CHAPTER 20

FAST COMPILATION TECHNIQUES

Ken Eguro, Scott Hauck
Department of Electrical Engineering
University of Washington

Most users rely on sophisticated CAD tools to implement their circuits on field-programmable gate arrays (FPGAs). Unfortunately, since each of these tools must perform reasonably complex optimization, the entire process can take a long time. Although fairly slow compilation is fine for the majority of current FPGA users, there are many situations that demand more efficient techniques. Looking into the future, we see that faster CAD tools will become necessary for many different reasons.

FPGA scaling. Modern reconfigurable devices have a much larger capacity compared to those from even a few years ago, and this trend is expected to continue. To handle the dramatic increase in problem size, while maintaining current usability and compilation times, smarter and more efficient techniques are required.

Hardware prototyping and logic emulation systems. These are very large multi-FPGA systems used for design verification during the development of other complex hardware devices such as next-generation processors. They present a challenging CAD problem both because of the sheer number of FPGAs in the system and because the compilation time for the design is part of the user's debug cycle. That is, the CAD tool time directly affects the usability of the system as a whole.

Instance-specific design. Instance-specific designs are applications where a given circuit can only solve one particular occurrence of a problem. Because of this, every individual hardware implementation must be created and mapped as the problems are presented. Thus, the true solution time for any specific example includes the netlist compilation time.

Runtime netlist compilation. Reconfigurable computing systems are often constructed with an FPGA or an array of FPGAs alongside a conventional processor. Multiple programs could be running in the system simultaneously, each potentially sharing the reconfigurable fabric. In some of the most aggressive systems, portions of a program are individually mapped to the FPGA while the instructions are in flight. This creates a need for almost real-time compilation techniques.

For each of these systems, the runtime of the CAD tools is a clear concern. In this chapter, we consider each scenario and cover techniques to accelerate the

various steps in the mapping flow. These techniques range from fairly cost-neutral optimizations that speed the CAD flow without greatly impacting circuit quality to more aggressive optimizations that can significantly accelerate compilation time but also appreciably degrade mapping quality.

FPGA scaling

The mere scaling of VLSI technology itself has created part of the burden for conventional FPGA CAD tools. Fulfilling Moore's Law, improvements in lithography and manufacturing techniques have radically increased the capabilities of integrated circuits over the last four decades. Of course, just as these advancements have increased the performance of desktop computers, they have increased the logic capacity of FPGAs. Correspondingly, the size of desired applications has also increased. Because of this simultaneous scaling across the industry, reconfigurable devices and their applications become physically larger at approximately the same rate that general-purpose processors become faster.

Unfortunately, this does not mean that the time required to compile a modern FPGA design on a modern processor stays the same. Over a particular period of time, desktop computers and compute servers will become twice as fast and, concurrently, FPGA architectures and user circuits will double in size. Since the complexity of many classical design compilation techniques scale super-linearly with problem size, however, the relative runtime for mapping contemporary applications using contemporary machines will naturally rise.

To continue to provide reasonable design compilation time across multiple FPGA generations, changes must be made to prevent a gap between available computational power and netlist compilation complexity. However, although application engineers depend on compilation times of at most a few hours to meet fast production timelines, they also have expectations about the usable logic block density and achievable clock frequency for their applications. Thus, any algorithmic improvements or architectural changes made to speed up the mapping process cannot come at the cost of dramatically increased critical-path timing or reduced mapping density.

Hardware prototyping and logic emulation systems

The issue of nonscalable compilation is even more obvious in large prototyping or logic emulation systems. These devices integrate multiple FPGAs into a single system, harnessing tens to thousands. As Chapter 30 discusses in more detail, the fundamental size of typical circuits on these architectures suggests fast mapping techniques. However, even more critical, the compilation time of the netlists themselves may become a limiting factor in the basic usefulness of the entire system.

Hardware prototyping is often employed for many reasons. One of the greatest advantages of hardware emulation over software simulation is its extremely fast validation time. During the design and debug cycle of hardware development, hundreds of thousands of test vectors may be applied to ensure that a given implementation complies with design specifications. Although an FPGA-based prototyping system cannot be expected to achieve anywhere near the clock rate of the dedicated final product, the sheer volume of tests that need to be performed

every time a change is made to the system makes software simulation too slow to have inside the engineering design loop. That said, software simulation code can easily accommodate design updates and, more important, the changes have a predictable compilation time of minutes to hours, not hours to days. Still, since reconfigurable logic emulation systems maintain such a runtime advantage over software simulation, prototyping designers are willing to exchange some of the classical FPGA metrics of implementation quality, critical-path timing, and logical density for faster and more predictable compilation time.

Instance-specific design

Similar to logic emulation systems, the netlist compilation time of instance-specific circuits can greatly affect the overall value of an FPGA-based implementation. For example, although Boolean satisfiability is NP-complete, the massive parallelism offered by reconfigurable fabrics can often solve these problems extremely quickly—potentially on the order of milliseconds (see Chapter 29). Unfortunately, these FPGA implementations are equation-specific, so the time required to solve any given SAT problem is not determined by the vanishingly short runtime of the actual mapped circuit running on a reconfigurable device, but instead is dominated by the compilation time required to obtain the programming bitstream in the first place—potentially on the order of hours.

Because of this reliance on netlist compilation, the Boolean satisfiability problem differs strongly from more traditional reconfigurable computing applications for two reasons.

First, if we disregard compilation time, FPGA-based SAT solvers can obtain two to three orders of magnitude better performance than software-based solutions. Thus, the critical path and, by extension, the overall quality of the mapping in the classical sense are virtually irrelevant. As long as compilation results in *any* valid mapping, the vast majority of the performance benefit will be maintained. While some effort is required to reliably produce routable circuits, we can make huge concessions in terms of circuit quality in the name of speeding compilation. Mappings that are quickly produced, but possibly slow, will still drastically improve the overall solution runtime.

Second, features of the SAT problem itself suggest that application-specific approaches might be worthwhile. For example, because SAT solvers typically have very structured forms, fast SAT-specific CAD tools can be created. One possibility is the use of preplaced and prerouted SAT-specialized macros that simply need to be assembled together to create the overall system. To extend the concept of application-specialized tuning to its logical end, architectural changes can even be made to the reconfigurable fabric itself to make the device particularly amenable to simple, fast mapping techniques. That said, the large engineering effort this would involve must be weighed against the possible benefits.

Runtime netlist compilation

All reconfigurable computing systems have a certain amount of overhead that eats away at their performance benefit. Although kernel execution might be blindingly fast once started on the reconfigurable logic, its overall benefit is limited by the

need to profile operations, transfer data, and configure or reconfigure the FPGA. Reconfigurable computing systems that use dynamically compiled applications have the additional burden of runtime netlist compilation. These systems only map application kernels to the hardware during actual system execution, in the hope that runtime data, such as system loads, resource availability, and execution profiles, can improve the resultant speedups provided by the hardware. Their almost real-time requirements demand the absolutely fastest compilation techniques. Thus, even more so than instance-specific designs, these systems are only concerned with compilation speed.

Mapping stages

When evaluating mapping techniques for high-speed circuit compilation, we have to remember that the individual tools are part of a larger system. Therefore, any quality degradation in an early stage may not only limit the performance of the final mapping, but also make subsequent compilation problems more difficult. If these later mapping phases are more difficult, they may require a longer runtime, overwhelming the speedups achieved in earlier steps. For example, a poor-quality placement obtained very quickly will likely make the routing problem harder. Since we are interested in reducing the runtime of the compilation phase as a whole, we must ensure that we do not simply trade placement runtime for routing runtime. We may even run the risk of increasing total compilation time, since a very poor placement might be impossible to route, necessitating an additional placement and routing attempt.

Although logic synthesis, technology mapping, and logic block packing are considered absolutely necessary parts of a modern, general-use FPGA compiler flow, the majority of research into fast compilation has been focused on efficient placement and routing techniques. Not only do the placement and routing phases make up a large portion of the overall mapping runtime, in some cases the other steps can be considered either unsuitable or unnecessary to accelerate. Sometimes high-level synthesis and technology mapping may be unnecessary because designs are assumed to be implemented in low-level languages, or it is assumed that they can be performed offline and thus outside the task's critical path. Furthermore, although logic synthesis and technology mapping can be very difficult problems by themselves, they are also common to all hardware CAD tools—not just FPGA-based technologies. On the other hand, placement and routing tools for reconfigurable devices have to deal with architectural restrictions not present in conventional standard cell tools, and thus generally must be accelerated with unique approaches.

20.1 ACCELERATING CLASSICAL TECHNIQUES

An obvious starting point to improve the runtime of netlist compilation is to make minor algorithmic changes to accelerate the classical techniques already in use. For example, simulated annealing placement has some obvious parameters that can be changed to reduce overall runtime. The initial annealing temperature

can be lowered, the freezing point can be increased, the cooling schedule can be accelerated, or the number of moves per iteration can be reduced. These approaches all tend to speed up the annealing, but at some cost to placement quality.

20.1.1 Accelerating Simulated Annealing

Because of the adaptive nature of modern simulated annealing temperature schemes, any changes made to the structure of the cooling schedule itself can have unreliable runtime behavior. Not only have the settings of initial and final temperatures been carefully selected to thoroughly explore the solution space, changing these values may dramatically affect final placement quality while still not guaranteeing satisfactorily shorter runtime.

As described in Chapter 14, VPR updates the current temperature based on the fraction of moves accepted out of those attempted during a given iteration. Thus, decreasing the initial temperature cuts off the phase in which sweeping changes can easily occur early in the annealing. Simply starting the system at a lower initial temperature may cause the annealing to compensate by lingering longer at moderately high temperatures. Similarly, modifying the cooling schedule to migrate toward freezing faster fundamentally goes against the basic premise of simulated annealing itself. This will have an unpredictable, and likely undesirable, effect on solution quality.

It is generally accepted that the most predictable way to scale simulated annealing effort is by manipulating the number of moves attempted per temperature iteration. For example, in VPR the number of moves in a given iteration is always based on the size of the input netlist: $O(n^{1.33})$. The annealing effort is simply adjusted by scaling up or down the multiplicative constant portion of this value. In VPR, the "fast" placement option simply divides the default value by 10, which in testing indeed reduces the overall placement time by a factor of 10 while affecting final circuit quality by less than 10 percent [3]. Furthermore, as shown by Mulpuri and Hauck [12], simply changing the number of moves per iteration allows a continuous and relatively predictable spectrum of placement effort versus placement quality results.

Haldar and colleagues [11] exploited a very similar phenomenon to reduce mapping time by distributing the simulated annealing effort across multiple processors. In the strictest sense, simulated annealing is very difficult to parallelize because it attempts sequential changes to a given placement in order to slowly improve the overall wirelength. To be most faithful to this process while attempting multiple changes simultaneously, different processors must try non-overlapping changes to the system; otherwise, multiple processors may try to move the same block to two different locations or two different blocks to the same location. Not only is this type of coordination typically very difficult to enforce, it also generally requires a large amount of communication between processors. Since all processors begin each move operating on the same placement, they all must communicate any changes that are made after each step. However, a slightly less faithful but far simpler approach can take advantage of

the idea that reducing the number of moves attempted per temperature iteration can gracefully reduce runtime.

In this case, all of the processors agree upon a single placement to begin a temperature iteration. At this point, though, each processor performs simulated annealing independently of the others. To reduce the overall runtime, given N processors, each only attempts $1/N$ of the originally intended moves per iteration. At the end of the iteration, the placements discovered by all of the processors are compared and the best one is broadcasted to the rest for use during the next iteration. This greatly reduces the communication overhead and produces nearly linear speedup for two to four processors while reducing placement quality by only 10 to 25 percent [11].

Wrighton and DeHon [19] also parallelized the simulated annealing process, but approached the problem in a completely different manner. In this case, instead of attempting to develop parallel software, they actually configure an FPGA to find its own placement for a netlist. They divide a large array into distinct processing elements that will each keep track of one node in a small netlist. In their testing, the logic required to trace the inputs and outputs of a single LUT required approximately 400 LUTs. Because every processing element represents the logic held at a single location in the array, a large emulation system consisting of approximately 400 FPGAs can place a netlist for one device at a time, or one large FPGA can place a netlist requiring approximately 1/400 of the array.

Each processing element is responsible for keeping track of both the block in the netlist currently mapped to that location and the position of the sinks of the net sourced by this block. During a given timestep, each processing element determines the wirelength of its output net by evaluating the location of all of its sinks; the entire system is then perturbed in parallel by allowing each location to negotiate a possible swap with its neighbors. Just as in conventional simulated annealing, good moves are always accepted and bad moves are accepted with a probability dependent on the annealing temperature and how much worse the move makes the system as a whole. Similarly, although swaps can only be made one nearest neighbor to another, any block can eventually migrate to any other location in the array through multiple swaps. The system avoids having two blocks attempt to occupy the same location by always negotiating swaps pairwise.

As shown in Figure 20.1, a block negotiates a swap with each of its neighbors in turn. Phases 1 and 2 may swap blocks to the left or right, while phases 3 and 4 may swap with a neighbor above or below.

We should note that although very similar to the classical simulated annealing model, this arrangement does not necessarily calculate placement cost in the same way. The net bounding box calculated at each timestep cannot take into account the potential simultaneous movement of all the other blocks to which it is connected. That said, whatever inaccuracies might be introduced by this computation difference are relatively small.

Of much greater importance is the problem caused by communication bandwidth. It is possible that in a given timestep every processing element decides to swap with its neighbor. If this is the case, the location of all sinks will change.

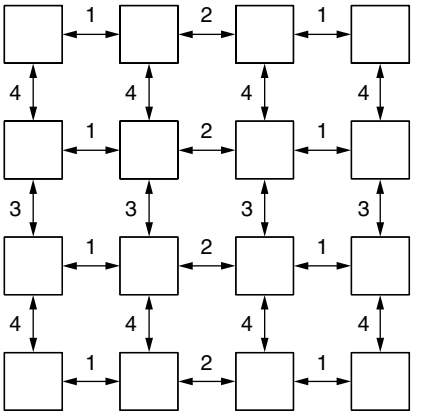

FIGURE 20.1 ■ Swap negotiation in hardware-assisted placement. (*Source*: Based on an illustration in Wrighton and DeHon [19]).

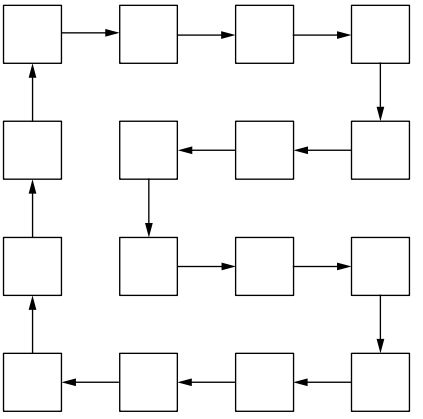

FIGURE 20.2 ■ Location update chain. (*Source*: Based on an illustration in Wrighton and DeHon [19]).

To keep completely consistent recordkeeping with conventional simulated annealing, this requires each processing element to notify its nets' sources of the block's new location. Of course, this creates a huge communication overhead. However, this can be avoided if the processing elements are allowed to calculate wirelength based on stale location information.

As shown in Figure 20.2, instead of a huge broadcast each time a block is relocated, position information marches through the system in a linear fashion. As blocks are moved during the annealing process, new positions for each one are communicated to other blocks via a dedicated location update chain. Thus, if the system has N processing elements, it might take N clock cycles before all relevant processing elements see the new placement of that block. Since the

processing elements are still calculating further moves, this means up to *N* cycles of stale data. Because of these inaccuracies, compared with a fast VPR run, this hardware-based simulated annealing system generally requires 36 percent more routing tracks to implement the same circuits. However, it also is three to four orders of magnitude faster.

As mentioned earlier, classical simulated annealing techniques have been very carefully tuned to produce high-quality placements. Most of the methodologies we have covered to accelerate simulated annealing rely on reducing the number of moves attempted. Thus, while they can produce reasonable placements quickly for current circuits, they do not necessarily perform well for all applications.

Mulpuri and Hauck [12] demonstrated that, while we may be able to reduce the number of moves per temperature iteration by a factor of 10 with little effect on routability, if we continue to reduce the placement effort, the quality of the placement drops off severely. The conclusion to be drawn is that, acceleration approaches, although reasonable for dealing with FPGA scaling in the short term, are not a permanent solution. Applying them on increasing netlist and device sizes will eventually lead to worse and worse placements, and, furthermore, they simply do not have the capability to produce useable placements quickly enough for either runtime netlist compilation or most instance-specific circuits.

On the other hand, hardware-assisted simulated annealing seems far more promising. Although this technique introduces some inaccuracy in cost calculation because of both simultaneously negotiated moves and stale location information, the effect of these factors is relatively predictable. The error introduced by simultaneous moves will always be relatively small because all swaps are performed between nearest neighbors. Also, the error introduced by stale location information scales linearly with netlist size. This means not only that such information will likely cause the placement quality to degrade gracefully but also that we can reduce this inaccuracy relatively easily by adding additional update paths, perhaps even a bidirectional communication network that quickly informs both forward and backward neighbors of a moved element. Since we hope that the majority of nets will cover a relatively small area, this should considerably reduce inaccurate cost calculation due to stale location information.

These trade-offs make hardware-assisted annealing an interesting possibility. Although it may impose a significant quality cost, that cost may not grow with increased system capacity, and it may be one of the only approaches that provide the drastic speedups necessary for both runtime netlist compilation and instance-specific circuits. This may make it of particular interest for future nanotechnology systems (see Chapter 38).

20.1.2 Accelerating PathFinder

Just as in placement, minor alterations can be made to classical routing algorithms to improve their runtime. Some extremely simple modifications may speed routing without affecting overall quality, or they may reduce routability in a graceful and predictable manner. Swartz et al. [15] suggest sorting the nets to be routed in order of decreasing fanout instead of simply arbitrarily. Although

high fanout nets generally make up a small fraction of a circuit, they typically monopolize a large portion of the routing runtime. By routing these comparatively difficult nets first in a given iteration, they may be presented with the lowest congestion cost and thus take the most direct and easily found paths. Lower fanout nets tend to be more localized, so they can deal with congestion more easily and their search time is comparatively smaller. This tends to speed overall routing, but since no changes are made to the actual search algorithm, it is not expected to affect routability.

Conversely, Swartz et al. [15] also suggest scaling present sharing and history costs more quickly between routing iterations. As discussed in Chapter 17, PathFinder gradually increases the cost of using congested nodes to discourage sharing over multiple iterations. Increasing present sharing and history costs more aggressively emphasizes removing congestion over route exploration. This may potentially decrease achievable routability, but the system may converge on a legal routing more quickly.

One of the most effective changes that can be made to conventional Dijkstra-based routing approaches is limiting the expansion of the search. Ignoring congestion, in most island-style FPGAs it is unnecessary for a given net to use routing resources outside the bounding box formed by its terminals. Of course, congestion must be resolved to obtain a feasible mapping, but given the routing-rich nature of modern reconfigurable devices, and assuming that routing is performed on a reasonable placement, the area formed by a net's bounding box is most likely to be used.

However, traditional Dijkstra's searches expand from the source of a net evenly in all directions. Given that the source of a 2-terminal net must lie on the edge of the bounding box, this is obviously wasteful since, again ignoring congestion costs, the search essentially progresses as concentric rings—most of which lie in the incorrect direction for finding the sink. As shown in Figure 20.3, it is unlikely that a useful route will require such a meandering path. If we would like to find

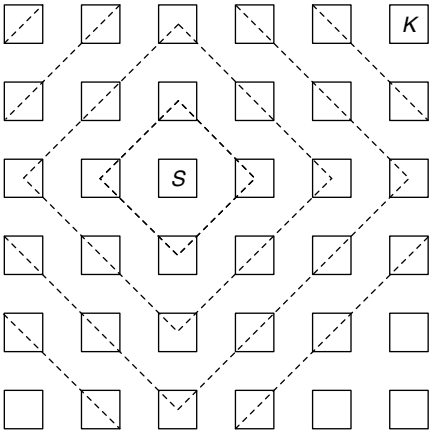

FIGURE 20.3 ■ A conventional routing search wave.

a route between blocks S and K, it is most likely that we will be able to find a direct route between them. Thus, we should direct the majority of our efforts upward and to the right before exploring downward or to the left. As described in Chapter 17, this is the motivation for adding A^* enhancements to the PathFinder algorithm. However, this concept can be taken even further by formally preventing searches from extending very far beyond the net's bounding box.

According to Betz et al. [3], a reasonable fixed limitation can prevent an exploration from visiting routing channels more than three steps outside of a net's bounding box. Although this technique may degrade routability under conditions of very high congestion, such situations may not be encountered. An architecture might have sufficient resources so that high-stress routing situations are never created, particularly in scenarios where the user is willing to reduce the amount of logic mapped to an FPGA to improve compilation runtimes.

Slightly more difficult to manage is the case of multi-terminal nets. Although the scope of a multisink search as a whole may be limited by the net's bounding box, this only alleviates one source of typically unnecessary exploration. PathFinder generally sorts the sinks of a multi-terminal net by Manhattan distance. However, each time a sink is discovered, the search for the next sink is restarted based on the entire routing tree found up to that point. As shown in Figure 20.4, this creates a wide search ring that is explored and reexplored each time a new sink is discovered, which is particularly problematic for high-fanout nets.

If we consider the new sink and the closest portion of the existing routing tree to be almost a 2-terminal net by itself, we can further reduce the amount of extraneous exploration. Swartz et al. [15] suggest splitting the bounding box of multi-terminal nets into gridlike bins. As shown in Figure 20.5, after a sink is found, a new search is launched for the next furthest sink, but explorations are only started from the portion of the routing tree contained in the bin closest to the new target. In our example, after a route to $K1$ is found, only the portion

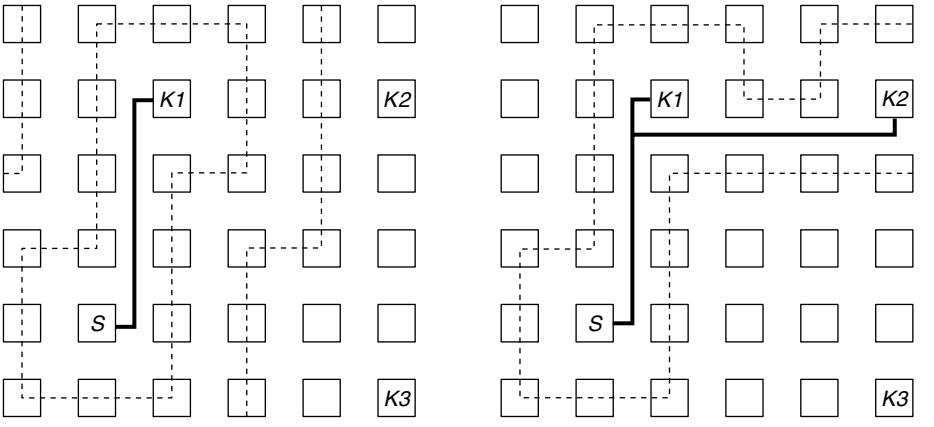

FIGURE 20.4 ■ PathFinder exploration and multi-terminal nets.

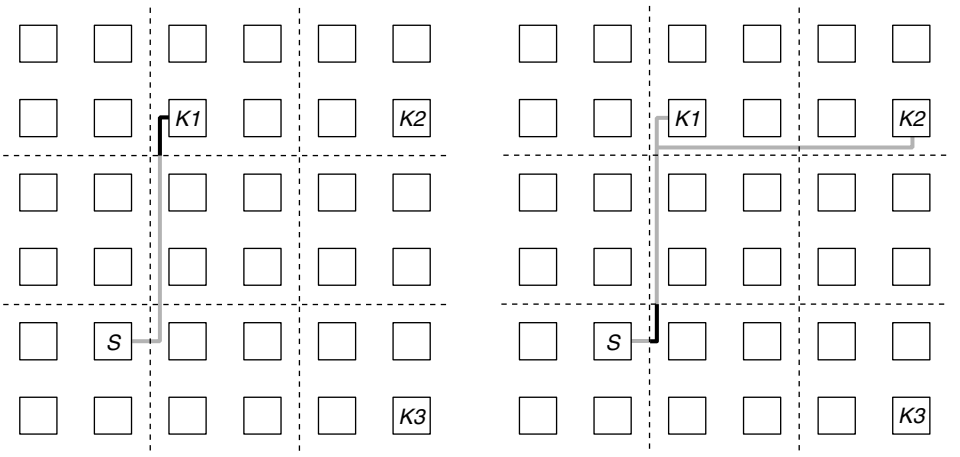

FIGURE 20.5 ■ Multi-terminal nets and region segmentation.

of the existing path in the topmost bin is used to launch a search for $K2$. The process of restricting the initialization of the search is repeated to find a route to $K3$. This may result in slightly longer branches, but, again, it is not an issue in low-stress routing situations.

Although potentially very effective, all of these techniques only attempt to improve the time required to route a single net. As described in Chapter 17, however, the PathFinder algorithm is relatively amenable to parallel processing. Chan et al. [7] showed that we can simply split the nets of a given circuit among multiple processors and allow each to route its nets mostly independently of the others. Similarly to what happens in parallel simulated annealing, complete faithfulness to the original PathFinder algorithm requires a large amount of communication bandwidth. This is because we have no guarantees that one processor will not attempt to route a signal on the same wire as another processor during a given iteration unless they are in constant communication with each other. However, because PathFinder already has a mechanism to discourage the overuse of routing resources between different nets over multiple iterations, such continuous communication is unnecessary. We can allow multiple processors to operate independently of one another for an entire routing iteration.

When all processors have routed all of their nets, we can simply determine which nodes were accidentally shared by different processors and increase their present sharing and history costs appropriately. Just as it discourages sharing between nets in classical single-processor PathFinder, this gradually discourages sharing between different processors over multiple iterations. We are using the built-in conflict-resolution mechanism in a slightly different way, but this allows us to reduce the communication overhead considerably. That said, after we have resolved the large-scale congestion in the system, the last few routing iterations likely must be performed on a single processor using conventional PathFinder.

Overall, these techniques are extremely effective on modern FPGAs. Most of today's reconfigurable architectures include a wealth of routing resources that are sufficient for a wide range of applications. Because of this, all of these approaches to accelerating PathFinder-style routing produce good results. Ordering of nets, fast growth of present sharing and history costs, and limiting the scope of exploration to net bounding boxes are common in modern FPGA routing tools. Unfortunately, however, they are still not fast enough for the most demanding applications such as runtime netlist compilation. Even the parallel technique outlined here has an unavoidable serial component. Thus, while such techniques may be adequate to produce results for next-generation FPGAs or hardware prototyping systems, they must be much faster if we are to make runtime netlist compilation practical.

20.2 ALTERNATIVE ALGORITHMS

Although classical mapping techniques have proven that they can achieve high-quality results, there is a limit to their acceleration through conventional means if we want to maintain acceptable quality for many applications. For example, in the case of placement the number of moves attempted in the inner loop of simulated annealing can only be reduced to a certain point before solution quality is no longer acceptable. While the runtime on a single processor can be cut by a factor of 10 with relatively little change in terms of routability or critical-path timing, even such modest degradation may not meet the most demanding design constraints. Furthermore, as discussed earlier, attempting to scale this technique beyond the 10x point generally results in markedly lower quality because the algorithm simply does not have sufficient time to adequately explore the solution space. To achieve further runtime improvements without resorting to potentially complex parallel implementations and without abandoning solution quality, we must make fundamental algorithmic changes.

20.2.1 Multiphase Solutions

One of the most popular ways to accelerate placement is to break the process into multiple phases, each handled by a different algorithm. Although many techniques use this method, a common thread among them all is that large-scale optimization is performed first by a fast but relatively imprecise algorithm. Slower, more accurate algorithms are reserved for local, small-scale refinement as a secondary step. A good example of this approach is shown in papers such as that by Xu and Kalid [20]. Here, the authors use a quadratic technique to obtain a rough placement and then work toward a better solution with a short simulated annealing phase.

In quadratic placement, the connections between blocks in the netlist are converted into linear equations, any valid solution to which indicates the position of each block. A good placement solution is found by solving the matrix equations while attempting to minimize another function: the sum of the squared

wirelength for each net. Unfortunately, one of the problems with this approach is that, in order for the equations to be solved quickly, they must be unconstrained. Thus, the placements found directly from the quadratic solver will likely have many blocks that overlap.

Xu and Kalid [20] identify these overlapping cells and, over multiple iterations, slowly add equations that force them to move apart. This is a comparatively fast process, but the additional placement legalization factors are added somewhat arbitrarily. Thus, although the quadratic placement might have gotten all of the blocks in roughly the correct area, there is still quite a bit of room for wirelength and timing improvements.

In contrast, while simulated annealing produces very good results, much of the runtime is devoted to simply making sense of a random initial placement. By combining the two approaches, and starting a low-temperature annealing only after we obtain a reasonable initial placement from the quadratic solver phase, we can drastically reduce runtime and still maintain the majority of the solution quality. Similar approaches can substitute force-directed placement for large-scale optimization or completely greedy optimization for small-scale improvement [12].

Another way to quickly obtain relatively high-quality initial placements is with partitioning-based approaches. As mentioned in Chapter 14, although recursive bipartitioning can be performed very quickly, reducing the number of signals cut by the partitions is not necessarily the same thing as minimizing wirelength or critical path delay. A similar but more sophisticated method is also discussed in Chapter 14. In hierarchical placement, as described by Sankar and Rose [13], the logical resources of a reconfigurable architecture are roughly divided into K separate regions. Multiple clustering steps then assign the netlist blocks into groups of approximately the correct size for the K logical areas. At this point, the clusters themselves can be moved around via annealing, assuming that all of the blocks in a cluster are at the center of the region.

This annealing can be performed very quickly since the number of clusters is relatively small compared to the number of logic blocks in the netlist. We can obtain a relatively good logic block-level placement by taking the cluster-level placement and decomposing it. Here, we can take each cluster in turn and arbitrarily place every block somewhere within the region assigned to it earlier. This initial placement can then be refined with a low-temperature annealing.

Purely mechanical clustering techniques are not the only way to group related logic together and obtain rough placements very quickly. In fact, the initial design specification itself holds valuable information concerning how the circuit is constructed and how it might best be laid out. Unfortunately, this knowledge is typically lost in the conventional tool flow. Regardless of whether they are using a high-level or low-level hardware description language, the organizational methods of humans naturally form top-level designs by connecting multiple large modules together. These large modules are, in turn, also created from lower-level modules. However, information about the overall design organization is generally not passed down through logical synthesis and technology mapping tools.

Packing, placement, and routing are typically performed on a completely flattened netlist of basic logic blocks. However, as suggested in works by Gehring and Ludwig and colleagues [10] and Callahan et al. [6], for example, for most applications this innate hierarchy can suggest which pieces are heavily interconnected and should be kept close together during the mapping process. Furthermore, information about multiple instances of the same module can be used to speed the physical design process.

The datapath-oriented methodology described in Chapter 15 uses a closely related concept to help design highly structured computations. In datapath composition, the entire CAD toolflow, from initial algorithm specification to floorplanning to placement, is centered on building coarse-grained objects that have obvious, simple relationships to one another. The entire computation is built from regular, snap-together tiles that can be arranged in essentially the same order in which they appear in the input dataflow graph. Although many applications simply do not fit the restrictive nature of the datapath computation model, applications that can be implemented in this way benefit greatly from the highly regular structures these tools create.

There may not be as much regularity in most applications, but we can still use organizational information to accelerate both placement and routing. At the very least, such information provides some top-level hints to reasonable clustering boundaries and can be used to roughly floorplan large designs. In some sense, this is exactly the aim of hierarchical placement, although it attempts to accomplish this without any a priori knowledge. Extending this idea, for very large systems we can use these natural boundaries to create multiple, more or less independent top-level placement problems. Even if we place each of the large system-level modules serially on a single processor, it is likely that, because of nonlinear growth in problem complexity, the total runtime will still be smaller than if we had performed one large, unified placement.

We can also employ implicit organizational information on a smaller scale in a bottom-up fashion. For example, many modern FPGAs contain dedicated fast carry-chain logic between neighboring cells. To use these structures, however, the cells must be placed in consecutive vertical logic block locations. If we were to begin with a random initial placement for a multibit adder, we would probably not find the optimal single-column placement despite the fact that, based on higher-level information, the best organization is obvious. Such very common operations can be identified and then preplaced and routed with known good solutions. These blocks then become hard macros. Less common or larger calculations can be identified and turned into soft macros. As suggested by projects such as Tessier's [17], using the high-level knowledge of macros within a hierarchical-style placement tool can improve runtime by a factor of up to 50 without affecting solution quality.

Still, while macro identification can significantly improve placement runtime, its effect on routing runtime is likely negligible. Soft macros still need to be routed because each instance may be of a different shape. Furthermore, although hard macros do not need to be repeatedly routed, and may be relatively common, their nets represent a small portion of the overall runtime because

they are typically short and are simple to route. Rather, to substantially improve routing runtime we need to address the nets that consume the largest portion of the computational effort—high-fanout nets. As discussed earlier, multi-terminal nets present a host of problems for routers such as PathFinder. In many circuits, the routing time for one or two extremely high-fanout nets can be a significant portion of the overall routing runtime. However, this effort might be unnecessary since, even though these nets are ripped up and rerouted in every iteration, they go nearly everywhere within their bounding box. This means that virtually all legal routing scenarios will create a relatively even distribution of traffic within this region and none are markedly better than any other. For this reason, we can easily route these high-fanout nets once at the beginning of the routing phase and then exclude them from following a conventional PathFinder run without seriously affecting overall routability. At the very least, if we do not want to put these nets completely outside the control of PathFinder congestion resolution, we can rip up and reroute them less frequently, perhaps every other or every third iteration.

Regardless of how the placement and routing problem is divided into simpler subproblems, multiphase approaches are the most promising way to deal with the issues associated with FPGA technology scaling. Of course, when possible it is best to gather implicit hierarchical information directly from the source hardware description language specification. This not only allows us to create both hard and soft macros very easily, but gives strong hints regarding how large designs might be floorplanned. That said, we may not have information regarding high-level module organization. In these cases we can fall back on hierarchical or partitioning placement techniques to make subsequent annealing problems much more manageable. All of these placement methodologies scale very well, and they represent algorithms that can solve the most pressing issues presented by growing reconfigurable devices and netlists.

When applicable, constructive techniques, such as the datapath-oriented methodology described in Chapter 15, or macro-based approaches can be very useful for mapping hardware prototyping systems and instance-specific circuits. These methodologies naturally produce reasonable placements very quickly. Because hardware emulation systems and instance-specific circuits do not necessarily need optimal area or timing results, these techniques often produce placements that can be used directly without the need for subsequent refinement steps.

20.2.2 Incremental Place and Route

Incremental placement and routing techniques attempt to reduce compilation time by combining and extending the same ideas exploited by multiphase compilation approaches: (1) begin with a known reasonable placement and (2) avoid ripping up and rerouting as many nets as possible.

In many situations, multiple similar versions of a given circuit might be placed and routed several times. In the case of hardware emulation, for example, it is unlikely that large portions of the circuit will change between consecutive

designs. Far more likely is that small bug fixes or local modifications will be made to specific portions of the circuit, leaving the vast majority of the design completely unchanged. Incremental placement and routing methodologies identify those portions of a circuit that have not changed from a previous mapping and attempt to integrate the changed portions in the least disruptive manner. This allows successive design updates to be compiled very quickly and minimizes the likelihood of dramatic changes to the characteristics of the resultant mapping.

The key to incremental mapping techniques is to modify an existing placement as little as possible while still finding good locations for newly introduced parts. The largest hurdle to this is merely finding a legal placement for all new blocks. If the changes reduce the overall size of the resulting circuit, any new logic blocks can simply fit into the void left by the old section. However, if the overall design becomes larger, the mapping process is more complex. Although the extra blocks can simply be dropped into any available location on the chip, this will probably result in poor timing and routability. Thus, incremental mapping techniques generally use simple algorithms to slightly move blocks and make vacant locations migrate toward the modified sections of the circuit.

The most basic approaches, such as those described by Choy et al. [4], determine where the closest empty logic block locations are and then simply slide intervening blocks toward these vacancies to create space where it is needed. Singh and Brown [14] use a slightly more sophisticated approach that employs a stochastic hill-climbing methodology, similar to a restricted simulated annealing run. This algorithm takes into account where additional resources are needed, the estimated critical path of the circuit, and the estimated required wirelength. In this way, logic blocks along noncritical paths will preferentially be moved to make room for the added logic.

Incremental techniques not only speed up the placement process, but can accelerate routing as well. Because so much of the placement is not disturbed, the nets associated with those logic blocks do not necessarily have to be rerouted. Initially, the algorithm can attempt to route only the nets associated with new or moved logic blocks. If this fails, or produces unacceptable timing results, the algorithm can slowly rip up nets that travel through congested or heavily used areas and try again. Either way, it will likely need to reroute only a very small portion of the overall circuit.

Unfortunately, there are many situations in which we do not have the prior information necessary to use incremental mapping techniques. For example, the very first compilation of a netlist must be performed from scratch. Furthermore, it is a good idea to periodically perform a complete placement and routing run, because applying multiple local piecework changes, one on top of another, can eventually lead to disappointing global results. However, as mentioned earlier, incremental compilation is ideal for hardware prototyping systems because they are typically updated very frequently with minor changes. This behavior also occurs in many other development scenarios, which is why incremental compilation is a common technique to accelerate the engineering/debugging design loop.

However, there are some situations in which it is very difficult to apply incremental approaches. For example, these techniques rely on the ability to determine what portions of a circuit do or do not change between design revisions. Not only can merely finding these similarities be a difficult problem, we must also be able to carefully control how high-level synthesis, technology mapping, and logic block packing are performed. These portions of the mapping process must be aware when incremental placement and routing is going to be attempted, and when major changes have been made to the netlist and placement and routing should be attempted from scratch.

20.3 EFFECT OF ARCHITECTURE

Although we have considered many algorithmic changes that can improve compilation runtime, we should also consider the underlying reasons that the FPGA mapping problem is so difficult. Compared to standard cell designs, FPGAs are much more restrictive because the logic and routing are fixed. Technology mapping must target the lookup tables (LUTs) and small computational cores available on a given device, placement must deliver a legal arrangement that coincides with the array of provided logic blocks, and routing must contend with a fixed topology of communication resources.

For these reasons, the underlying architecture of a reconfigurable device strongly affects the complexity of design compilation. For example, routing on a device that had an infinite number of extremely fast and flexible wires in the communication network would be easy. Every signal could simply take its shortest preferred path, and routing could be performed in a single Dijkstra's pass. Furthermore, placement would also be obvious on such an architecture since even a completely arbitrary arrangement could meet design constraints. Granted, real-world physical limitations prevent us from developing such a perfect device, but we can reduce the necessary CAD effort with smart architectural design that emphasizes ease of compilation—potentially even over logic capacity and clock speed.

The Plasma architecture [2] is a good example of designing an FPGA explicitly for simple mapping. Plasma was developed as part of the Teramac project [1]—an extremely large reconfigurable computing system slated to contain hundreds or thousands of individual FPGAs. Even given that a large design would be separated into smaller pieces that could be mapped onto individual FPGAs, contemporary commercial reconfigurable devices required tens of minutes to complete placement and routing for each chip. To further compound this issue, even after placement was completed once, there was no guarantee that all of the signals could be successfully routed, so the entire process might have to be repeated. This meant that a design that utilized thousands of conventional FPGAs could require days or weeks of overall compilation time. For the Teramac system to be useful in applications such as hardware prototyping, in which design changes might be made on a daily or even hourly basis, mapping had to be orders of

magnitude faster. Thus, the Plasma FPGA architecture was designed explicitly with fast mapping in mind.

Although Plasma differed from contemporary commercial FPGAs in several key ways, its most important distinction was high connectivity. Plasma was built from 6-input, 2-output logic blocks connected hierarchically by two levels of crossbars. As seen in Figure 20.6, logic blocks are separated into groups of 16 that are connected by a full crossbar that spans half the width of the chip. These groups are then connected to other groups by a central partial crossbar. The central vertical lines span a quarter of the height of the array, but have the capability to be connected together to span the entire distance. Since full crossbars would

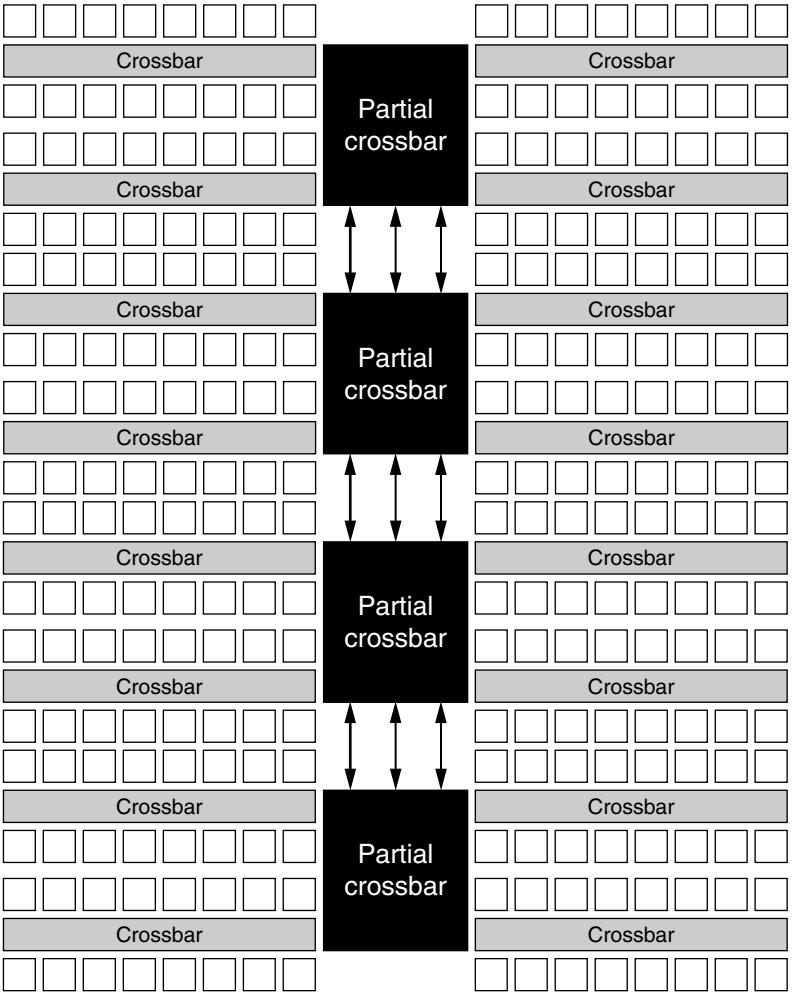

FIGURE 20.6 ■ The Plasma interconnect network.

have been prohibitively large, the developers used empirical testing to determine what level of connectivity was typically used in representational benchmarks. In addition to high internal connectivity, Plasma also contained an unusually large number of off-chip I/O pins.

Although this extremely dense routing fabric consumed 90 percent of the overall area, and its reliance on very long wires reduced the maximum operating frequency considerably, placement and routing could reliably be performed on the order of seconds on existing workstations. Given Teramac's target applications, the dramatic increase in compilation speed and the extremely consistent place and route success rate was considered to be more important than logical density or execution clock frequency.

Of course, not all applications can make such an extreme trade-off between ease of compilation and general usability metrics. However, manipulating the architecture of an FPGA does not necessarily require dramatically altering the characteristics of the device. For example, it is possible to make small changes to the interconnect to make routing simpler. One possibility is using a track domain architecture, which restricts the structure of the switch boxes in an island-style FPGA.

As shown in Figure 20.7, the connectivity of an architecture's switch boxes can affect routability. While each wire in both the top and bottom switch boxes have the same number of fanouts, the top switch box allows tracks to switch wire domains, eventually migrating to any track through multiple switch points.

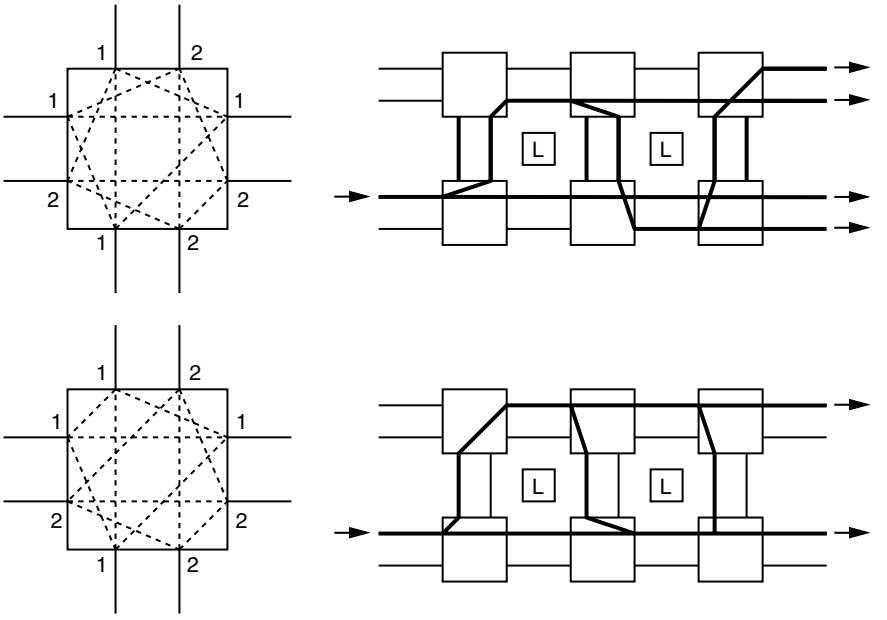

FIGURE 20.7 ■ Switch box style and routability.

This allows a signal coming in on one wire on the left of the top architecture to reach all four wires exiting the right. However, the symmetric switch box shown on the bottom does not allow tracks to switch wire domains and forces a signal to travel along a single class of wire. This means that a signal coming in from the left of the bottom architecture can only reach two of the four wires exiting to the right. Although this may reduce the flexibility of the routing fabric somewhat [18], potentially requiring more wires to achieve the same level of routability [8], this effect is relatively minor.

Even though we may need to increase the channel width of our architecture because of the restrictive nature of track domain switch boxes, routing on this type of FPGA can be dramatically faster than on more flexible systems. As shown by Cabral et al. [5], since the routing resources on track domain FPGAs are split into *M* different classes of wire, routing becomes a parallel problem. First, *N* processors are each assigned a small number of track domains from a given architecture. Then the nets from a circuit placed onto the architecture are simply split into *N* groups. Because each track domain is isolated from every other due to the nature of the architecture, each processor can perform normal PathFinder routing without fear that the paths found by one processor will interfere with the paths found by another. When a processor cannot route a signal on its allotted routing resources, it is given an additional unassigned track domain. Although load balancing between processors and track domains is somewhat of a problem, this technique has shown linear or even super-linear speedup with a very small penalty to routability. In this case, Cabral and colleagues [5] were able to solve the problems encountered by the parallel routing approaches that were discussed earlier by modifying the architecture itself.

Another way to modify the physical FPGA to speed routing is by offering specialized hardware to allow the device to route its own circuits. Although similar to the approach discussed earlier in which simulated annealing is implemented on a generic FPGA to accelerate the placement of its own circuits, DeHon et al. [9] suggest that by modifying the actual switch points internal to an FPGA, we can create a specialized FPGA that can assist a host processor to perform PathFinder-like routing by performing its own Dijkstra searches. In this type of architecture, the switch points have additional hardware that gives them the ability to remember the inputs and outputs currently being used when the FPGA is put into a special compilation time-only "routing search" mode.

After the placement of a given circuit is found, we configure the FPGA to perform routing on itself. This begins by clearing the occupancy markers on all of the switch points. During the routing phase, the host processor requests that each net in turn drive a signal from its source, which helps discover a path to each of its sinks. Every time this signal encounters a switching element, the switch allows the signal to propagate though unallocated resources but prevents it from continuing along occupied segments. In this way, the device explores all possible paths virtually instantaneously. When a route is found between the source and a sink, the switch point occupancy markers along this path are updated to reflect the "taken" status of these resources. When a route cannot be found for a given net, because all of the legal paths have been occupied

by earlier nets, the system simply victimizes a random previously routed path and rips it up until the blocked net can successfully route. Nets are continuously routed and ripped up in this round-robin fashion until all nets have been routed. Although this approach does not have the same sophistication as PathFinder, the experiments by DeHon and colleagues [9] show that hardware-assisted routing can obtain extremely similar track counts (only 1 to 2 additional tracks) with 4 to 6 orders of magnitude speedup in terms of runtime on the largest benchmarks.

Of course, modifying an FPGA architecture can involve a great deal of engineering effort. For example, while hardware-assisted routing is one of the only approaches that is fast enough to make runtime netlist compilation feasible, it involves completely redesigning the communication network. That said, not all of our architecture modifications need to be that drastic. For example, commercial FPGA manufacturers have already made modifications to their architectures that accelerate routing. As mentioned earlier, commercial FPGAs offer a resource-rich, flexible routing fabric to support a wide range of applications. Their high bandwidth and connectivity naturally make the routing problem simpler and much faster to solve. Following this logic, it seems natural that FPGAs might switch to track domain architectures in the future. While such devices require only minor layout changes that slightly affect overall system routability, they enable very simple parallel routing algorithms to be used. This becomes more and more important as reconfigurable devices scale and as multi-threaded and multicore processors gain popularity.

20.4 SUMMARY

In this chapter we explored many techniques to accelerate FPGA placement and routing. Ultimately, all of them have restrictions, benefits, and drawbacks. This means that our applications, architectures, and design constraints must dictate which methodologies can and should be used. Several of the approaches do not provide acceptable runtime given problem constraints, while some may not offer sufficient implementation quality. Some techniques may not scale adequately to address our issues, while we may not have the necessary information to use others.

FPGA scaling. Although classical block-level simulated annealing techniques have been the cornerstone of FPGA CAD tools for decades, these methodologies must eventually be replaced. Hierarchical and macro-based techniques seem to scale much more gracefully while preserving the large-scale characteristics of high-quality simulated annealing. On the other hand, routing will likely depend on PathFinder and other negotiated congestion techniques for quite some time. That said, for compilation time to keep pace given newer and larger devices, FPGA developers need to make some architectural changes that simplify the routing problem. Track domain

systems seem to be a natural solution given that modern desktops and workstations offer multiple types of parallel processing resources.

Hardware prototyping and logic emulation systems. While these systems benefit greatly from incremental mapping techniques, they still require fast place and route algorithms when compilation needs to be performed from scratch. Hardware-assisted placement seems an obvious choice that can take full advantage of the multichip arrays present in these large devices. Furthermore, since optimal critical-path timing is not essential and application source code is generally available to provide hierarchical information, datapath and macro-based approaches can be very effective.

Instance-specific designs. Datapath and macro-based approaches are even more important to instance-specific circuits because they cannot take advantage of many other techniques. However, the limited scope of these problems and the dramatic speedup made possible by these systems also make specialized architectures attractive. While the overhead imposed by architectures such as Plasma may not be practical for most commercial devices, these drawbacks are far less important to instance-specific circuits given the significant CAD tool benefits.

Runtime netlist compilation. Reconfigurable computing systems that require runtime netlist compilation present an incredibly demanding real-time compilation problem. Correspondingly, these systems require the most aggressive architectural approaches to make this possible. Radical system-wide modifications that provide huge amounts of routing resources significantly simplify the placement problem. However, just providing more bandwidth does not necessarily accelerate the routing process. These systems need to provide communication channels that either do not need to be negotiated or, through hardware-assisted routing, can automatically negotiate their own connections. An open question is whether the advantages of runtime netlist compilation are worth the attendant costs and complexities they introduce.

References

[1] R. Amerson, R. Carter, W. Culbertson, P. Kuekes, G. Snider. Teramac—configurable custom computing. *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, 1995.

[2] R. Amerson, R. Carter, W. Culbertson, P. Kuekes, G. Snider, L. Albertson. Plasma: An FPGA for million gate systems. *Proceedings of ACM Symposium on Field-Programmable Gate Arrays*, 1996.

[3] V. Betz, J. Rose, A. Marquardt. *Architecture and CAD for Deep-Submicron FPGAs*, Kluwer Academic, 1999.

[4] C. Choy, T. Cheung, K. Wong. Incremental layout placement modification algorithm. *IEEE Transactions on Computer-Aided Design* 15(4), April 1996.

[5] L. Cabral, J. Aude, N. Maculan. TDR: A distributed-memory parallel routing algorithm for FPGAs. *Proceedings of International Conference on Field-Programmable Logic and Applications*, 2002.

[6] T. Callahan, P. Chong, A. Dehon, J. Wawrynek. Fast module mapping and placement for datapaths in FPGAs. *Proceedings of ACM Symposium on Field-Programmable Gate Arrays*, 1998.
[7] P. Chan, M.D.F. Schlag, C. Ebeling. Distributed-memory parallel routing for field-programmable gate arrays. *IEEE Transactions on Computer-Aided Design* 19(8), August 2000.
[8] Y. Chang, D. F. Wong, C. K. Wong. Universal switch modules for FPGA design. *ACM Transactions on Design Automation of Electronic Systems* 1(1), January 1996.
[9] A. DeHon, R. Huang, J. Wawrzynek. Hardware-assisted fast routing. *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, 2002.
[10] S. Gehring, S. Ludwig. Fast integrated tools for circuit design with FPGAs. *Proceedings of ACM Symposium on Field-Programmable Gate Arrays*, 1998.
[11] M. Haldar, M. A. Nayak, A. Choudhary, P. Banerjee. Parallel algorithms for FPGA placement. *Proceedings of the Great Lakes Symposium on VLSI*, 2000.
[12] C. Mulpuri, S. Hauck. Runtime and quality trade-offs in FPGA placement and routing. *Proceedings of ACM Symposium on Field-Programmable Gate Arrays*, 2001.
[13] Y. Sankar, J. Rose. Trading quality for compile time: Ultra-fast placement for FPGAs. *Proceedings of ACM Symposium on Field-Programmable Gate Arrays*, 1999.
[14] D. Singh, S. Brown. Incremental placement for layout-driven optimizations on FPGAs. *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, 2002.
[15] J. Swartz, V. Betz, J. Rose. A fast routability-driven router for FPGAs. *Proceedings of the ACM Symposium on Field-Programmable Gate Arrays*, 1998.
[16] R. Tessier. Negotiated A* routing for FPGAs. *Proceedings of the Canadian Workshop on Field-Programmable Devices*, 1998.
[17] R. Tessier. Fast placement approaches for FPGAs. *Transactions on Design Automation of Electronic Systems* 7(2), April 2002.
[18] S. Wilton. *Architecture and Algorithms for Field-Programmable Gate Arrays with Embedded Memory*, Ph.D. thesis, University of Toronto, 1997.
[19] M. Wrighton, A. DeHon. Hardware-assisted simulated annealing with application for fast FPGA placement. *Proceedings of the ACM Symposium on Field-Programmable Gate Arrays*, 2003.
[20] Y. Xu, M.A.S. Kalid. QPF: Efficient quadratic placement for FPGAs. *Proceedings of the International Conference on Field-Programmable Logic and Applications*, 2005.