node and edge, there should be a delay time d. C_{ap} is the capacity of a node, which is defined as the maximum number for different nets that can occupy this node in one microcycle. We set C_{ap} equals to 1 for the nodes that correspond to the wires in our architecture. This is because at any microcycle a wire must be used by at most one net. Fig. 3 illustrates the routing-resource graph of

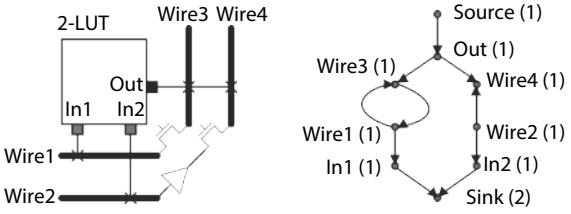


Fig. 3. Routing resource graph of TM-ARCH architecture.

a TM-ARCH device when $M_c = 2$. The number listed next to the node name shows the capacity of this node. It should be noted that the sink and source are two different dummy nodes. A capacity of 2 is allocated to simulate the logical equivalence of the two input pins of a 2-input LUT.

For a signal i which is needed to be optimized in TM-ARCH, its net N_i can be considered as a set of terminals, including source terminal S_i and sink terminals S_{ij}. N_i constitutes a subset of V. A routing solution for net N_i can be considered as a directed routing tree RT_i embedded in G and connecting S_i with all S_{ij}.

The router of TM-ARCH is designed for optimizing the circuit delay and routing all the nets at the same time. The router should identify and then schedule multiple qualified nets to a time-multiplex wire because the wire is time-multiplexed in TM-ARCH architecture.

4.2. Occupation bitmap

Our algorithm can identify a net (or a signal) that can be multiplexed to a wire with other signals. We record the time when a signal arrives and leaves. This helps us to get the signal's occupation bitmap, which shows at which microcycle this signal may occupy this wire.

4.2.1. Arrival and leave time

A Maze router[23] is the core of our algorithm, the Pathfinder algorithm[24] and the VPR 5 algorithm. This router routes a net by wave-expanding from source and throughout the routing-resource graph until the wave-front reaches the sink of the net. It then back-traces the path and records it from its source to sink.

We compute two timing values: $t_{arrival}$ and t_{leave} for a wire which is being expanded. $t_{arrival}$ records when the signal arrives at this wire. Furthermore, t_{leave} records when the signal leaves this wire. The two values can be denoted as:

$$t_{arrival}(n) = \sum_{m \in source(i) \rightsquigarrow n} d_m + T_{arrival}(source(i)), \quad (1)$$

$$t_{leave}(n) = t_{arrival}(n) + d_n, \quad (2)$$

In Eq. (1), the first item on the right side indicates the time needed from the source of net i to node n. The second item indicates the time when a signal arrives at the source of net i. In Eq. (2), d_n is the delay of node n. The value of d_n is illustrated in the FPGA architecture files from its manufacturer. We get the signal leave time by adding $t_{arrival}$ and d_n. In this work, we consider the delay as constant. All timing values in both equations are the maximum possible delay. We record the time that a signal arrives and leaves for every segment in the net. One wire may be included by the routes of multiple nets during the routing process. At this time, we record all

```
        t_arrival: Signal arrival time at wire n;
        t_leave: Signal leave time at wire n;
        T_crit: Circuit critical path delay;
        M_c: Number of microcycles in a clock cycle;

1       for (i = 1; i <= M_c; i++) {
2           Bitmap [i] = 0;
3       }
4       T_ucycle = T_crit/M_c;
5       for (i = 1; i <= M_c; i++) {
6           if (t_arrival > (i − 1) * T_ucycle + T_gb &&
7               t_arrival < i * T_ucycle) {
8               begin = i;
9               break;
10          }
11      }
12      for (i = 1; i <= M_c; i++) {
13          if (t_leave > (i − 1) * T_ucycle &&
14              t_leave < i * T_ucycle − T_gb) {
15              end = i;
16              break;
17          }
18      }
19      for (i = begin; i <= end; i++) {
20      Bitmap [i] = 1;
21      }
```

Fig. 4. Pseudo code for computing the occupation bitmaps.

the time that signal arrives and leaves from the wire.

4.2.2. Occupation bitmap

We can compute the occupation bitmap by using all the t_{arrival} and t_{leave} values we record. This bitmap can be used to represent the occupation of a wire n by one net at different microcycles. The bitmap is an array that consists of M_{c} elements. Each element corresponds to one microcycle: the first element corresponds to the first microcycle, and the last element corresponds to the last microcycle.

Each element in the bitmap array can be 0 or 1. "1" means that the net uses the wire at the corresponding microcycle. "0" means that the net does not occupy.

We use the following function to calculate each element in the array,

$$Bitmap[i] = \begin{cases} 0, & \text{if} \quad t_{\text{arrival}} > iT_{\text{ucycle}} + T_{\text{gb}}, \\ 0, & \text{else if} \quad t_{\text{leave}} < (i-1)T_{\text{ucycle}} - T_{\text{gb}}, \\ 1, & \text{else}, \end{cases} \tag{3}$$

$$T_{\text{ucycle}} = T_{\text{crit}}/M_{\text{c}}. \tag{4}$$

We also use T_{gb} as the guard band for the skew of the microcycle clock network. The basic idea of this equation is that the net will only occupy this wire at a particular microcycle.

Fig. 4 gives the pseudo code for computing bitmaps. We first set all bitmap array elements to zero. Next, we compute which microcycle signal will arrive and give it to variable *begin*. Then we compute which microcycle signal will leave, and give it to a variable *end*. Finally, we assign all elements between *begin* and *end* to 1, which means in these microcycles this net occupies the wire.

We also record the bitmap array for every wire in net i. Because we record the time when net i's signal arrives and leaves the wire, we use these data to calculate the bitmap. We record multiple occupation bitmaps if this wire is occupied by the routes of multiple nets.

4.3. Congestion penalties at microcycles

Because wires in our architecture can be multiplexed by nets, our algorithm can record how many nets are currently using this wire at every microcycle by using the micro occupancy. This micro occupancy illustrates the degree of congestion at each microcycle for a wire. Our algorithm can compute the congestion penalties of wire at each microcycle based on its micro occupancy values when the wavefront arrives at the wire. Larger micro occupancy will cause larger congestion penalty.

4.3.1. Micro occupancy

Micro occupancy is a matrix that consists of M_{c} elements for each wire. Each element corresponds to one microcycle and takes an integer value, which means how many nets are currently occupying this wire at the corresponding microcycle.

Firstly we set all micro occupancy elements to zero. When wire n is included in the route of net i, the micro occupancy of n will be updated by the bitmap in section IV-B.

$$uOcc[j] = \begin{cases} uOcc[j] + 1, & \text{if} \quad Bitmap[j] = 1, \\ uOcc[j], & \text{else}, \end{cases} \tag{5}$$

where j is in $[1, M_{\text{c}}]$. If the j-th element of bitmap is 1, it means this net uses this wire in the j-th microcycle and then j-th element of micro occupancy's j-th element will be increased by one.

When a net i is being disassembled, the micro-occupation of all wires in the route of net i is updated.

$$uOcc[j] = \begin{cases} uOcc[j] - 1, & \text{if} \quad Bitmap[j] = 1, \\ uOcc[j], & \text{else}, \end{cases} \tag{6}$$

where j is in $[1, M_{\text{c}}]$. Since this wire was used by net i in the j-th microcycle, we should decrease the j-th element of micro occupancy by one.

4.3.2. Historical and present congestion penalty

With the records of micro occupancy, we can calculate the historical and present congestion penalties of a wire in each microcycle. The functions we use are shown as follows:

$$p[j] = 1 + \max(0, p_{f_{\text{ac}}}[uOcc[j] + 1 - C_{\text{ap}}]), \tag{7}$$

$$h[j]^i = \begin{cases} 1, & i = 1, \\ h[j]^{i-1} + \max(0, h_{f_{\text{ac}}}[uOcc[j] - C_{\text{ap}}]), & i > 1. \end{cases} \tag{8}$$

where j is in $[1, M_{\text{c}}]$. We will update $p[j]$ for all wires affected if there still have any net disassembled like VPR. We also update $h[j]$ of all wires only after one entire iteration.

The routing schedule shows what $h_{f_{\text{ac}}}$ and $p_{f_{\text{ac}}}$ can be in each routing iteration. Table 1 illustrates the routing schedule which is also used in VPR 5.

4.4. Multiplexing-aware congestion cost

Cost calculation of multiplexing-aware congestion is an innovation of this algorithm. We do not consider it as congestion if a wire in our architecture is used by two more signals and they occupy the wire at different time.

Fig. 5 shows the pseudo code for computing congestion cost of wire in the wave expansion process. In this pseudo

Table 1. Resource utilization of the system.

Routing schedule	Value
$p_{f_{ac}}$	0.5 in the first and the second routing iteration; 1.3 times its previous value from the third iteration onwards.
$h_{f_{ac}}$	1.0 for all the iterations.

```
        Bitmap [1..Mc]: Bitmap array of net i at wire n;
        Pn [1..Mc]: Present congestion penalty array of wire n;
        hn [1..Mc]: Historical congestion penalty array of wire n;
        bn: Base cost of wire n;

1       accCost = 0.;
2       for (i = 1; i < = Mc; i + +) {
3           accCost = max (accCost, hn [i])
4       }
5       for (i = 1; i < = Mc; i + +) {
6           if (Bitmap [i] = = 1) {
7               begin = i;
8               break;
9           }
10      }
11      for (i = begin; i < = Mc; i+ +) {
12          if (Bitmap [i] = = 0) {
13              end= i − 1;
14              break;
15          }
16      }
17      presCost = 0.;
18      for (i = begin; i < = end; i + +) {
19          presCost = max (presCost, pn [i])
20      }
21      cCost = presCost * accCost * bn;
```

Fig. 5. Pseudo code for computing the congestion cost.

code, we should have already computed the bitmap array and the newest historical and present congestion penalties for every microcycle.

Lines 1–4 look for the largest historical congestion penalty. Lines 5–10 look for the position where the first "1" value is occurred in a bitmap, and save it to begin index. This means from which microcycle the wire starts to be occupied. Lines 11–16 saves the end index in a similar way, which means at which microcycle the wire occupation ends. Lines 17–20 record the largest present congestion penalty between the begin microcycle and end microcycle. Line 21 computes the overall congestion cost by multiplying the largest historical and present congestion penalties and the base cost of wire.

$$cCost = p_n \times h_n \times b_n. \tag{9}$$

Eq. (9) is used after the VPR router obtains the new congestion cost. We use the largest historical congestion penalty in all microcycles as h_n and use the largest present congestion penalty in the microcycles during which the wire is occupied. We only award the congestion of occupying wires in one net from the *begin*-th microcycle until the *end*-th microcycle.

4.5. Overall cost function

The overcall cost function is the sum of two factors as in a VPR router. The first factor is the congestion sensitive cost, which is computed in Section 4.5. The second factor is the delay sensitive cost. We use the wire's intrinsic delay as the delay cost and weighs the two parts based on timing criticality. Eq. (10) shows the overall cost function in our algorithm.

```
        Gt: Circuit timing graph
        Q(i): Priority queue used while routing net Ni
        RT(i): Routing trcc of net Ni
        Source(i): Source of net Ni

1       Back-annotate placement delays of all nets into Gt;
2       Propagate timing in Gt, and compute Tcrit;
3       Crit(i, j) = 1.0 for all i and j;
4       while (overused resources exist) {
5         for (each net Ni) {
6           Rip-up RT(i), and update p(n) for all nodes n in RT(i);
7           RT(i) = Source(i);
8           for (each sink j of Ni in decreasing Crit(i, j) order) {
9             Q(i) = RT(i);
10            while (sink(i, j) not found) { /* Wave expansion */
11              Remove lowest cost node, m, from Q(i);
12              for (all fanout nodes n of node m) {
13                Calculate tarrival and tleave for n;
14                Calculate Bitmap (i, n) using current Tcrit;
15                Evaluate c(n);
16                Add n to the Q(i);
17              }
18            } /* Routing of one sink is finished. */
19            for (all nodes, n, in path from RT(i) to sink(i, j)) {
20              Update p(n);
21              Add n to RT(i);
22              Calculate tarrival and tleave for n;
23              Calculate Bitmap(i, n) using current Tcrit;
24            }
25            Update Elmore delay of RT(i);
26            Update tarrival and tleave for all n in RT(i);
27            Update Bitmap(i, n) all n in RT(i);
28          } /* Routing of one net is finished. */
29        } /* Routing of all nets are finished. */
30        Back-annotate Elmore delays of RT(i) of all nets i into Gt;
31        Propagate timing in Gt, and compute Tcrit;
32        Update Bitmap(i, n) for all nets Ni on all nodes n;
33        Update h(n) for all nodes n;
34        Update Crit(i, j);
35      } /* End of one routing iteration*/
```

Fig. 6. Pseudo code for computing the congestion cost.

$$c(n) = Crit(i,j) \times d(n) + [1 - Crit(i,j)] \times cCost(n). \tag{10}$$

4.6. Legal routing solution

Our routing algorithm will check whether the current routing is effective after each iteration. If it is, our router iteration ends and keeps this solution. But if not, the router will start a new turn iteration.

A valid routing solution should not contain any overused routing resources. Although a wire can be multiplexed to be used by multiple nets, it is not overused as long as it is occupied by at most one net in any microcycle. Our router checks if Eq. (11) are met in every microcycle.

$$uOcc[i] \leqslant 1, \quad i \in [1, M_c]. \tag{11}$$

4.7. Pseudo code

Fig. 6 is the pseudo code of time-multiplexing-aware routing algorithm. For each signal in a net, we firstly use the maze router to route one signal from its source to sink over the routing-resource graph and record the path it has expanded. For each wire being expanded, we also record the arrival time and leave time to calculate the occupation bitmap. We use the bitmap to compute and update congestion penalties on this wire and then evaluates this wire's overall cost. We aim to decrease the criticality of the connection while doing the wave expansion. For all nodes in each net, we up-

date its occupied time and Elmore delay after this net has been completely routed. After finishing routing all nets, we compute propagate timing in circuit timing graph and calculate critical path delay. If there is no more overused resources or reach the maximum number of iterations, the program will stop and return the final results.

Lines 6 and 20 show that this algorithm can compute the present congestion penalties at each microcycle for wires. Lines 14, 23, 27, 32 illustrate the fact that our algorithm can compute and update the occupation bitmap on wires. Lines 13, 22, and 26 are where our algorithm compute and update the time of signal arrival and leaving. Line 15 reflects the fact that this algorithm computes the time-multiplexing-aware congestion cost for a wire and then analyzes overall cost of this wire. Line 33 computes the historical congestion penalties at microcycles. Finally, line 4 checks the routing algorithm if there still have any remaining congested routing resources. This has been detailed in Section 4. In addition, our router can also check whether the next condition holds in every microcycle.

4.8. Algorithm analysis

In this part, we mainly focus on its unroutability detection, time complexity and memory requirement.

4.8.1. Unroutability detection

Our algorithm only declares the circuit is unroutable after 50 iterations on the given FPGA like Pathfinder algorithm. Because this process will take a long time, we will enhance our algorithm for quicker unroutability detection.

4.8.2. Time conplexity

The algorithm is based on iterations. In practice, the iteration number is usually limited to a certain number of times. Therefore, it is sufficient to analyze the iteration complexity in this algorithm.

The pseudo-code in Fig. 6 shows that, every iteration is made up of two parts. Netlist routing is the first (lines 5–29), and post-processing is the second (lines 30–34). Netlist routing means running routing algorithm for every net in netlist. Previous work shown that the complexity of the net routing algorithm in Pathfinder is $O(k^2 \log k)$[3]. k means how many terminals does a net has. In the routing process, the extra computations included in our algorithm include computing signal's arrival and leave times and then computing the occupation bitmap.

From Section 4.2, we know that the time consume on a routing-resource node n is constant. We also know that the complexity of computing bitmap is $O(M_c)$. M_c means how many microcycles does one user's clock cycle have. In our work, we limit M_c to 8. As a result, we consider it takes constant time to get the occupation bitmap. Taking the above into consideration, we can know that, for a k-terminal net, routing in our algorithm will typically take $O(k^2 \log k)$ time, which is same as in Pathfinder.

Next we will do post-processing. The main computation is computing T_{crit} by doing static timing analysis on timing graph in this part. We also used the critical path method (CPM) which is used in VPR 5 algorithm. Sapatnekar *et al.*[25] shown that the complexity of this algorithm is $O(V + E)$. V means the vertices number in the circuit timing graph, and E means the edges number in the timing graph.

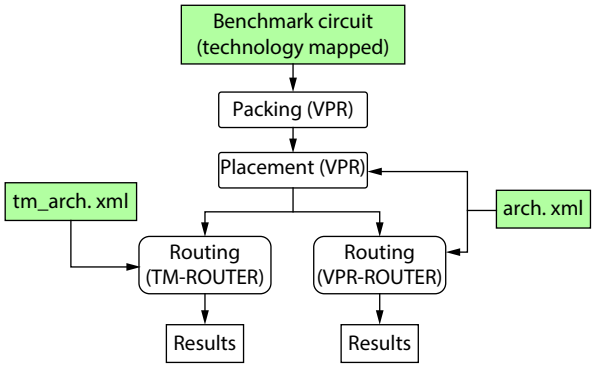


Fig. 7. (Color online) The TM-ARCH and TM-ROUTER evaluation framework.

Back-annotation Elmore delay (line 30) will take $O(C)$ time for other operations in the post-processing portion, where C means how many connections between source and sink have in this circuit. Calculating occupation bitmap and updating historical congestion penalty of all nodes requires $O(R)$ time. R represents the nodes number in routing-resource graph.

4.8.3. Memory requirement

In our algorithm, FPGA routing-resource graph and circuit timing graph are requiring a substantial amount of main memory requirement.

We record many information for all nodes in the routing-resource graph, such as its connectivity information, physical information, congestion information, timing information and some other information used for wave expansion. $O(R)$ is the memory requirement from routing-resource graph and R means nodes number in routing-resource graph.

We record the timing information and connectivity information for all vertex in timing graphs. For edges in timing graph, we record the timing information and connectivity information. Typically, $O(V + E)$ is the memory requirement for timing graph. V in this means vertices number in the circuit timing graph, and E means edges number in the timing graph.

5. Algorithm validation

We verify our time-multiplexing-aware routing algorithm through experiments. By using this algorithm, we implement benchmark circuits for the TM-ARCH architecture. For comparison, we also implement the same set of benchmark circuits for conventional architectures. To achieve an easy and fair comparison, we use MCNC 20 benchmark circuits with a standard EDA flow.

5.1. Experimental setup

5.1.1. EDA flow

Fig. 7 shows our EDA flow. First, LUTs are packaged into the cluster logic block using the TVPack tool in the VPR 5 package. Next, place the circuit using VPR 5. The wiring is performed twice according to the same placement result. The VPR 5 timing-driven router is used to route circuits to traditional island FPGAs. TM-ARCH FPGAs with time-multiplexed interconnects are supported by our time-multiplexing-aware router. We call these two routing branches as conventional routing and time-multiplexing routing. Since then, we will call VPR timing-driven router and our time-multiplexing-

Table 2. Main Features of Baseline FPGA Architecture.

Feature parameter	Value/specification
LUT size	4
Logic block size	10
Logic block inputs	22
Amount of bias between horizontal and vertical channels	No bias
Uniformity of routing channels in the same direction	Uniform
Aspect ratio	1 : 1 (Assuming square logic blocks)
Segmentation distribution	100% length 4 wires
Switch types used	Uni-directional single driver switches
Switch block topology	Wilton
Switch block internal population	100%
Connection block internal population	100%

Table 3. Minimum channel width for different M_c values.

Parameter	$M_c = 1, W_{min}$	$M_c = 2, W'_{min}$	$M_c = 4, W'_{min}$	$M_c = 6, W'_{min}$	$M_c = 8, W'_{min}$
Alu4	48	48	30	36	26
Apex2	62	62	36	34	22
Apex4	64	64	50	36	28
Bigkey	44	44	32	32	30
Clma	78	76	74	48	34
Des	44	42	34	32	32
Diffeq	38	36	34	32	30
Dsip	38	36	30	30	30
Elliptic	62	58	52	N.A.	34
Ex1010	74	74	60	40	34
Ex5p	68	68	48	34	22
Frisc	74	74	44	40	40
Misex3	54	54	42	26	22
Pdc	90	90	128	60	70
S298	34	34	28	28	20
S38417	48	48	50	26	22
S38584.1	50	50	48	26	22
Seq	60	60	48	26	22
Spla	74	72	N.A.	52	34
Tseng	46	46	40	34	32
Geo.Mean	56	55	44	34	29
Reduction	–	–1.78%	–21.43%	–39.28%	–48.21%

aware router as VPR-ROUTER and TM-ROUTER.

M_c is input to TM-ROUTER by command-line option. In our experiment, we restrict the value of M_c to be 2, 4, 6, or 8. We think 8 is a reasonable upper limit because of two reasons. First, a too large a M_c means a very high-frequency clock Clk_f, because our architecture requires that frequency of Clk_f should be M_c times the user's clock. Second, too large a M_c will result in a large amount of area overhead. It can be seen from the example implementation of the TM switch provided in the paper.

5.1.2. Assumptions of FPGA architecture

In this work, we only consider similar FPGA architectures. Even with this assumption, FPGA architecture's design space is still quite large, so we cannot fully study the time-multiplexed interconnects. In order to make the work easier to do, we first fix a representative benchmark FPGA architecture. The benchmark FPGA architecture is represented in Fig. 7 as "arch.xml", which uses conventional interconnects. We replace all wires with our multiplexable wires, so we got the TM-ARCH FPGA architecture shown in Fig. 7 as "tm_arch.xml".

We choose the XML file used in iFAR[26] as the baseline FPGA architecture. Table 2 shows main features of this FPGA architecture.

5.2. Results

5.2.1. Minimum channel width

In this part of the experiments, a conventional routing and the time-multiplexing routing perform a binary search to find the minimum channel width for each circuit. VPR-ROUTER routes circuits to FPGA with conventional architecture to find the minimum channel width, and TM-ROUTER does the same work on TM-ARCH FPGA. We set 50 as the maximum iterations number for both routing algorithms.

Table 3 shows the minimum channel width experimental results. The numbers listed under "W_{min}" column are the minimum channel widths routed by VPR-ROUTER in the conventional algorithm. Number listed under "W'_{min}" column are that TM-ROUTER achieves for different M_c values. When M_c = 2, TM-ARCH FPGA does not have advantages over conventional architecture. TM-ARCH FPGA achieve slightly better results

Table 4. Percentages (%) of wire used in each individual microcycle for 20 benchmark circuits with $M_c = 2, 4$.

Parameter	$M_c = 2$		$M_c = 4$			
	1st	2nd	1st	2nd	3rd	4th
Alu4	94.59	4.67	56.04	29.64	4.19	0.45
Apex2	97.09	2.42	67.82	25.32	2.17	0.21
Apex4	87.35	10.56	28.13	56.96	9.45	1.00
Bigkey	93.65	4.77	61.57	28.13	4.77	0.00
Clma	96.50	2.97	68.86	25.69	2.83	0.12
Des	90.08	9.15	63.16	24.61	6.50	1.96
Diffeq	95.97	3.80	74.31	18.47	3.71	0.10
Dsip	94.70	4.28	59.70	31.89	4.04	0.18
Elliptic	95.35	4.46	87.51	7.36	3.65	0.73
Ex1010	90.04	8.67	23.65	64.19	7.98	0.51
Ex5p	82.63	14.85	21.70	56.98	12.71	1.99
Frisc	86.87	12.06	68.23	18.50	10.58	1.41
Misex3	93.09	5.60	52.19	33.60	4.64	0.80
Pdc	95.09	4.36	46.93	42.92	3.88	0.42
S298	85.82	13.48	63.21	21.00	9.76	3.16
S38417	92.87	6.60	65.09	24.77	5.83	0.67
S38584.1	97.41	1.88	73.30	19.91	1.59	0.26
Seq	97.50	2.20	69.24	24.66	1.99	0.17
Spla	95.98	3.60	49.20	41.06	3.31	0.25
Tseng	96.48	3.34	88.49	6.82	2.35	0.73
Geo.Mean	92.85	5.17	55.83	26.19	4.49	0.51

for only 5 circuits (des, diffeq, dsip, elliptic, and tseng) in 20 circuits. This result is counter-intuitive, because we expect TM-ARCH FPGA to result in the minimum channel width of all circuits.

We think that, in one clock cycle there are two microcycles that can only supplies little opportunity to TM-ROUTER to achieve time multiplexing of wires. Table 4, "$M_c = 2$" lists the wire percentage used in the first two microcycles for MCNC 20 benchmark circuits, when we divide a clock cycle into to two microcycles. The data are generated in the manner as follows. We adjust VPR-ROUTER codes in order to record the time when the signal occupies wire to the result. Then we start VPR-ROUTER routing process with the minimum channel width listed in Table 3 column "W_{min}". After that, we use a program to analyze the results and determine which microcycle each line segment is used in according to Eq. (3). The program also counts the portion of wires used in each microcycle in the final routing.

Table 4 shows that, in the first microcycle, 92.85% of the wires are used, while in the second microcycle, only 5.17% wires are used. This means that TM-ROUTER has very limited opportunities for time-multiplexing in this condition. To implement time-multiplexing in FPGAs, our routing algorithm will match the same wire in the first microcycle and the second microcycle. Table 4 shows that used wire segments are most likely to be used by net in the first microcycle, and less likely to be used by another net in the second microcycle, so our time-multiplexing-aware algorithm may less likely to implement time-multiplexing wires in this situation. Looking back at Section 4, the signal does not be actively delayed by our routing algorithm. But scheduling algorithm of Francis *et al*'s does it to implement more time-multiplexing[9]. We can see that, when a clock cycle is divided into four microcycles, Table 4, column "$M_c = 4$", tabulates the microcycle distribution using the wires. The process of generating these data is similar to for $M_c = 2$. In this condition, the distribution between the first and second microcycle seems more balanced than when $M_c = 2$. This gives TM-ROUTER more opportunities to implement time-multiplexing.

TM-ARCH generally needs a smaller channel width from $M_c = 4$. This follows our expectations that multiplex can lead to minimum channel width reduction. The result will be more significant when the division of microcycle is more detailed: the reduction of minimum channel width is 19.86% ($M_c = 4$), 37.74% ($M_c = 6$), and 47.89% ($M_c = 8$). With M_c increases, one wire can be multiplexed by more nets, so fewer wires are required to route a given number of nets.

5.2.2. Circuit critical path delay

FPGAs routing resources are often not heavily utilized in order to reduce latency in real applications. So we use 20% more minimum channel width than Table 3 to do the routing process. In this section, we also assume that TM switch has same delay with its conventional switch.

Table 5 lists the critical path delays with low-stress routing for 20 benchmark circuits. "T_{crit}" column is the result of the conventional architecture, while the "T'_{crit}" columns is the result the TM-ARCH architecture. When $M_c = 2$, the critical path delay of TM-ARCH is slightly larger (1.13%) and the channel width used is slightly smaller (4.08%). However, when $M_c = 4$, 6, and 8, the channel width uses by TM-ROUTER is reduced by 19.69%, 37.87%, and 46.97%. Moreover, the critical path delay of TM-ROUTER can be moderately reduced: 1.79%, 3.84%, and 1.28%.

In conventional FPGAs, circuit critical path may not be routed in the shortest path due to limited routing. This is true even when low-stress wiring is performed. Unlike conventional FPGAs, the TM-ARCH architecture allows multiple signals multiplex wires in different clock cycles. This effectively mitigates its routing congestion. As a result, the circuit critical path

Table 5. Minimum channel width for different M_c values.

Parameter	M_c = 1		M_c = 2		M_c = 4		M_c = 6		M_c = 8	
	W_{ls}	T_{crit}	W'_{ls}	T'_{crit}	W'_{ls}	T'_{crit}	W'_{ls}	T'_{crit}	W'_{ls}	T'_{crit}
Alu4	58	3.31	58	3.31	36	3.31	44	3.23	32	3.45
Apex2	74	3.90	74	3.90	46	3.69	44	3.69	26	3.65
Apex4	76	3.80	76	3.80	58	3.13	40	3.20	32	3.20
Bigkey	52	1.80	52	1.87	38	1.80	38	1.80	38	1.80
Clma	94	6.74	94	6.74	88	6.63	58	6.70	44	6.70
Des	52	2.86	48	N.A.	40	2.78	38	2.85	38	2.85
Diffeq	46	4.44	44	4.51	40	4.37	38	4.44	36	4.37
Dsip	46	1.73	44	1.73	36	1.80	36	1.80	36	1.80
Elliptic	74	6.22	70	5.52	62	5.66	N.A.	N.A.	40	5.37
Ex1010	88	4.42	88	4.42	72	4.49	48	N.A.	40	4.42
Ex5p	82	3.55	82	3.55	58	3.34	40	3.23	26	3.37
Frisc	88	7.63	88	7.63	52	7.42	48	7.42	48	7.49
Misex3	64	3.13	64	3.13	50	3.06	32	3.20	26	3.27
Pdc	108	5.26	108	5.26	154	4.49	72	4.49	84	4.49
S298	40	N.A.	40	N.A.	34	6.14	34	6.17	24	6.07
S38417	58	4.68	58	4.47	60	4.47	32	4.61	26	4.40
S38584.1	60	3.71	60	3.71	58	N.A.	32	3.64	26	3.65
Seq	72	3.13	72	3.13	58	3.06	32	3.20	26	3.06
Spla	88	4.46	88	4.46	N.A.	N.A.	62	4.14	40	4.07
Tseng	56	4.43	52	4.43	48	4.43	40	4.43	38	4.43
Geo.Mean	66	3.90	65	3.95	53	3.83	41	3.75	35	3.85
Reduction	N.A.	N.A.	–1.51%	1.28%	–19.69%	–1.79%	–37.87%	–3.84%	-46.97%	-1.28%

is likely to be routed in a shorter path.

6. Conclusion

In this paper we propose a time-multiplexing FPGA architecture and its routing algorithm. Our algorithm actively identifies the qualified interconnects that can be multiplexed on our new FPGA architecture. We have validated the architecture and algorithm by implementing it as a multiplexing-aware router and using this router to implement benchmark circuits to FPGAs with time-multiplexed interconnects. The results show that compared with an existing router targeting a conventional island-style architecture, 38% smaller minimum channel width and 3.8% smaller circuit critical path delay can be achieved.

References

[1] Lemieux G, Lewis D. Design of interconnection networks for programmable logic. Dordrecht: Kluwer Academic Publishers, 2004

[2] Trimberger S, Carberry D, Johnson A, et al. A time-multiplexed FPGA. The 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 1997, 22

[3] Betz V, Rose J. VPR: A new packing, placement and routing tool for FPGA research. International Workshop on Field Programmable Logic and Applications. Springer, Berlin, Heidelberg, 1997, 213

[4] Luu J, Kuon I, Jamieson P, et al. VPR 5.0: FPGA CAD and architecture exploration tools with single-driver routing, heterogeneity and process scaling. ACM Trans Reconfig Technol Syst, 2011, 4(4), 32

[5] Trimberger S. Scheduling designs into a time-multiplexed FPGA. Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays, 1998, 153

[6] Lin C C, Chang D, Wu Y L, et al. Time-multiplexed routing resources for FPGA design. Proceedings of Custom Integrated Circuits Conference, 1996, 152

[7] Francis R, Moore S, Mullins R. A network of time-division multiplexed wiring for FPGAs. Proceedings of the Second ACM/IEEE International Symposium on Networks-on-Chip, 2008, 35

[8] Shen M, Zhang W, Luo G, et al. Serial-equivalent static and dynamic parallel routing for FPGAs. IEEE Trans Comput-Aid Des Integr Circuits Syst, 2018

[9] Shen M, Luo G, Xiao N. Exploiting box expansion and grid partitioning for parallel FPGA routing. 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2018, 209

[10] Shen M, Luo G. Accelerate FPGA routing with parallel recursive partitioning. Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, 2015, 118

[11] Shen M, Xiao N. Fine-grained parallel routing for FPGAs with selective expansion. 2018 IEEE 36th International Conference on Computer Design (ICCD), 2018, 577

[12] Shen M, Xiao N. Raparo: resource-level angle-based parallel routing for FPGAs. 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2019, 312

[13] Shen M, Luo G. Megrez: Parallelizing FPGA routing with strictly-ordered partitioning. 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2017, 27

[14] Vercruyce D, Vansteenkiste E, Stroobandt D. CRoute: a fast high-quality timing-driven connection-based FPGA router. 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2019, 53

[15] Wang D, Duan Z, Tian C, et al. A runtime optimization approach for FPGA routing. IEEE Trans Comput-Aid Des Integ Circuits Syst, 2017, 37(8), 1706

[16] Patil S B, Liu T, Tessier R. A bandwidth-optimized routing algorithm for hybrid FPGA networks-on-chip. 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2018, 25

[17] Chaplygin Y, Novozhilov I, Losev V, et al. Algorithm for design

and structure optimization of the FPGA routing block with a given number of trace signals. 2019 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EICon-Rus), 2019, 1589

[18] Farooq U, Chotin-Avot R, Azeem M, et al. Using timing-driven inter-FPGA routing for multi-FPGA prototyping exploration. 2016 Euromicro Conference on Digital System Design (DSD), 2016, 641

[19] Omam S R, Tang X, Gaillardon P E, et al. A study on buffer distribution for RRAM-based FPGA routing structures. 2015 IEEE 6th Latin American Symposium on Circuits and Systems (LASCAS), 2015, 1

[20] Chen S C, Chang Y W. FPGA placement and routing. Proceedings of the 36th International Conference on Computer Aided Design, 2017, 914

[21] Huriaux C, Sentieys O, Tessier R. Effects of I/O routing through column interfaces in embedded FPGA fabrics. 2016 26th International Conference on Field Programmable Logic and Applications (FPL), 2016, 1

[22] Kashif A, Khalid M A S. Experimental evaluation and comparison of time-multiplexed multi-FPGA routing architectures. 2016 IEEE 59th International Midwest Symposium on Circuits and Systems (MWSCAS), 2016, 1

[23] Palczewski M. Plane parallel A* maze router and its application to FPGAs. Proceedings 29th ACM/IEEE Design Automation Conference, 1992, 6911

[24] McMurchie L, Ebeling C. PathFinder: a negotiation-based performance-driven router for FPGAs. Proceedings of the 1995 ACM Third International Symposium on Field-Programmable Gate Arrays, 1995, 111

[25] Sapatnekar S. Timing. Springer Science & Business Media, 2004

[26] Kuon I, Rose J. iFAR –intelligent FPGA architecture repository. 2008