12. Time-Switched Field Programmable Gate Arrays

We established in Chapter 7 that active interconnect area consumed most of the space on traditional, single-context FPGAs. In Chapter 10, we saw that adding small, local, context memories allowed us to reuse active area and achieve smaller task implementations. Even in these multicontext devices, we saw that interconnect consumed most of the area (Section 10.4.2). In Chapter 11, we added architectural registers for retiming and saw more clearly the way in which multiple context evaluation saves area primarily by reducing the need for active interconnect. In this chapter, we describe the Time-Switched Field Programmable Gate Array (TSFPGA), a multicontext device designed explicitly around the idea of time-switching the internal interconnect in order to implement more effective connectivity with less physical interconnect.

One issue which we have not addressed in the previous sections is the complexity of physical mapping and, consequently, the time it takes to perform said mapping. Because of the computational complexity, physical mapping time can often be the primary performance bottleneck in the edit-compile-debug cycle. It can also be the primary obstacle to achieving acceptable mapping time for large arrays and multi-chip systems.

In particular, when the physical routing network provides limited interconnectivity between LUTs, it is necessary to carefully map logical LUTs to physical LUTs in accordance with both netlist connectivity and interconnect connectivity. The number of ways we can map a set of logical LUTs to a set of physical LUTs is exponential in the the number of mapped LUTs, making the search for an acceptable mapping which simultaneously satisfies the netlist connectivity constraints and the limited physical interconnect constrains – *i.e.* physical place and route – computationally difficult. Finding an optimal mapping is generally an NP-complete problem. Consequently, in traditional FPGAs, this mapping time can be quite large. It often take hours to place and route designs with a couple of thousand LUTs. The computational complexity arises from two features of the mapping problem:

1. As noted in Section 11.1, traditional FPGAs **must** have enough routing resources to physically route all task connections simultaneously.
2. Since interconnect is the dominant area in FPGAs (Chapter 7), conventional FPGAs try to use as little interconnect as feasible to provide high computational density.

The result is a large set of simultaneous constraints which **must** be satisfied during mapping, making the task of physical mapping computationally intensive. TSFPGA virtually eliminates the simultaneous constraint satisfaction required to successfully route a component, making it possible to rapidly map tasks to the array. Simultaneous constraint satisfaction is still necessary to achieve the highest performance mappings on TSFPGA, but is not necessary to achieve any mapping. This gives the device user control over mapping time and quality.

This chapter details a complete TSFPGA design including:

1. Time-switched input register

2. Techniques used by TSFPGA to avoid constraints
3. Sample interconnect model for time-switched routing
4. Complete gate-array architecture built around:
   (a) time-switched input register
   (b) switched interconnect
   (c) pipelined interconnect
5. Area and time estimates for TSFPGA building blocks
6. Experimental, quick mapping software
7. Mapped benchmark results using experimental software and a sample design point

TSFPGA was developed jointly by Derrick Chen and André DeHon. Derrick worked out VLSI implementation and layout issues, while André developed the architecture and mapping tools.

12.1 Time-Switched Input Registers

As noted in Section 11.1, if all retiming can be done in input registers, only a single wire is strictly needed to successfully route the task. The simple input register model used for the previous chapter had limited temporal range and hence did not quite provide this generality. In this section, we introduce an alternative input strategy which extends the temporal range on the inputs without the linear increase in input retiming size which we saw with the shift-register based input microarchitecture in the previous chapter.

The trick we employ here is to have each logical input load its value from the active interconnect at just the right time. As we have seen, multicontext evaluation typically involves execution of a series of microcycles. A subset of the task is evaluated on each microcycle, and only that subset requires active resources in each microcycle. We call each microcycle a **timestep** and, conceptually at least, number them from zero up to the total number of microcycles required to complete the task. If we broadcast the current timestep, each input can simply load its value when its programmed load time matches the current timestep.

Figure 12.1 shows a 4-LUT with this input arrangement which we will call the *Time-Switched Input Register*. Each LUT input can load any value which appears on its input line in any of the last i cycles. The timestep value is $\lceil \log_2(i) \rceil$ bits wide, as is the comparator. With this scheme, if the entire computation completes in i timesteps, all retiming is accomplished by simply loading the LUT inputs at the appropriate time – *i.e.* loading each input just when its source value has been produced and spatially routed to the destination input. Since the hardware resources required for this scheme are only logarithmic in the total number of timesteps, it may be reasonable to make i large enough to support most all desirable computations.

With this input structure, logical LUT evaluation time is now decoupled from input arrival time. This decoupling was not true in FPGAs, DPGAs, or even iDPGAs. With FPGAs, the LUT is evaluated only while the inputs are held stable. With DPGAs, the LUT is evaluated only on the microcycle when the inputs are delivered. With the iDPGA, the LUT must be evaluated on the

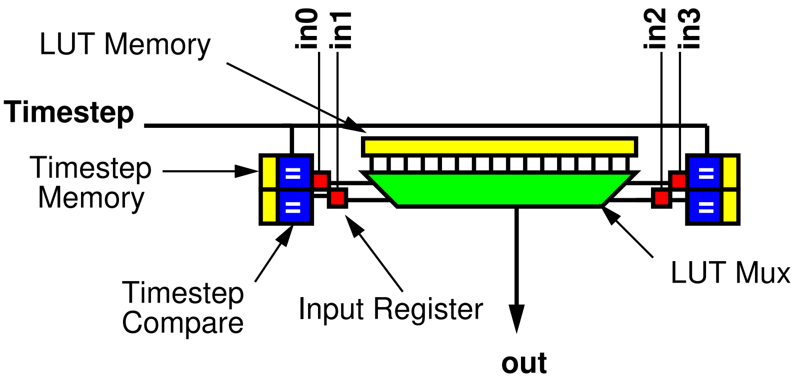


Figure 12.1: 4-LUT with Time-Switched Input Register

correct cycle relative to the arrival of the input, and the range of feasible cycles was limited by i. Further, with the time-switched input register, the inputs are stored, allowing the LUT result to be produced on any microcycle, or microcycles, following the arrival of the final input. In the extreme case of a single wire for interconnect, each LUT output would be produced and routed on a separate microcycle. Strictly speaking, of course, with a single wire there need be only one physical LUT, as well.

This decoupling of the input arrival time and LUT evaluation time allows us to remove the simultaneous constraints which, when coupled with limited interconnectivity, made traditional programmable gate array mapping difficult. We are left with a single constraint: schedule the entire task within i timesteps.

12.2 Switched Interconnect – Folding

Now that we no longer need to involve the physical interconnect in temporal transport, we are free to reuse physical interconnect resources at their minimum operating time. This reuse allows us to employ less physical interconnect than traditional FPGAs, while simultaneously providing more connectivity.

12.2.1 Subarray Structure

Conceptually, let us consider array interconnect as composed of a series of fully interconnected subarrays. That is, we arrange groups of LUTs in subarrays, as in the DPGA prototype (See Section 10.4). Within a subarray, LUTs are fully interconnected with a monolithic crossbar. Also feeding into and out of this subarray crossbar are connections to the other subarrays.

The subarray contains a number of LUTs, N_{sa_LUT}, where we consider $N_{sa\_LUT} = 64$ as typical. Connecting into the subarray are N_{sa_in} inputs from outside. Similarly, N_{sa_out} connect out. N_{sa_in}, and N_{sa_out} are typically governed by Rent's Rule. With $N_{sa\_LUT} = 64$, $k = 4$, and $0.5 \leq p \leq 0.7$, we might expect $32 \leq N_{sa\_in} = N_{sa\_out} \leq 74$, and consider $N_{sa\_in} = N_{sa\_out} =$

64 typical and convenient.

Together, this suggests a $(N_{sa\_LUT} + N_{sa\_in}) \times (k \times N_{sa\_LUT} + N_{sa\_out})$ crossbar, which is 128×320 for the typical values listed above. This amounts to 640 switches per 4-LUT, which is about 2-3$\times$ the values used in conventional FPGA architectures as we reviewed in Section 7.5. Conventional architectures, effectively, only populate 30-50% of the switches in such a block relying on placement freedom to make up for the missing switches. It is, of course, the complexity of the placement problem in light of this depopulation which is largely responsible for the difficulty of place and route on conventional architectures.

We also need to interconnect these subarrays. For small arrays it may be possible to simple interwire the connections between subarrays without substantial, additional switching. This is likely the case for the 100-1000 LUT cases reviewed in Section 7.5. For larger arrays, more, inter-array switching will be required to provide reasonable connectivity. As we derived in Section 7.6, the interconnect requirements will grow with the array size.

12.2.2 Interconnect Folding

With switched interconnect, we can realize a given level of connectivity without placing all of the physical switches that such connectivity implies. Rather, with the ability to reuse the switches, we effect the complete connectivity over a series of microcycles.

We can view this reuse as a folding of the interconnect in time. For instance, we could map pairs of LUTs together such that they share input sets. This, coupled with cutting the number of array outputs (N_{sa_out}) in half, will cut the number of crossbar outputs in half and hence halve the subarray crossbar size. For full connectivity, it may now takes us two microcycles to route the connections, delivering the inputs to half the LUTs and half the array outputs in each cycle. In this particular case we have performed **output folding** by sharing crossbar outputs (See Figure 12.2). Notice that the time-switched input register allows us to get away with this folding by latching and holding values on the correct microcycle. The input register also allows the non-local subarray outputs to be transmitted over two cycles. In the most trivial case, the array outputs will be connected directly to array inputs in some other array and, through the destination array's crossbar, they will, in turn be connected to LUT inputs where they can be latched on the appropriate microcycle as they arrive.

There is one additional feature worth noting about output folding. When two or more folded LUTs share input values all the LUTs can load the input when it arrives. For heavily output folded scenarios, these shared inputs can be exploited by appropriate grouping to allow the task to be routed in less microcycles than the total network sharing.

We can also perform **input folding**. With input folding, we pair LUTs so that they share a single LUT output. Here we cut the number of array inputs (N_{sa_in}) in half, as well. The array crossbar now requires only half as many inputs as before and is, consequently, also half as large in this case. Again, the latched inputs allow us to load each LUT input value only on the microcycle on which the associated value is actually being routed through the crossbar. For input folding, we must add an effective pre-crossbar multiplexor so that we can select among the sources which share a single crossbar input (See Figure 12.3).

It is also possible to fold together distinct functions. For example, we could perform an input fold such that the 64 LUT outputs each shared a connection into the crossbar with the 64 array inputs. Alternately, we could perform an output fold such that LUT inputs shared their connections

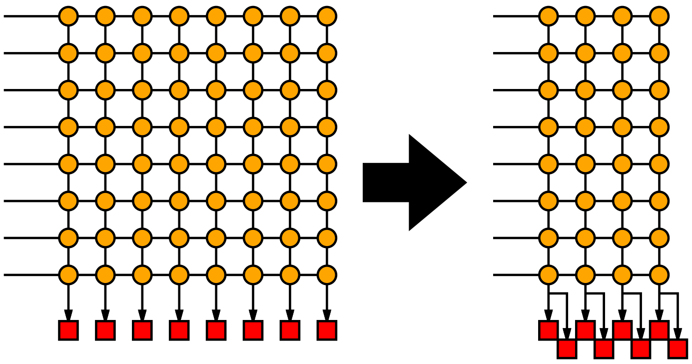


Figure 12.2: Output Folding

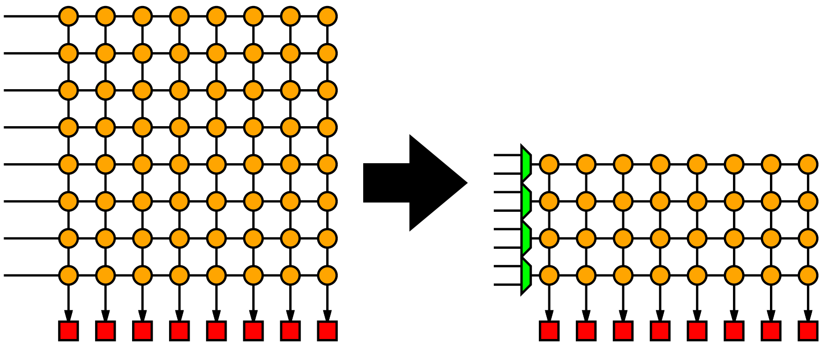


Figure 12.3: Input Folding

with array outputs.

Finally, note that we can perform input folding and output folding simultaneously (See Figure 12.4). We can think of the DPGAs introduced in Chapter 10 as folded interconnect where we folded both the network input and output c times. Each DPGA array element (See Figure 11.2) shared c logical LUT inputs on one set of physical LUT inputs and shared c logical LUT outputs on a single LUT output. Figure 12.5 shows how a two context DPGA results from a single input and output fold. In the DPGA, we had only c routing contexts for this c^2 total folding. To get away with this factor of c reduction in interconnect description, we had to restrict routing to temporally adjacent contexts. As we saw, in Chapter 10 this sometimes meant we had to allocate LUTs for through routing when connections were needed between contexts.

Routing on these folded networks naturally proceeds in both space and time. This gives the networks the familiar time-space-time routing characteristics pioneered in telephone switching systems.

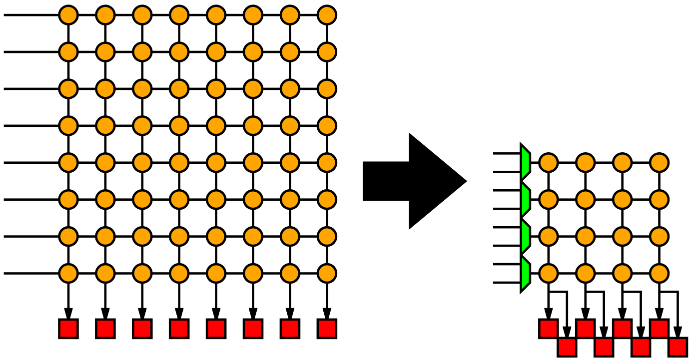


Figure 12.4: Input and Output Folding

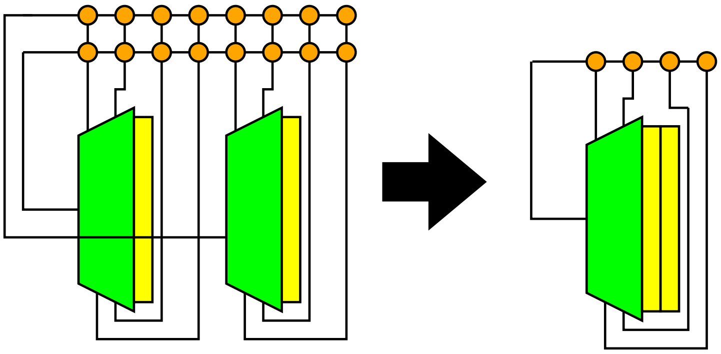
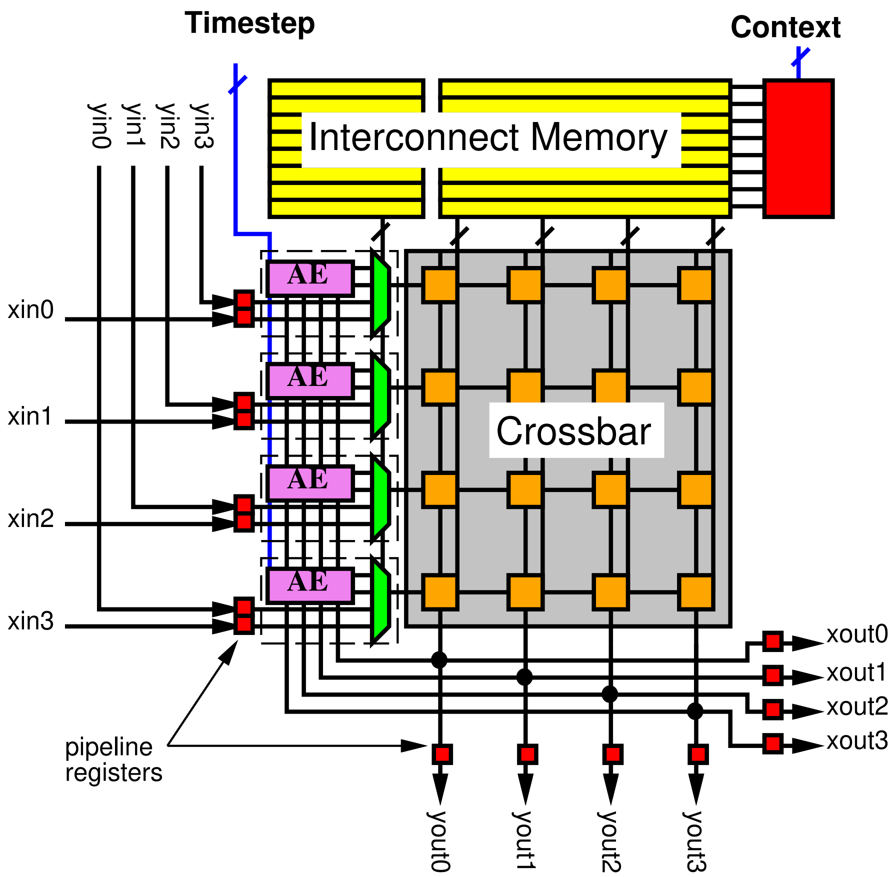


Figure 12.5: Two-Context DPGA as Input and Output Fold

12.3 Architecture

In this section, we detail a complete TSFPGA architecture. The basic TSFPGA building block is the subarray tile (See Figure 12.6) which contains a collection of LUTs and a central switching crossbar. LUTs share output connections to the crossbar and input connections from the crossbar in the folded manner described in the previous section. Communication within the subarray can occur in one TSFPGA clock cycle. Non-local input and output lines to other subarrays also share crossbar I/O's to route signals anywhere in the device. Routes over long wires are pipelined to maintain a high basic clock rate.

Array Element The TSFPGA array element is made up of a number of LUTs which share the same crossbar outputs and input (See Figure 12.7). The LUT output into the crossbar is selected based on the routing context programming. As shown, each array element shares its crossbar input with several network inputs.



(Subarray shown is smaller than typically used in practice in order to avoid unnecessarily complicating the diagram.)

Figure 12.6: TSFPGA Subarray Composition

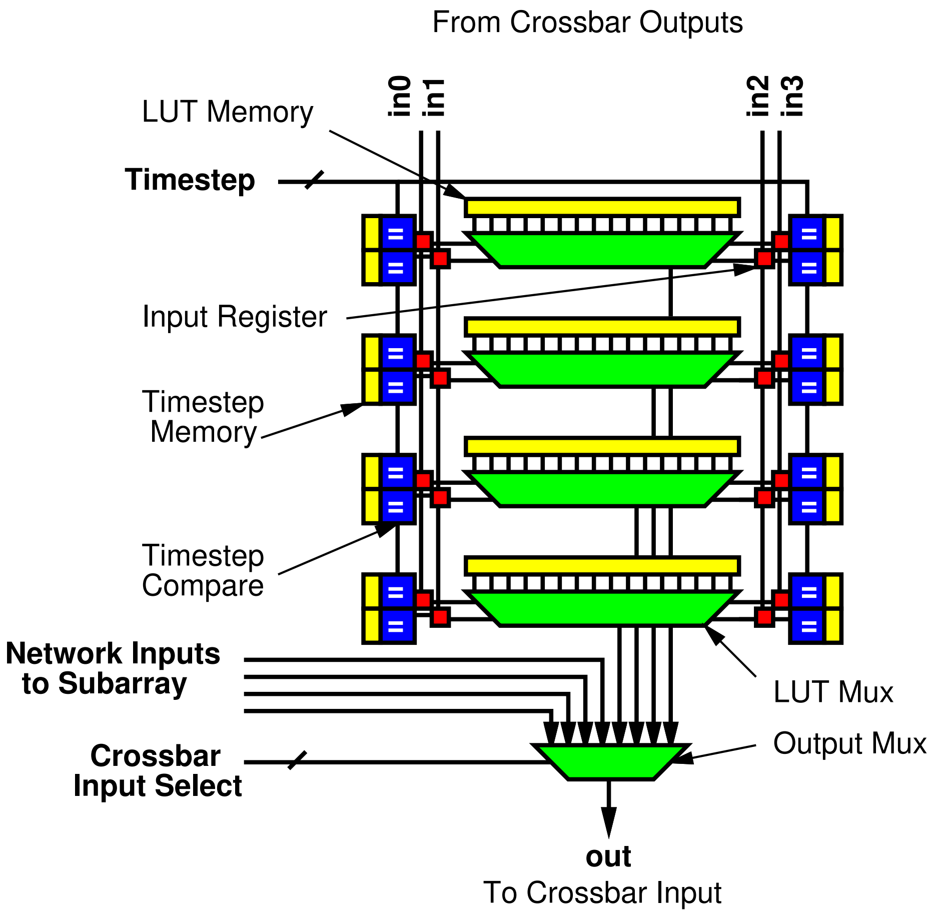


Figure 12.7: TSFPGA Array Element Composition

The LUT input values are stored in time-switched input registers. The inputs to the array element are run to all LUT input registers. When the current timestep matches the programmed load time, the input register is enabled to load the value on the array-element input. When multiple LUTs in an array element take the same signal as input, they may be loaded simultaneously.

Unlike the DPGA architectures detailed in Chapters 10 and 11, the LUT multiplexor is replicated for each logical LUT. As we saw in Section 10.4.2, the LUT mux is only a tiny portion of the area in an array. Replicating it along with context memory avoids the need for final LUT input multiplexors which would otherwise be in the critical path. When considering both the additional input multiplexors and the requirements for selecting among the LUT programming memory, the benefit of resource sharing at this level have been minimal in most of the implementations we have examined.

Crossbar The primary switching element is the subarray crossbar. As shown in Figures 12.6 and 12.7, each crossbar input is selected from a collection of subarray network inputs and subarray LUT outputs via by a pre-crossbar multiplexor. Subarray inputs are registered prior to the pre-crossbar multiplexor and outputs are registered immediately after the crossbar, either on the LUT inputs or before traversing network wires. This pipelining makes the LUT evaluations and crossbar traversal a single pipeline stage. Each registered, crossbar output is routed in several directions to provide connections to other subarrays or chip I/O.

Inter-subarray wire traversals are isolated into a separate pipeline stage between crossbars. As we saw both in Section 7.1.3 and the DPGA prototype implementation (Section 10.4.2), wire and switch traversals are responsible for most of the delay in programmable gate arrays. By pipelining routes at the subarray level, we can achieve a smaller microcycle time and effectively extract higher capacity from our interconnect.

Notice that the network in TSFPGA is folded such that the single subarray crossbar performs all major switching roles:

1. **output crossbar** – routing data from LUT outputs to destinations or intermediate switching crossbars
2. **routing crossbar** – routing data through the network between source and destination subarrays
3. **input crossbar** – receiving data from the network and routing it to the appropriate destination LUT input

This sharing avoids dedicating specialized routing resources to any single function so that the available resources can be deployed as needed by the task. Connections on TSFPGA are statically routed in a distributed, multistage switching fashion.

Intra-Subarray Switching Communication within the subarray is simple and takes one clock cycle per LUT evaluation and interconnect. Once a LUT has all of its inputs loaded, the LUT output can be selected as an input to the crossbar, and the LUT's consumers within the subarray may be selected as crossbar outputs. At the end of the cycle, the LUT's value is loaded into the consumers' input registers, making the value available for use on the next cycle.

Inter-Subarray Switching Figure 12.8 shows the way a subarray may be connected to other subarrays on a component. A number of subarray outputs are run to each subarray in the same row and column. For large designs, hierarchical connections may be used to keep the bandwidth between subarrays reasonable for while maintaining a limited crossbar size and allowing distant connections. The hierarchical connections can give the logical effect of a three or four dimensional network.

Routing data within the same row or column involves:

1. Route LUT output through crossbar to the outputs headed for the destination subarray.
2. Traverse the wire between subarrays.
3. Select network input with source value as a crossbar source and route through the crossbar to the destination LUT input.

When data needs to traverse both row and column:

1. Route LUT output to first dimension destination (row, column).
2. Traverse first dimension interconnect.

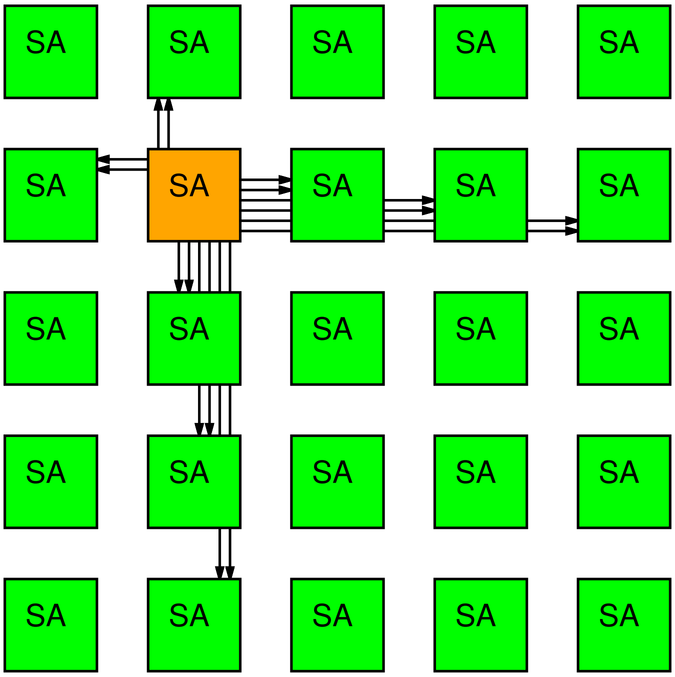


Figure 12.8: Sample Inter-Subarray Network Connections

3. Switch output in second dimension (column, row).
4. Traverse second dimension interconnect.
5. Switch to LUT input and load.

Pipelining places each of these operations in a different clock cycle. Long wire connections may merit multiple clock cycles for wire traversal – this is likely to be true for long, hierarchical connections. Short wires, particularly the nearest neighbor connections, may not always merit a separate pipeline stage for wire traversal.

I/O Connections I/O connections are treated like hierarchical network lines and are routed into and out of the subarrays in a similar manner. Each input has an associated subarray through which it may enter the switched network. Similarly, each output is associated with the crossbar output of some subarray. Device outputs are composed of time-switched input registers and load values from the network at designated timesteps like LUT inputs. Alternately, an output may look like inter-subarray pipeline register for routing in multichip systems.

Array Control Two "instruction" values are used to control the operation of each subarray on a per clock cycle basis, *timestep* and routing *context* (shown in Figure 12.6). The routing context serves as an instruction pointer into the subarray's routing memory. It selects the configuration of the crossbar and pre-crossbar multiplexors on each cycle. *timestep* denotes time events and

k	Number of LUT inputs
i	Maximum retiming depth
N_{sa_LUT}	LUTs in subarray
N_{sa_in}	External inputs to subarray
N_{sa_out}	External outputs to subarray
N_{xbar_in}	Subarray crossbar inputs
N_{xbar_out}	Subarray crossbar outputs
c_r	Number of routing contexts

Table 12.1: TSFPGA Subarray Parameters

indicates when values should be loaded from shared lines.

These two values are distinct in order to allow designs which take more microcycles to complete than they actually require contexts. The evaluation of a function will take a certain number of TSFPGA clock cycles. This time is dictated by the routed delay. For designs with large serial paths, long critical paths, or poor locality of communications, the routed delay may be large without consuming all the routing resources. For this reason, it is worthwhile to segregate the notion of timestep from the notion of routing contexts. Each routing interconnect pattern, or context, may be invoked multiple times at different timesteps during an evaluation. This allows us to have a small number of routing contexts even when the design topology necessitates a large number of timesteps.

As a trivial example, consider the case of an unfolded subarray. With full subarray interconnect there may be enough physical interconnect in a single context to actually route the complete design. However, since the design has a critical path composed of multiple LUT delays, it will take multiple microcycles to evaluate. In this case, it is not necessary to allocate separate routing contexts for each timestep as long as we segregate these two specifications into separate entities.

12.4 Architecture Parameters

The subarray composition effectively determines the makeup of a TSFPGA component. Table 12.1 summarizes the base parameters characterizing a TSFPGA subarray implementation. From these, we can calculate resource size and sharing:

$$N_{pre\_xbar\_in} = \frac{N_{sa\_in} + N_{sa\_LUT}}{N_{xbar\_in}} \tag{12.1}$$

$$N_{share\_xbar\_out} = \frac{N_{sa\_out} + k \cdot N_{sa\_LUT}}{N_{xbar\_out}} \tag{12.2}$$

$$N_{sw/LUT} = \underbrace{\frac{N_{xbar\_in} \cdot N_{xbar\_out}}{N_{sa\_LUT}}}_{\text{crossbar}} + \underbrace{\frac{N_{sa\_in} + N_{sa\_LUT}}{N_{sa\_LUT}}}_{\text{pre-crossbar mux}} \tag{12.3}$$

Assuming we need to route through one intermediate subarray, on average, the number of routing contexts, c_r, needed for full connectivity is:

$$c_r \leq N_{pre\_xbar\_in} \cdot N_{share\_xbar\_out} \tag{12.4}$$

That is, we need one context to drive each crossbar source for each crossbar sink. When we have the same number of contexts as we have total network sharing, we can guarantee to route anything. Relation 12.4 assumes that N_{sa_in} and N_{sa_out} are chosen reasonably large to support traffic to, from, and through the array. If not, the sharing of inter-subarray network lines will dominate strict crossbar sharing, and should be used instead.

In practice, we can generally get away with less contexts than implied by Relation 12.4 by selectively routing outputs and inputs as needed. When LUT inputs sharing a crossbar output also share input values or when the mapped design requires limited connectivity, less switching is needed, and the routing tasks can be completed with fewer contexts. The freedom to specify which output drives each crossbar input on a given cycle, as provided by the TSFPGA subarray, is not strictly necessary. We could have a running counter which enables each source in turn. However, with a fixed counter it would always take $N_{pre\_xbar\_in}$ cycles to pull out any particular source, despite the fact that only a few are typically needed at any time. The effect is particularly acute when we think about levelized evaluation where we may be able to simultaneously route all the LUTs whose results are currently ready and needed in a single cycle. For this reason, TSFPGA provides independent control over the pre-crossbar input mux.

In total, the number of routing bits per LUT, then, is:

$$N_{route\_bits/LUT} = c_r \cdot \left(\frac{\log_2\left(N_{pre\_xbar\_in}\right) \cdot N_{xbar\_in} + \log_2\left(N_{xbar\_in}\right) \cdot N_{xbar\_out}}{N_{sa\_LUT}} \right) \tag{12.5}$$

Additionally, each LUT has its own programmable function and matching inputs:

$$\begin{aligned} N_{bits/LUT} &= N_{logic\_bits/LUT} + N_{ir\_bits/LUT} + N_{route\_bits/LUT} && (12.6) \\ N_{logic\_bits/LUT} &= 2^k && (12.7) \\ N_{ir\_bits/LUT} &= k \cdot \lceil \log_2(i) \rceil && (12.8) \end{aligned}$$

12.5 TSFPGA Implementation Estimates

12.5.1 Area

The time-switched input register is the most novel building block in the TSFPGA design. A prototype layout by Derrick Chen was:

Time-Switched Input register with comparators ($i = 256$)	32Kλ^2
LUT multiplexor with SRAM function memory	32Kλ^2
Complete base LUT (A_{base})	160Kλ^2

Note that A_{base} contains the LUT multiplexor, LUT function memory, all $k = 4$ input registers, their associated comparators, and the comparator programming.

The amortized area per LUT then is:

$$A_{LUT} = A_{base} + N_{sw/LUT} \cdot A_{SW} + N_{route\_bits/LUT} \cdot A_{mem\_cell} \tag{12.9}$$

Using the $N_{sa\_LUT} = N_{sa\_in} = N_{sa\_out} = 64$ subarray as a reference, a version with no folding has, $N_{route\_bits/LUT} \approx 35$, $N_{sw/LUT} = 640$, making $A_{LUT} = 1800$Kλ^2, which is about 2-3$\times$ the size of typical 4-LUTs. But, as we noted above, unfolded, we have 2-3$\times$ as many switches as a conventional FPGA implementation. Also, unfolded, the expensive, matching input register is not needed.

If we fold the input and output each once, $N_{xbar\_in} = 64$, $N_{xbar\_out} = 160$, the number of switches drops to $N_{sw/LUT} = 162$. With four routing contexts ($c_r = 4$), routing bits rise to $N_{route\_bits/LUT} = 64$. The total area is $A_{LUT} \approx 640$Kλ^2, which is comparable in size with modern FPGA implementations, while providing 2-3$\times$ the total connectivity.

Focus Design Point For the sake of evaluation, we settled on a single, highly folded, design point for close inspection. From our experience with the DPGA and other VLSI efforts, we chose to use a 16×16 crossbar as the base interconnect ($N_{xbar\_in} = N_{xbar\_out} = 16$), balancing the desire to keep the crossbar compact and fast with the desire to perform as high radix switching as feasible. Per LUT switches drops to almost a trivial level, $N_{sw/LUT} = 6$. With 64 routing contexts, switching bits rises to $N_{route\_bits/LUT} = 112$. Along with the 16 bits for LUT function programming, and 32 bits for input match programming, this brings the total number of programming bits per LUT up to 160, which is comparable to conventional FPGAs (See Table 7.2). The LUT area is $A_{LUT} \approx 310$Kλ^2, or about half the size of a conventional FPGA 4-LUT.

At this size, each TSFPGA 4-LUT is effectively larger than the logical 4-LUT area in the iDPGAs of the previous chapter. The added complexity and range ($i = 256$) of the time-switched input registers is largely responsible for the greater size. The time-switched input register features, in turn, are what allow us map designs without satisfying a large number of simultaneously constraints.

12.5.2 Timing

Within the subarray, the critical path for the operating cycle of this design point contains:

1. clock to Q delay on the context address

2. Context memory read from 64-word deep memory
3. 8:1 pre-crossbar input mux
4. 16×16 crossbar traversal
5. setup time for the crossbar output flip-flops

For higher performance, the context read could be placed in its own pipeline stage. As noted, wire traversal already operate as a separate pipeline stage of its own. When wire delays begin to exceed the intra-subarray cycle delay, we can add additional pipelining to wire traversal. From simulations, it looks feasible to run with a 200 MHz microcycle. This is roughly twice the microcycle frequency for the DPGA design. The speedup here comes primarily from separating intra-subarray routing and inter-subarray routing into separate pipeline stages.

12.6 TSFPGA Fast Circuit Mapping

Traditional logic and state-element netlists can be mapped to TSFPGA for levelized logic evaluation similar to the DPGA mapping in the previous two chapters. Using this model, only the final place-and-route operation must be specialized to handle TSFPGA's time-switched operation. Of course, front-end netlist mapping which takes the TSFPGA architecture into account may be able to better exploit the architecture, producing higher performance designs.

`tspr`, our first-pass place-and-route tool for TSFPGA, performs placement by min-cut partitioning and routing by a greedy, list-scheduling heuristic. Both techniques are employed for their simplicity and mapping-time efficiency rather than their quality or optimality. The availability of adequate switching resource, expandable by allocating more configuration contexts, allows us to obtain reasonable performance with these simple mapping heuristics. For the most part, the penalty for poor quality placement and routing in TSFPGA is a slower design, not an unroutable design.

Timesteps and Contexts It is again worth noting that the number of *timesteps* and routing *contexts* are dictated by different properties of the mapped network.

The topology of the circuit will determine the critical path length, or the number of logical LUT delays between the inputs and outputs of the circuit. This critical path length is one lower bound on the number of timesteps required to evaluate a circuit. However, once placed onto subarrays, there is another, potentially longer, bound, the *distance delay* through the network. The distance delay is the length of the longest path through the circuit including the cycles required for inter-subarray routing. If all the LUTs directly along every critical path can be mapped to a single subarray, it is possible that the distance delay is equal to the critical path length. However, in general, the placed critical path crosses subarrays resulting in a longer distance delay. The quality of the distance delay is determined entirely during the placement phase.

The actual routed delay is generally larger than the distance delay because of contention. That is, if the architecture does not provide enough physical resources to route all the connections in the placed critical path simultaneously, or the if the greedy routing algorithms allocates those resources suboptimally, signals may take additional microcycles to actually be routed.

Placement Partitioning is based on the Fiduccia-Mattheyses min-cut heuristic [FM82]. Netlists are recursively partitioned along TSFPGA dimension boundaries. That is, for a simple, two-dimensional network topology, as shown in Figure 12.8, the design is first partitioned for columns, then columns are partitioned into subarrays. For larger networks, top-level row and column partitioning would precede low-level row and column partitioning. The Fiduccia-Mattheyses heuristic aims to minimize the size of the cut net, but does not, directly, minimize the effect of cuts on circuit delay. As a consequence partitioning is useful in reducing the routing congestion contribution to routed delay, but does not explicitly try to minimize the distance delay.

For the fastest mapping times, no sophisticated placement is done. Circuit netlists are packed directly into subarrays as they are parsed from the netlists. Such oblivious placement may create unnecessarily long paths by separating logically adjacent LUTs and may create unnecessary congestion by not grouping tightly connected subgraphs. However, with enough routing contexts the TSFPGA architecture allows us to succeed at routing with such poor placement.

Routing Routing is directed by the circuit netlist topology using a greedy, list-scheduling heuristic. At the start, a ready list is initialized with all inputs and flip-flop outputs. Routing proceeds by picking the output in the ready list which is farthest from the end set of primary outputs and flip-flop inputs. Routing a signal implies reserving switch capacity in each context and timestep involved in the route. If a route cannot be made starting at the current evaluation time, the starting timestep for the route is incremented and the route search is repeated. Currently, only minimum distance routes are considered. Assuming adequate context memory, every route will eventual succeed. Once a route succeeds, any LUT outputs which are then ready for routing are added to the ready list. In this manner, the routing algorithm works through the design netlist from inputs to outputs, placing and routing each LUT as it is encountered.

Modulo Context Routing The total number of contexts is dictated by the amount of contention for shared resources. Since some timesteps may route only a few connections, a routing context may be used at multiple timesteps. In the simplest case, switches in a routing context not used during one timestep may be allocated and used during another. In more complicated cases, a switch allocated in one context can be reused with the same setting in another routing context. This is particularly useful for the inter-subarray routing of patterns, but may be computationally difficult to exploit.

Our experimental mapping software can share contexts among routing timesteps by modulo context assignment. That is context n mod max_ctx is used to route on timestep n. As we will see in the next section, this generally allows us to reduce the number of required contexts. Further context reduction is possible when we are willing to increase the number of timesteps required for evaluation. More sophisticated sharing schemes are likely to be capable of producing better results.

Design	Netlist Size		Target Array			Quick Map				Performance Map			Best
	LUTs	IOs	Tiles (SA)	LUTs	IOs	Time (sec.)	Delays LUT	Dist	Rte	Time (sec.)	Delays Dist	Rte	Map Rte
5xp1	46	17	2×1	128	32	0.05	11	14	19	0.67	14	19	18
9sym	123	10	2×1	128	32	0.18	8	15	29	4.02	15	25	23
9symml	108	10	2×1	128	32	0.15	9	17	27	9.48	13	24	21
C499	85	73	3×2	384	96	0.15	11	22	34	3.06	23	33	25
C880	176	86	3×2	384	96	0.34	22	44	48	9.67	31	36	32
alu2	169	16	2×2	256	64	0.28	20	43	45	10.00	43	47	34
apex6	248	234	4×4	1024	256	0.69	10	27	37	34.06	19	23	23
apex7	77	86	3×2	384	96	0.16	8	19	24	3.15	16	19	19
b9	46	62	2×2	256	64	0.08	8	15	21	0.74	12	14	14
clip	121	14	2×1	128	32	0.19	10	23	29	5.08	18	26	23
cordic	367	25	3×2	384	96	0.98	14	47	60	26.59	40	43	39
count	46	51	2×2	256	64	0.10	17	26	27	1.22	24	25	21
des	1267	501	6×6	2304	576	6.30	14	51	66	626.40	37	43	35
e64	230	130	3×3	576	144	0.63	10	29	40	18.90	32	33	26
f51m	45	16	2×1	128	32	0.07	18	21	22	0.05	21	22	22
misex1	20	15	1×1	64	16	0.03	7	10	16	0.02	10	13	13
misex2	38	43	2×2	256	64	0.07	9	15	18	0.95	15	16	15
rd73	105	10	2×1	128	32	0.14	11	18	27	4.35	14	22	21
rd84	150	12	2×2	256	64	0.24	10	30	35	4.88	26	30	24
rot	293	242	4×4	1024	256	0.76	17	44	45	21.14	28	31	31
sao2	73	14	2×1	128	32	0.11	10	14	22	1.79	13	20	18
vg2	60	33	2×2	256	64	0.11	10	17	23	1.07	14	19	19
z4ml	8	11	1×1	64	16	0.03	8	11	12	0.02	11	12	12

Run times given are in seconds on a SparcStation 20 Model 71 (rated at 125 SPECint92).

Table 12.2: TSFPGA Mappings for MCNC Circuit Benchmarks

12.7 Circuit Mapping

In this section we show the results of mapping the same MCNC benchmark circuit suite used for the DPGA in the previous two chapters to TSFPGA. These benchmarks are mapped viewing TSFPGA simply as an FPGA with time-switched interconnect, ignoring the way one might tailor tasks to take full advantage of the architecture.

Table 12.2 shows the results of mapping the benchmark circuits to TSFPGA. The same area mapped circuits from `sis` and `Chortle` used in Sections 10.5.3 and 11.4 were used for this mapping. Each design was mapped to the smallest rectangular collection of subarray tiles which supported both the design's I/O and LUT requirements. Quick mapping does oblivious placement while the performance mapping takes time to do partitioning. Both the quick and performance map

Design	Target Ratios LUTs	IOs	Quick Map Delay Ratios Dist	Route	Performance Map Delay Ratios Dist	Route	Best Route Ratio
	% used						
5xp1	0.36	0.53	1.27	1.73	1.27	1.73	1.64
9sym	0.96	0.31	1.88	3.62	1.88	3.12	2.88
9symml	0.84	0.31	1.89	3.00	1.44	2.67	2.33
C499	0.22	0.76	2.00	3.09	2.09	3.00	2.27
C880	0.46	0.90	2.00	2.18	1.41	1.64	1.45
alu2	0.66	0.25	2.15	2.25	2.15	2.35	1.70
apex6	0.24	0.91	2.70	3.70	1.90	2.30	2.30
apex7	0.20	0.90	2.38	3.00	2.00	2.38	2.38
b9	0.18	0.97	1.88	2.62	1.50	1.75	1.75
clip	0.95	0.44	2.30	2.90	1.80	2.60	2.30
cordic	0.96	0.26	3.36	4.29	2.86	3.07	2.79
count	0.18	0.80	1.53	1.59	1.41	1.47	1.24
des	0.55	0.87	3.64	4.71	2.64	3.07	2.50
e64	0.40	0.90	2.90	4.00	3.20	3.30	2.60
f51m	0.35	0.50	1.17	1.22	1.17	1.22	1.22
misex1	0.31	0.94	1.43	2.29	1.43	1.86	1.86
misex2	0.15	0.67	1.67	2.00	1.67	1.78	1.67
rd73	0.82	0.31	1.64	2.45	1.27	2.00	1.91
rd84	0.59	0.19	3.00	3.50	2.60	3.00	2.40
rot	0.29	0.95	2.59	2.65	1.65	1.82	1.82
sao2	0.57	0.44	1.40	2.20	1.30	2.00	1.80
vg2	0.23	0.52	1.70	2.30	1.40	1.90	1.90
z4ml	0.12	0.69	1.38	1.50	1.38	1.50	1.50
Average	0.46	0.62	2.08	2.73	1.80	2.24	2.01

All delay ratios are $\frac{\{\text{Dist,Route}\}\text{ Delay}}{\text{LUT Delay}}$

Table 12.3: TSFPGA Mappings for MCNC Circuit Benchmarks (Ratios)

use the same, greedy routing algorithm. As noted in Section 12.6, fairly simple placement and routing techniques are employed, so higher quality routing results are likely with more sophisticated algorithms. Quick mapping can route designs in the order of seconds, while performance mapping runs in minutes. The experimental mapping software implementation has not been optimized for performance, so the times shown here are, at best, a loose upper bound on the potential mapping time. The "Best map" results in Table 12.2 summarize the best results seen over several runs of the "performance" map.

Table 12.3 shows usage and time ratios derived from Table 12.2. All of the mapped delay ratios are normalized to the number of LUT delays in the critical path. We see that the quick mapped

Design	Min Delay				Min Contexts			
	delay	**# ctx**	**Δ ctx**	**Δ %**	**delay**	**# ctx**	**Δ ctx**	**Δ %**
5xp1	19	15	4	0.21	28	12	7	0.37
9sym	25	21	4	0.16	38	17	8	0.32
9symml	24	20	4	0.17	41	19	5	0.21
C499	33	19	14	0.42	48	16	17	0.52
C880	36	29	7	0.19	54	19	17	0.47
alu2	47	43	4	0.09	107	28	19	0.40
apex6	23	19	4	0.17	34	16	7	0.30
apex7	19	14	5	0.26	25	11	8	0.42
b9	14	10	4	0.29	18	8	6	0.43
clip	26	22	4	0.15	42	19	7	0.27
cordic	43	39	4	0.09	106	34	9	0.21
count	25	21	4	0.16	31	14	11	0.44
des	43	39	4	0.09	67	35	8	0.19
e64	33	30	3	0.09	58	28	5	0.15
f51m	22	18	4	0.18	40	13	9	0.41
misex1	13	10	3	0.23	16	8	5	0.38
misex2	16	12	4	0.25	20	9	7	0.44
rd73	22	17	5	0.23	32	14	8	0.36
rd84	30	26	4	0.13	52	25	5	0.17
rot	31	27	4	0.13	48	21	10	0.32
sao2	20	16	4	0.20	33	15	5	0.25
vg2	19	15	4	0.21	29	14	5	0.26
z4ml	12	4	8	0.67	16	3	9	0.75

Table 12.4: Modulo Context Sharing for MCNC Benchmarks

delays are almost 3× the critical path LUT delay, while the performance mapped delays are closer to 2×. As we noted in Section 12.5.2, the basic microcycle on TSFPGA is half that on the DPGA, suggesting that the performance mapped designs achieve roughly the same average latency as full, levelized evaluation on the DPGA. We can see from the distance delay averages that placement dictated delay is responsible for a larger percentage of the difference between critical path delay and routed delay. However, since the routed delay is larger than the distance delay, network resource contention and suboptimal routing are partially responsible for the overall routed delay time.

Context Compression As noted in Section 12.6 we can use modulo context assignment to pack designs into fewer routing contexts at the cost of potentially increasing the delay. Table 12.4 shows the number of contexts into which each of the designs in Table 12.2 can be packed both with and without expanding their delay. Figure 12.9 shows how routed delay of several benchmarks increases as the designs are packed into fewer routing contexts.

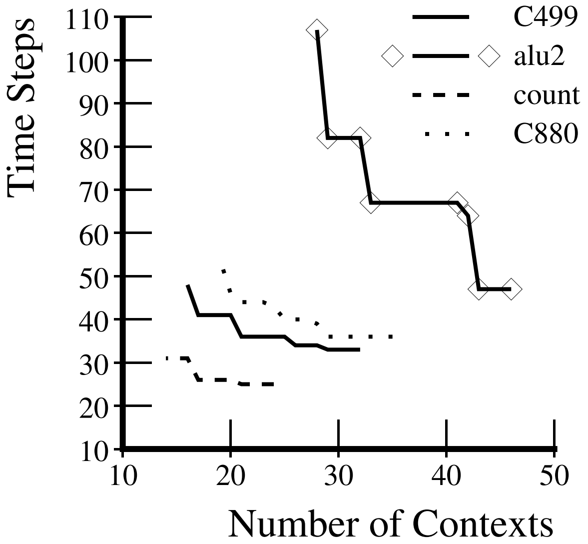


Figure 12.9: Sample Delay Increases with Context Packing

12.8 Related Work

Dharma [BCK93] time-switched two monolithic crossbars. It made the same basic reduction as the DPGA – that is, rather than having to simultaneously route all connections in the task, one only needed to route all connections on a single logical evaluation level. To make mapping fast, Dharma used a single monolithic crossbar. For arrays of decent size, the full crossbar connecting all LUTs at a single level can still be prohibitively large. Further, Dharma had a rigid assignment of LUT evaluation and hence routing resources to levels. As we see in TSFPGA, it is not always worth dedicating all of one's routing resources, an entire routing context, to a single evaluation timestep. Dharma deals with retiming using a separate flow-through crossbar. While the flow through retiming buffers are cheaper than full LUTs, they still consume active routing area which is expensive. As noted in Chapter 11, it is more area efficient to perform retiming, or temporal transport, in registers than to consume active interconnect.

VEGA [JL95], noted in Sections 10.2 and 11.5, uses a 1024 deep context memory, essentially eliminating the spatial switching network, and uses a register file to retime intermediate data. The VEGA architecture allows similar partitioning and greedy routing heuristics to be used for mapping. However, the heavy multicontexting and trivial network in VEGA means that it achieves its simplified mapping only at the cost of a 1024$\times$ reduction in active peak capacity and 100$\times$ penalty in typical throughput and latency over traditional FPGA architectures.

PLASMA [ACC$^+$96] was built for fast mapping of logic in reconfigurable computing tasks. It uses a hierarchical series of heavily populated crossbars for routing on chip. The existence of rich, hierarchical routing makes simple partitioning adequate to satisfy interconnect constraints. The heavily populated crossbars make the routing task simple. The basic logic primitive in PLASMA is the PALE, which is roughly a 2-output 6-LUT. The PLASMA IC packs 256 PALEs into 16.2mm$\times$16.2mm in a 0.8μ CMOS process, or roughly 1.6Gλ^2. This comes to 6.4Mλ^2 per PALE. If we generously assume each PALE is equivalent to 8 4-LUTs, the area per 4-LUT is 800Kλ^2, which is commensurate with conventional FPGA implementations, or about 2-2.5$\times$ the size for the TSFPGA design point described above. In practice, the TSFPGA LUT density will be even greater since it is not always the case that 8 4-LUTs can be packed into each PALE. PLASMA's size is a direct result of the fact that it builds all of its rich interconnect as active switches and wires. Routing is not pipelined on PLASMA, and critical paths often cross chip boundaries. As a result, typical PLASMA designs run with high latency and a moderately slow system clock rate (1-2 MHz). This suggests a time-switched device with a smaller amount of physical interconnect, such as TSFPGA, could provide the same level of mapping speed and mapped performance in substantially less area.

Virtual Wires [BTA93] employs time-multiplexing to extract higher capacity out of the I/O and inter-chip network connections only. Virtual Wires folds the FPGA and switching resources together, using FPGAs for inter-FPGA switching as well as logic. Since Virtual Wires is primarily a technique used on top of conventional FPGAs, it does accelerate the task of routing each individual FPGA or provide any greater use of on-chip active switching area.

Li and Cheng's **Dynamic FPID** [LC95] is a time-switched Field-Programmable Interconnect Device (FPID) for use in partial-crossbar interconnection of FPGAs. Similarly, they increase switching capacity, and hence routability and switch density, by dynamically switching a dedicated FPID.

UCSB researchers [cLCWMS96] consider adding a second routing context to a conventional FPGA routing architecture. Using a similar circuit benchmark suite, they find that the second context reduces wire and switching requirements by 30%. Since they otherwise use a conventional FPGA architecture, there is no reduction in mapping complexity for their architecture.

12.9 Conclusions

We have developed a new, programmable gate-array architecture model based around time-switching a modest amount of physical interconnect. The model defines a family of arrays with varying amounts of active interconnect. The key, enabling feature in TSFPGA is an input register which performs a wide range of signal retiming, freeing the active interconnect from performing data retiming or conveying data at rigidly defined times. Coupling the flexible retiming with reusable interconnect, we remove most of the constraints which make the place and route task difficult on conventional FPGA architectures. Consequently, even large designs can be mapped onto a TSFPGA array in seconds. More sophisticated mapping can be used, at the cost of longer mapping times, to achieve the lowest delay and best resource utilization. We demonstrated the viability of this fast mapping scheme by developing experimental mapping software for one design point and mapping traditional benchmark circuits onto these arrays. At the heavily time-switched design point which we explored in detail, the basic LUT size is half that of a conventional FPGA LUT while mapped design latency is comparable to the latency on fully levelized DPGAs.

12.10 Open Issues

At this point, we have left a number of interesting issues associated with TSFPGA unanswered.

- Performance using traditional place and route strategies – The fast mapping which we used above employs fast heuristics which are purposely limited to linear mapping complexity. Traditional mapping software uses different techniques, such as simulated annealing, which will consider simultaneous constraints to minimize resource usage and routed path length. It will be worthwhile to understand how well tasks can be mapped to members of the TSFPGA architecture when we are willing to take the time to perform a quality mapping job. In normal usage, one might use the fast mapping during design development and debug, then use the slower, higher quality mapping once a design becomes stable.

- Explore defined architectural space – We have focussed on a single point in the defined architectural space. It will be worthwhile to map tasks across various architectural points to determine the level of connectivity required to meet typical throughput and latency requirements, and to determine the most area efficient implementation points.

- Multichip extension – The specifics explored here focus on single-chip implementations, but there is a natural extension to multiple chip systems. The dimensional routing organization used on-chip should extend between chips when an array of TSFPGA components is employed. Partitioning, placement, and routing amongst components will be very similar to partitioning, placement, and routing amongst the subarrays on a single TSFPGA component. The boundary i/o will provide a more severe bottleneck between subarrays on distinct chips, requiring heavier time-multiplexing. Inter-TSFPGA routes will require more pipeline stages than inter-subarray routes.