Simultaneous Retiming and Placement for Pipelined Netlists

Ken Eguro and Scott Hauck
Department of Electrical Engineering
University of Washington
Seattle, WA 98195 USA
{eguro, hauck}@ee.washington.edu

Abstract

Although pipelining or C-slowing an FPGA-based application can potentially dramatically improve the performance, this poses a question for conventional reconfigurable architectures and CAD tools: what is the best way to support these new extra registers? While there have been multiple research efforts to address this problem, they generally impose strict architectural requirements, offer limited retiming capabilities, or require multiple iterations with no guarantees regarding feasible implementations.

In this paper we introduce a new simulated annealing-based placement and retiming approach that provides the capability to aggressive apply retiming on a wide range of netlists, for arbitrary architectures, while maintaining predictable results. Our results show that for heavily pipelined applications, this methodology can produce netlists and placements with 1.65x better post-routing critical path delay as compared to the classical approach of retiming before timing-driven VPR placement, and 1.08x better than retiming before our improved timing-driven placer.

1. Introduction

Despite the inherent speed penalties generally associated with reconfigurable devices, the flexibility and low engineering effort of FPGA-based platforms have made them very popular for a wide variety of different applications. To combat the naturally lower clock frequency of FPGA implementations, application developers often pipeline or C-slow their circuits whenever possible. Retiming is a powerful optimization technique that can be applied to such registered netlists to help maximize the achievable clock frequency.

One problem is that retiming is generally performed prior to packing, placement and routing. Unfortunately, since interconnect delay often plays a major role in determining overall performance, accurate timing information can only be obtained after placement. Retiming is generally restricted to early parts of the CAD flow because, while retiming does not alter the logical behavior of the circuit, it can cause extensive changes to the structure of the netlist itself. Retiming alters both the number and relative positioning of the registers between logical elements.

For most reconfigurable architectures, this means that the placement must also be changed to accommodate the new registers. Regrettably, this can lead to problems with timing stability and convergence since the new placement may not have the same timing characteristics as the original placement. This means that we cannot guarantee the accuracy of the retiming, potentially slowing the circuit down instead of speeding it up.

Multiple research groups have identified this issue and have suggested changes that can be made either to the underlying FPGA architecture or to the CAD tools. That said, the architectural solutions that have been proposed either have dramatic area overhead and latency restrictions that limit their practicality for non-pipelined netlists, or employ relatively exotic and piecemeal techniques that do not reflect the current trends seen in commercial devices. On the other hand, the CAD approaches that have been suggested either take a very conservative tact, limiting the potential benefits of heavy pipelining, or risk changing the netlist considerably, potentially compromising the overall quality of the final implementation.

In this paper we discuss the specific issues surrounding the aggressive application of retiming late in the CAD flow, and introduce a simulated-annealing based retiming methodology that integrates retiming and placement into a single unified tool.

2. Implications for Practical Register-Enhanced FPGAs

Many applications, particularly those of interest to reconfigurable computing, can tolerate a great deal of pipelining or C-slowing. While this can improve performance, it also produces netlists that are fundamentally different than conventional circuits. Some researchers have sought to meet these challenges with architectures specifically targeted to only heavily pipelined designs [12][14]. However, these devices have not become popular because, while they can provide an impressive performance boost, they also impose a large 2x-4x area penalty and require enormous latency compared to classical systems. This makes these devices unsuitable to mainstream users, and thus we have not seen any commercial success on this front.

An alternative, demonstrated by Brian Von Herzen [13], is to see how commodity FPGAs can be used for heavily pipelined applications. Essentially, since we would like to completely pipeline all of the connections in the system, registers end up dominating the logic portion of the circuit. Thus, we end up using the vast majority of the CLBs in the target device only for their registers. While this is a better approach since we are able to use a commodity device, in a sense we also pay a 2x-4x area penalty when considering the logic resources of the system.

We are working on a middle ground. What if we could augment a commodity device with a relatively small number of additional registers? These resources would only degrade the logic capacity of commodity devices by a few percent for standard users, but will allow these FPGAs to achieve very high clock rates for latency-tolerant designs. In this way we can combine the two design philosophies we have discussed and produce a commodity-style architecture that leverages main-stream users for economies of scale, yet allows them to boast very high performance for heavily pipelined applications.

This paper is one step in the pursuit of this ideal. In previous work we have developed an architecture-adaptive router for pipelined interconnects [9], as well as a timing-driven version of this tool [5]. In this paper we introduce an architecture-adaptive placer for pipelined FPGAs, leveraging our previous work on a much more efficient timing-driven placer for pipelined netlists [4]. When combined, this provides a complete pipeline-aware placement and routing tool suite for register-enhanced commodity-style FPGAs. Ultimately, we plan to use this toolflow in architectural studies to help design commercially viable register-rich architectures that can support both lightly and heavily pipelined netlists efficiently.

3. Background

Pipelining or C-slowing an application introduces additional registers into the netlist with the hope that this will increase the overall throughput of the system. However, since these new registers also increase the latency of the circuit, we must be careful to use these registers effectively in order to evenly distribute delay. The hope is that we can use these registers to achieve a sufficient improvement in critical path delay to offset the additional latency. This is where retiming plays a crucial role.

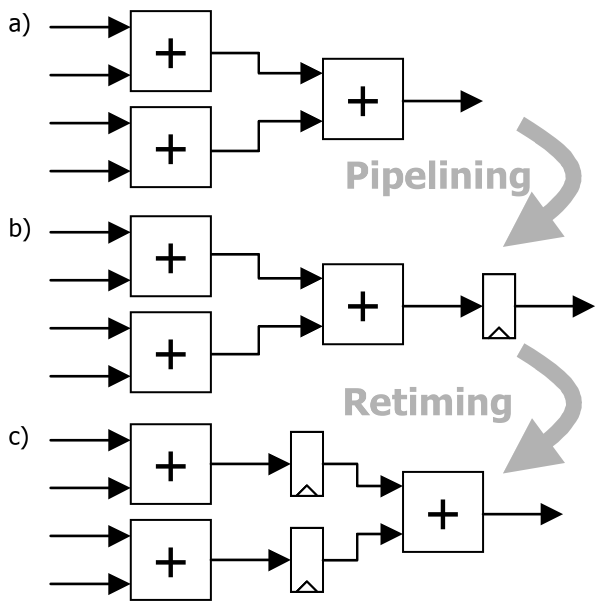

Figure 1 – Pipelining and Retiming

3.1 Classical Retiming

Retiming improves a circuit's critical path by better balancing the delay between different pipelining stages. It achieves this, without changing the external behavior of the circuit, by "pushing" registers through computational blocks. For example, consider the design in Figure 1. If we disregard interconnect delay for the moment, we see that the original netlist (Figure 1a) has a critical path delay of two adders, and a latency of one clock cycle. However, when we pipeline the system (Figure 1b), we increase the latency by a clock cycle without improving the critical path delay. We only see the benefits of pipelining when we apply retiming (Figure 1c). Specifically, retiming works because we can generally replace a register on a logic block output with a register on each of the inputs without changing the observable behavior to the outside world. By the same token we are also able to replace a register on each of the inputs of a logic block with a register on the output.

First described in [7], the most popular method of retiming is the Leiserson/Saxe approach. This is an iterative process that gradually pushes registers around a netlist until the provably minimum critical path delay is found. Although the optimality of this approach is nice, this accuracy does not necessarily carry over when we actually apply retiming within an FPGA CAD flow. This is because retiming is generally performed as an isolated step prior to packing, placement and routing.

The conventional toolflow has developers specify their application and compile it once, all the way from technology mapping though routing. If the resulting implementation does not meet timing specifications, the developer might edit their HDL to increase the level of pipelining/C-slowing, or attempt to retime the original technology-mapped netlist for another run of packing, placement and routing.

Unfortunately, this methodology can limit the advantages of retiming or create problems for timing closure. First, without any placement information, we can only very roughly estimate delay. Similar to the assumptions made for the example in Figure 1, we must retime using a simple unit delay model for logic blocks and largely ignore the potentially significant delay accumulated in the interconnect.

However, it is also unclear how useful it might be to try and forward timing information from a previous placement and routing back to the retimer for another run of the CAD tools. First, solely based on the algorithms discussed in [7], it is not necessarily obvious how Leiserson/Saxe retiming could represent interconnect delay. That said, even if we could effectively forward timing numbers from previous implementations, we cannot guarantee the accuracy or relevancy of this information to the new netlist, since the system may change considerably in the meantime. Nets that were timing critical in an earlier placement may not remain so, even if we do not change the netlist at all but simply re-run the simulated annealing. Furthermore, we certainly do not have such information for newly created registers. All of this puts an incredible burden on the retimer since additional registers may actually degrade performance if they are not distributed correctly.

3.2 Previous Architectural Solutions

One proposed solution to the problems associated with retiming and placement convergence is modifying the architecture itself to allow retiming to be performed after placement and routing, without disturbing the existing configuration. For example, the system suggested in [11] is a track-graph FPGA that replaces some of its track domains with specialized optionally-registered track domains that are specifically dedicated to retiming.

The toolflow for this system begins with conventional timing-driven placement and routing, ignoring the registers embedded in the interconnect. At this point, timing-critical links in the routed configuration are identified and singled out. If they are not already connected via a wire domain that is outfitted with optional registers, the connection is swapped to an equivalent wire domain that does. At this point, a restricted retiming algorithm is applied. Instead of performing true Leiserson/Saxe retiming, this approach limits the number of registers that can be pushed onto a specific connection to the number of optional retiming registers that already exist along the current route.

However, while this is a simple and closed-form solution, this greatly limits the optimizations available to the retimer. First, this does not allow the system to utilize registering resources that might exist in neighboring switchboxes or logic blocks. More importantly though, this approach requires a very specific and specialized registered interconnect structure.

3.3 Previous CAD Solutions

Supporting more general registering resources requires new CAD tools. Towards this goal, there have been a number of research projects that have attempted to perform retiming during placement. However, we believe that they do not necessarily adequately address the systematic issues created by retiming.

For example, [2] was among the first efforts towards integrated placement and retiming. Although this work actually involved floorplanning, it laid the groundwork for [3]. Here, the authors define a three-stage approach for retiming-aware placement. They begin with a specialized timing-driven placement. When the annealing is complete, they perform a classical retiming step to improve delay. This is followed by a short simulated annealing process to re-distribute registers that were created or deleted. Unfortunately, this work targets an ASIC development flow. Since ASICs create completely custom chips, the CAD tools are able to create or delete resources at will. Since FPGAs must use the finite resources offered by a specific architecture, there are strict limitations as to where we can and cannot create a register.

Works such as [15] and [8] have attempted to address FPGA-specific concerns. [15] suggests a two-phase approach. The authors begin with a full simulated annealing placement, but then retiming is performed without any restrictions on the number of registers that can be placed on a given link. This means that this methodology can create an unknown number of registers in potentially very sensitive areas of the array, with no good way of cleaning up the placement. On the other hand, [8] suggested a very straightforward two-phase solution in which conventional placement is followed by a constrained retiming step. Very similar to the work in [11], the retimer can only push a limited number of registers onto a specific link. In this case, it could choose to either use or not use the flip-flops present in the CLBs already allocated by the placement phase. Of course, this approach suffers some of the same limited retiming capabilities as [11].

The most encouraging work to date on integrated FPGA placement and retiming is [10]. Much like the work in [3], they use a three-phase approach that begins with a specialized timing-driven simulated annealing

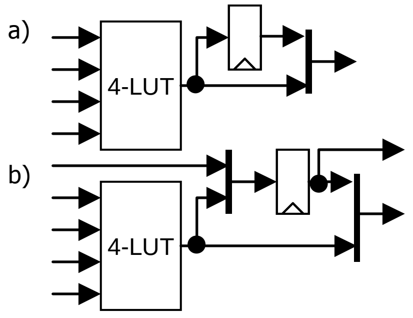

Figure 2 – Independently Accessible Flip-Flops

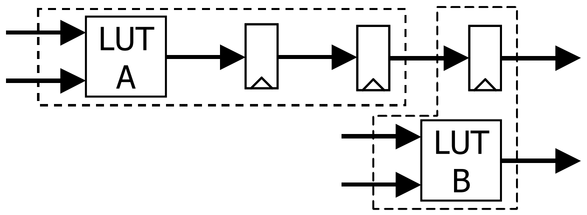

Figure 3 – Packing Implications for Heavily Registered Netlists

placement. This is followed by a heuristic retiming step that creates a new netlist given the timing and CLB occupancy of the existing placement. Primarily, this heuristic tries to insert registers into the system, keeping in mind the CLB legalization issues faced by an architecture with logic blocks that do not have full input and output connectivity, like those in Figure 2a. Any architectural violations in the placement of the retimed netlist are then resolved by a largely greedy legalization phase.

However, this approach has two issues. First, much of their work focuses on solving architecture-specific CLB input and output legalization problems that are not necessarily a concern for modern devices. Current generation FPGAs such as the Virtex II do not require cluster legalization. They not only provide independent access to LUTs and flip-flops (Figure 2b), they offer full input and output connectivity. That is, if there are eight 4-LUTs and eight flip-flops in a CLB, the logic blocks will have the capability to take 40 independent inputs and produce 16 independent outputs.

More importantly, this methodology still may not produce feasible or convergent placements. Since the retiming is wholly decoupled from the legalization phase, the retimer may produce a netlist that requires registers in an area that currently does not have any available in the existing placement. At this point, the post-processing step has to choose between producing an illegal placement or risk disrupting the timing of the system. This situation is particularly likely given heavily pipelined netlists, since, by the very nature of the netlist itself, there might be relatively few empty register locations in the array and many of the nets may be critical or nearly critical.

4. Simulated Annealing-Based Retiming

Although the multi-stage approach of retiming followed by some sort of cleanup phase is very promising, the methodologies we have looked at so far still face many of the same problems associated with the conventional toolflow. Taking a step back though, we can see that all of these complications stem from a single source – retiming cannot be performed as an isolated, single-shot optimization step if we want to provide the capability to aggressively retime while still maintaining a stable placement. Thus, we suggest a fundamentally different approach that intersperses several small retiming steps within the framework of a full annealing-based placement tool. This philosophy allows us to create a unified placement and retiming tool with smoother and more predictable behavior.

We begin with a timing-driven simulated annealing placement tool. Of course, it is clear that any optimizations we make, whether based on retiming or conventional placement, rely on good timing information. As shown in our prior work [4], the methodologies that timing-driven placement tools such as VPR generally use to track net criticality during annealing can be highly inaccurate. Primarily this is due to the computational requirements needed to perform static timing analysis and, subsequently, its infrequent application. We demonstrated that the incremental slack analysis and cost function introduced in that paper produce vastly superior placements with minimal additional effort as compared to VPR.

4.1 Flip-Flop Level Placement

One issue that we should consider is the characteristics of heavily registered circuits and the role of logic block packing. Conventional packing, such as the algorithm described in [1], generally attempts to pack flip-flops into the same CLB as their source LUT. While packing makes sense since it dramatically reduces the placement problem size, this can lead to problems when we consider deeply pipelined netlists.

Packing interferes with retiming during placement because it locks registers into specific logic blocks early in the compilation process. This can be seen in the example shown in Figure 3. If we assume we are mapping to an architecture that has two LUTs and 2 FF per CLB, the packing tool will wrap *LUT A* and its two following flip-flops into a single atomic unit before placement. This greatly limits the potential for these registers to mitigate interconnect delay. Furthermore, packing can fuse unrelated logic blocks and FFs together. In Figure 3, the third register on the output of *LUT A* cannot fit into the same CLB as its source, so it is arbitrarily combined with some other logic block before placement. This limits the placer's ability to use registers to pipeline the interconnect delay. Lastly, if we are performing retiming during annealing we must also be able to integrate newly created registers into the system efficiently. Putting these issues together, it is clear that to achieve the best performance the placer needs to have the ability to migrate critical flip-flops to different logic blocks without the strict assignments given during packing.

However, we cannot simply revert to placement at the individual LUT and flip-flop level without raising serious concerns. While this has obvious dramatic implications for the annealing runtime, we can also have problems simply finding high quality placements. For the majority of registers it makes sense for a LUT and its companion flip-flop to reside in the same CLB. Specifically, this configuration is special because the connection between the LUT and flip-flop does not incur the delay or potential wiring congestion associated with exiting and re-entering a CLB. However, if we only allow LUTs and flip-flops to move independently from one another, it makes it very easy for a LUT and flip-flop to separate, but much more difficult for them to reunite.

Consider the two possible states that a LUT and flip-flop can be in (together or apart), shown in Figure 4. If we consider the LUT and flip-flop to be initially together within a 5x5 grid of CLBs, we can see that all possible moves of either the LUT or flip-flop will break them apart. However, once in this state, only two in the 48 possible moves will bring them back together. Furthermore, once they are together it is unlikely that we will be able to move the LUT/flip-flop pair to any other location after the annealing cools past a certain critical temperature.

We solve both of these problems by adding a hybrid CLB/flip-flop-level move function to the placement tool described in [4]. As seen in Figure 5, we begin by selecting a random LUT or flip-flop in the netlist. If we select a LUT, we perform the customary CLB-level swap. However, if we select a flip-flop that is in the same CLB as its source and it is connected to a net that is at least 95% critical, we have a 10% chance to perform a flip-flop level move to separate the register (Figure 5, lines 4-9). On the other hand, if we select a flip-flop that is not in the same CLB as its source, we have a 10% chance to reunite the flip-flop with its source (Figure 5, lines 11-15). The exact criticality threshold and probabilities are not particularly sensitive, but empirically we have found that these values work well.

4.2 Simulated Annealing-Based Retiming

As we mentioned earlier, while Leiserson/Saxe retiming has some unique optimality characteristics, we believe that the key to better overall results is a more incremental approach. Thus, we propose leveraging the inherently balanced optimization aspects of simulated annealing by applying conditional retiming moves alongside standard placement moves as an integral part of the annealing process. To accomplish this we must define what a retiming move looks like. When we attempt to retime a register backwards or forwards through a logic block, there are multiple issues that we must address.

First, how do we deal with newly created registers? Moving between Figure 6a and Figure 6b, we must create two new registers on the inputs of *LUT B* to retime a register backwards. Before we attempt this move, we must first ensure that retiming this logic block is feasible. It is entirely possible that we simply do not have enough register locations available in the architecture. If this is the case, we do not attempt this retiming move. However, if it is a feasible move, we place these new registers into the closest available register location to the source of their signal. On the other hand, we might not have to create a register. Moving between Figure 6c and Figure 6d we see that one of the inputs to *LUT B* can share the input to *LUT C*.

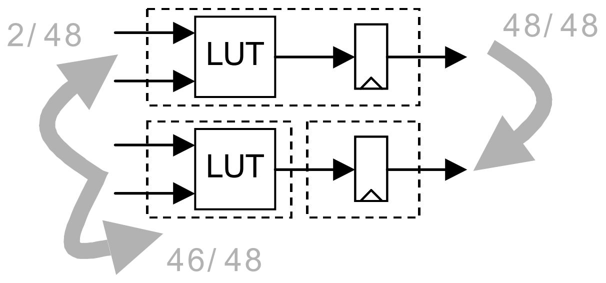

Figure 4 – Lower-Level Placement Moves

```
FF/CLB-Level Placement Move Function
0    select random LUT or FF in netlist
1    if selected LUT
2        swap entire CLB contents with random CLB
3    else
4        if FF in same CLB as source
5            if FF max link criticality >= 0.95 &&
                     rand <= 0.1 (separation roll)
6                swap FF with random FF
7            else
8                swap entire CLB contents with random CLB
9            end if
10       else
11           if rand <= 0.1 (homing roll)
12               swap FF with a FF in source CLB
13           else
14               swap FF with random FF
15           end if
16       end if
17   end if
```

Figure 5 – FF-Level Move Pseudo-Code

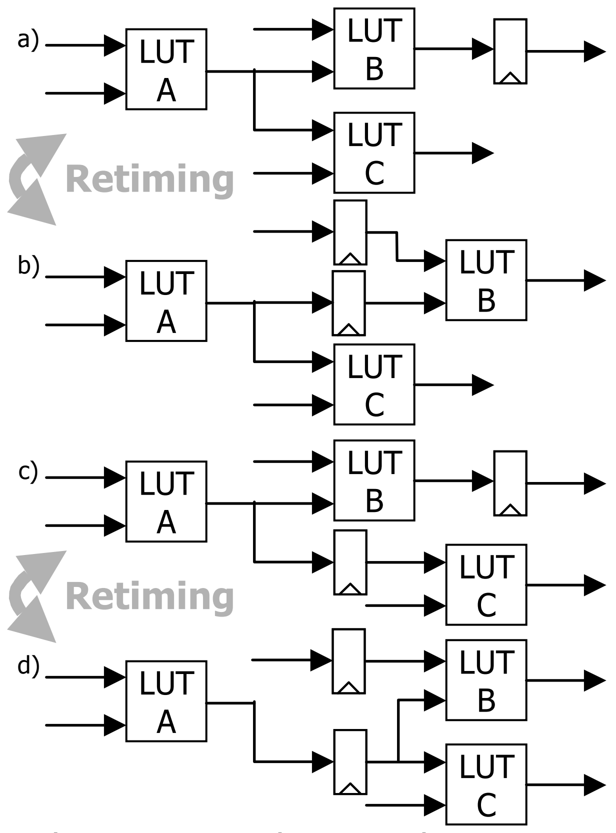

Figure 6 – Incorporating New Registers

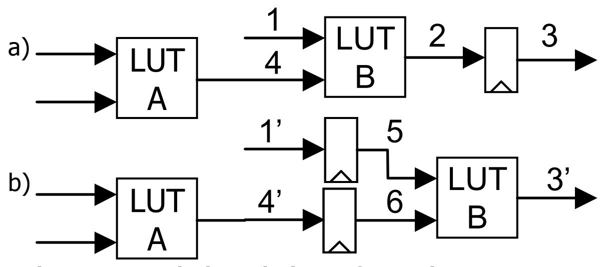

Figure 7 - Updating Timing Information

Next, when we select a given logic block, how do we know when we should attempt a retiming move versus a placement move? While we could simply flip a coin each time we select an eligible logic block, we have to consider the computational ramifications of performing a retiming move. Specifically, the large quality benefit of the placement tool we started with relies on the more accurate timing information provided by the incremental slack analysis approach [4]. Therefore, we have to consider the impact that retiming moves have on the accuracy of timing information.

Consider the retiming move performed in Figure 7. This move changes the slack (defined as the required time of the sink minus the departure time of the source) on all of the labeled nets. We can recalculate the slack on nets *1'* and *4'* with very little error because the departure time of the sources do not change and we know that the required time of a registers is always equal to the critical path of the entire system. We can also determine the new slack on nets *5* and *6* relatively accurately because the departure time for a register is always zero and we can relatively efficiently recalculate the required time of LUT *B*. Lastly, we can also relatively accurately recalculate the slack on net *3'*. This is because the required time of the sink of net *3'* does not change, and we can relatively efficiently recalculate the departure time of LUT *B*.

That said, because we may have changed the critical path of the netlist, we cannot be entirely certain of the accuracy of any of these values unless we perform full static timing analysis. Unfortunately, as discussed in [4], static timing analysis is too computationally expensive to perform after every move during simulated annealing. Since retiming moves can have a large impact on the critical path delay of the system, we group many of these move together and then perform full static timing analysis once the entire group has been completed.

As can be seen in the pseudo-code of our complete approach in Figure 8, we include a *retiming frequency factor* as a parameter to our integrated retiming and placement approach. Thus, although we continue to focus on performing conventional conditional placement moves, we also occasionally attempt to perform a concentrated suite of conditional retiming moves on the netlist.

Also seen in Figure 8, we provide a *retiming activation point* and a *retiming criticality threshold* to our tool. The retiming activation point controls when we actually begin to attempt retiming and placement, as opposed to placement only. Since the annealing begins with an arbitrary initial placement, the early portion of the placement process is primarily devoted to simply roughing out the large-scale structure of the netlist. Since the placement can change dramatically during these early stages, retiming is not very productive. In fact, the extra noise retiming creates in the netlist only serves to create problems for the placement tool. Instead, we would rather wait until the placement begins to settle down, and thus leave retiming to the later stages of the placement process.

Although there are multiple ways we could define this activation point, the placement tool we used contains VPR's adaptive temperature schedule [1]. The range limit window built into this approach is an easy way to gauge how far the overall annealing has progressed. Thus, as seen in lines 2-4 of the pseudo-code in Figure 8, the retiming activation point is simply an integer that can vary between the maximum size of the array, beginning retiming right from the start of placement, and one, beginning retiming relatively late during the annealing process. We have empirically found an activation point of 1 to work well.

```
Integrated Retiming and Placement
numMovesPerRetiming =  numAnnealMovesPerTemp /
                       retiming frequency
while (!exit)
  if range limit window <= retiming activation point
    activate retiming
  end if
  for i = 0 to numAnnealMovesPerTemp
    if retiming active && (i%numMovesPerRetiming == 0)
      for all logic blocks
        if max input criticality >= retimeCrit && can
                                        retime backwards
          try to retime once backwards
          accept or reject retiming(ΔCost, currTemp)
        end if
        if max output criticality >= retimeCrit && can
                                        retime backwards
          try to retime once forwards
          accept or reject retiming(ΔCost, currTemp)
        end if
      end for
      update critical path delay
    end if
    attempt placement move
    accept or reject move(ΔCost, currTemp)
  end for
  update critical path delay
  update currTemp
  update range limit window
  evaluate exit criteria
end while
```

Figure 8 – Simulated Annealing-Based Retiming & Placement Algorithm

As seen in Figure 8, the retiming criticality threshold filters logic blocks that are eligible for retiming based upon the maximum criticality of their input or output connections. Obviously, the more critical a given path is, the more important it becomes to retime the logic blocks along it. Again, since we would like to disrupt the placement as little as possible, we avoid retiming logic blocks that are not along highly critical paths. Given our empirical testing, a value of 1.0 (only retiming logic blocks along the critical path) produced good results.

That said, particularly for heavily pipelined netlists, our approach can leave logic blocks along highly critical paths unable to retime beyond a certain point because the necessary balancing registers along other, non-critical paths do not advance. Classical retiming approaches such as the Leiserson/Saxe techniques do not run into this problem because they perform systematic retiming. In the future, we would like to investigate a mechanism that can identify this situation and efficiently allow critical paths to single out non-critical paths for retiming.

Finally, once we have identified a good candidate for retiming we can actually perform the retiming move and restructure the netlist. Because we are performing this retiming move within a larger simulated annealing framework, we evaluate the cost of the new placement and, based upon the current temperature and change in cost, probabilistically either accept or reject this retiming move.

5. Testing & Results

We tested our integrated retiming and placement approach on two sets of MCNC benchmarks that represented a wide range of pipeline resource requirements. The first consisted of 10 sequential netlists out of the benchmarks included with the VPR tool suite. Obviously, retiming cannot be performed on purely combinational circuits that have no registers at all, so we thus exclude half of the customary benchmarks. The second group consisted of 21 original VPR benchmarks pipelined and/or C-slowed, then processed via conventional Leiserson/Saxe retiming to create circuits with a maximum logic depth of one LUT. Both sets of netlists were then packed into CLBs with T-VPack.

We placed these two groups of benchmarks with three different placement approaches: timing-driven VPR, our improved incremental slack timing-driven placement tool from [4], and our new simulated anneal-based placement and retiming approach. All three placement tools were followed by conventional timing-driven routing with A* optimizations turned off (we did not use *A** during any of our testing because A* is meant to strictly improve CAD runtime, not quality, and thus should not affect the results).

The target architecture selected was VPR's standard single 4-LUT, single flip-flop *4lut_sanitized* architecture, with one modification. Instead of logic blocks arranged as in Figure 1*a*, we provided logic blocks with independently accessible flip-flops like those shown in Figure 1*b*. We believe that this type of flip-flop accessibility is important for any register-rich FPGA architecture, and is more representative of modern devices. All testing was performed on minimum-sized architectures, with the customary low-stress routing case of 1.2 times the minimum channel width as found by VPR.

The placement parameters used for the incremental slack placement approach from [4] were the same suggested by that paper: λ=0.1, *Crit_Exp*=12.0 for the original MCNC netlists and λ=0.025, *Crit_Exp*=12.0 for the pipelined circuits. The retiming parameters we used for our integrated placement and routing tool were: *retiming activation point*=1, *retiming frequency factor*=1, *retiming criticality threshold*=1.0. To reiterate, this means that retiming began when the range limit window closed to a distance of one CLB, we performed a single retiming suite per temperature iteration, and we only considered retiming logic blocks that were 100% critical.

Table 1 – Best of 10 Wire Cost and Post-Route Crit. Path Delay Results for Original Sequential MCNC Netlists

	Raw Results						Normalized Results					
	Retime First & VPR Place		Increm. Slack Place Only		Integr. Retime & Placement		Retime & VPR Place		Increm. Slack Place Only		Integr. Retime & Placement	
Netlist	**Wire**	**CPD**	**Wire**	**CPD**	**Wire**	**CPD**	**Wire**	**CPD**	**Wire**	**CPD**	**Wire**	**CPD**
bigkey	210.57	6.21E-8	242.72	4.23E-8	242.15	4.01E-8	1.04	0.89	1.00	1.00	0.99	0.93
clma	1498.63	2.02E-7	1429.54	1.55E-7	1439.02	1.58E-7	1.08	0.94	1.00	1.00	0.99	0.94
diffeq	157.87	6.03E-8	149.75	5.57E-8	150.30	5.67E-8	1.09	1.01	1.00	1.00	1.01	0.94
dsip	194.34	6.12E-8	225.52	4.70E-8	238.64	3.61E-8	1.02	1.02	1.00	1.00	0.99	0.95
elliptic	511.90	1.03E-7	470.71	9.20E-8	474.08	9.34E-8	1.05	1.08	1.00	1.00	1.00	1.02
frisc	583.74	1.29E-7	535.74	1.27E-7	540.17	1.20E-7	0.97	1.10	1.00	1.00	0.98	0.88
s1423	16.74	5.70E-8	16.14	6.43E-8	15.93	5.95E-8	1.09	1.12	1.00	1.00	1.01	1.02
s298	228.50	1.29E-7	211.16	1.37E-7	209.49	1.29E-7	0.86	1.30	1.00	1.00	1.06	0.77
s38417	672.72	9.47E-8	690.62	8.62E-8	675.62	7.55E-8	1.05	1.31	1.00	1.00	1.01	1.02
tseng	100.75	5.29E-8	98.65	5.19E-8	97.58	4.95E-8	0.87	1.47	1.00	1.00	1.00	0.95
GeoMean							**1.01**	**1.11**	**1.00**	**1.00**	**1.00**	**0.93**

Table 2 – Best of 10 Wire Cost and Post-Route Crit. Path Delay Results for Pipelined/C-Slowed MCNC Netlists

	Raw Results						Normalized Results					
	Retime First & VPR Place		Increm. Slack Place Only		Integr. Retime & Placement		Retime & VPR Place		Increm. Slack Place Only		Integr. Retime & Placement	
Netlist	**Wire**	**CPD**	**Wire**	**CPD**	**Wire**	**CPD**	**Wire**	**CPD**	**Wire**	**CPD**	**Wire**	**CPD**
alu4	290.44	3.70E-8	254.17	3.15E-8	251.35	2.76E-8	1.14	1.17	1.00	1.00	0.99	0.88
apex2	405.33	3.99E-8	365.82	2.44E-8	367.32	2.25E-8	1.11	1.64	1.00	1.00	1.00	0.92
apex4	217.28	3.07E-8	200.46	2.44E-8	200.49	1.95E-8	1.08	1.26	1.00	1.00	1.00	0.80
bigkey	267.07	4.37E-8	246.45	2.89E-8	247.90	2.94E-8	1.08	1.51	1.00	1.00	1.01	1.02
clma	2379.44	8.46E-8	2105.72	6.45E-8	2107.38	6.33E-8	1.13	1.31	1.00	1.00	1.00	0.98
des	354.80	4.29E-8	344.98	1.99E-8	340.84	2.22E-8	1.03	2.15	1.00	1.00	0.99	1.12
diffeq	499.57	5.34E-8	460.58	2.82E-8	485.66	2.64E-8	1.08	1.89	1.00	1.00	1.05	0.94
dsip	258.44	4.02E-8	207.41	3.36E-8	218.19	3.13E-8	1.25	1.20	1.00	1.00	1.05	0.93
e64	45.27	1.93E-8	41.51	1.17E-8	42.16	1.11E-8	1.09	1.65	1.00	1.00	1.02	0.95
elliptic	1408.34	7.67E-8	1326.81	4.54E-8	1329.50	4.36E-8	1.06	1.69	1.00	1.00	1.00	0.96
ex1010	868.76	5.38E-8	796.30	3.64E-8	791.35	3.96E-8	1.09	1.48	1.00	1.00	0.99	1.09
ex5p	223.66	2.42E-8	212.14	1.64E-8	211.88	1.68E-8	1.05	1.47	1.00	1.00	1.00	1.02
frisc	1402.54	6.71E-8	1383.91	2.76E-8	1398.71	2.66E-8	1.01	2.43	1.00	1.00	1.01	0.96
misex3	269.65	3.37E-8	240.00	2.73E-8	240.93	2.09E-8	1.12	1.23	1.00	1.00	1.00	0.76
pdc	1210.80	5.67E-8	1108.66	3.39E-8	1103.05	3.25E-8	1.09	1.67	1.00	1.00	0.99	0.96
S1423	70.24	1.88E-8	69.87	8.83E-9	69.30	8.77E-9	1.01	2.13	1.00	1.00	0.99	0.99
s298	452.99	4.31E-8	417.91	2.76E-8	411.53	2.79E-8	1.08	1.56	1.00	1.00	0.98	1.01
S38417	1969.65	6.83E-8	1898.96	3.29E-8	1965.55	3.04E-8	1.04	2.07	1.00	1.00	1.04	0.92
seq	349.44	4.02E-8	326.53	2.68E-8	326.32	2.48E-8	1.07	1.50	1.00	1.00	1.00	0.93
spla	851.73	5.15E-8	783.69	5.47E-8	780.89	3.20E-8	1.09	0.94	1.00	1.00	1.00	0.59
tseng	316.46	3.60E-8	302.56	2.34E-8	307.18	2.23E-8	1.05	1.54	1.00	1.00	1.02	0.95
GeoMean							**1.09**	**1.53**	**1.00**	**1.00**	**1.01**	**0.93**

Table 3 – Critical Path Delay Comparison with Singh and Brown Retiming Approach

	Normalized to VPR Results[2]			
Netlist	**Retime First & VPR Place**	**Singh & Brown Retimer [10]**	**Increm. Slack Place Only**	**Integr. Retime & Placement**
bigkey	1.00	0.93	0.66	0.67
diffeq	1.00	0.96	0.53	0.50
dsip	1.00	0.75	0.84	0.78
elliptic	1.00	0.93	0.59	0.57
frisc	1.00	0.90	0.41	0.40
S38417	1.00	0.82	0.48	0.45
tseng	1.00	0.91	0.65	0.62
GeoMean	**1.00**	**0.88**	**0.58**	**0.55**

[1] Notice that in the *Normalized to VPR Results*, the S*ingh & Brown Retimer* numbers from [10] have been normalized to the VPR results also reported in [10]. Our *Incremental Slack Placement Only* and *Integrated Retime and Placement* results have been normalized to VPR results given in Table 1.

Table 1 and Table 2 show the best of ten placement attempts. We report the raw and normalized geometric mean final placement wire cost and routed critical path delay for all three placement approaches. Looking at these values, we can first verify that using the incremental slack placement methodology as a starting point for our retiming tool was indeed a wise choice. By itself, it produced placements with 1.11x better critical path delay without affecting wirelength for the original sequential MCNC netlists and 1.53x better critical path delay with 1.09x better wirelength for the heavily pipelined circuits. If we look at the results when we apply our simulated annealing-based retimer we see that the gap widens slightly to 1.19x better critical path delay for the original sequential MCNC netlists and 1.64x better critical path delay for the heavily pipelined circuits.

If we consider the gains solely due to our retiming methodology, we see that both the original sequential MCNC netlists and the heavily pipelined circuits perform 1.08x faster. These delay improvements came with no cost to routability.

Unfortunately, any direct comparison to the Singh and Brown toolflow [10] is relatively difficult. We do not have access to their code base and the paper is fairly vague regarding their exact testing conditions. Specifically, while not specified in the paper we believe they were mapping to an architecture similar to the clustered *4x4lut_sanitized* system. Furthermore, they also did not test on minimum-sized architectures, but on slightly oversized devices that provided additional registers.

That said, they reported results for a subset of the sequential MCNC netlists that we used, with VPR as a comparison point. Thus, while not ideal, we can get some idea of how the two methodologies line up if we normalize both sets of data to their respective VPR runs. We show these results in Table 3. We can see that they achieved a 1.14x average improvement in critical path delay for these netlists while we achieved a 1.82x average improvement. Thus, our simulated annealing retiming approach nearly doubles the performance of the conventional VPR approach. That said, our incremental slack methodology by itself improves placements so much that it is much more difficult to achieve further gains. We believe it is fair to claim that the approach suggested by Singh and Brown benefits from some of the optimization opportunities VPR leaves on the table.

6. Implications

The integrated retiming and placement technique that we have presented has several key advantages over previous methodologies:

Our retiming is less disruptive and leads to more predictable results. This is largely the product of three main issues. First, because we perform multiple small retiming moves, we do not have to try and massively retime a single register through multiple levels of logic all at once. Second, since our approach is based around simulated annealing, we can leverage many of the natural balancing aspects of the cost function and cooling schedule. Third, each of these smaller retiming moves is smoothed into the rest of the placement using the well-understood aspects of full simulated annealing.

Guaranteed legal placements, regardless of architecture. Since we never attempt to retime a logic block if it would create an illegal placement, we can never have problems with legalization.

This is a one-pass CAD flow, with no convergence issues. Since the retimer is built directly into the annealing process and is already naturally iterative, developers do not have to worry about not being happy with the final results, then attempting to re-run the tools and face problems with timing convergence.

7. Conclusions

In this paper we have investigated some of the issues surrounding registered netlists, placement tools and retiming methodologies. Specifically, we identified a key characteristic of many existing CAD approaches that either limits the benefits of retiming or puts placement convergence at risk: we cannot apply aggressive retiming to a netlist as a separate, single-use step and then expect a post-processing phase to be able to clean up the results satisfactorily.

As an alternative, we presented a new integrated approach that performs retiming during simulated-annealing placement. This approach allows us to leverage many of the benefits of the simulated annealing framework to produce a minimally disruptive retimed netlist. In our testing against VPR, we demonstrated that our technique produced 1.19x better critical path delay without negatively affecting routability for lightly registered netlists, and 1.64x better critical path delay with 1.09x better wirelength for heavily pipelined netlists. Our approach produced 1.08x faster critical path delay than the highly-improved timing-driven placement tool it was based on for both lightly registered and heavily pipelined applications.

Perhaps most importantly though, this integrated retiming technique is inherently convergent and architecture-independent. Thus, it opens the door to the development of practical register-rich FPGAs.

8. Acknowledgments

This work was sponsored in part by grant *CCF0426147* from the National Science Foundation, and grant *DE-FG52-06NA27507* from the Department of Energy. The authors would also like to thank Nikhil

Subramanian for his background research into the Xilinx Virtex II architecture.

9. References

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

[2] Cong, J. and S. Lim, “Physical Planning with Retiming”, International Conference on Computer-Aided Design, 2000: 2-7.

[3] Cong, J. and X. Yuan, “Multilevel Global Placement with Retiming”, Design Automation Conference, 2003: 208-13.

[4] Eguro, K. and S. Hauck, “Enhancing Timing-Driven FPGA Placement for Pipelined Netlists”, to Appear in Design Automation Conference, 2008.

[5] Eguro, K. and S. Hauck. “Armada: Timing-Driven Pipeline-Aware Routing for FPGAs”, ACM/SIGDA Symposium on Field-Programmable Gate Arrays, 2006: 169-78.

[6] Marquardt, A., V. Betz and J. Rose, “Timing-Driven Placement for FPGAs”, ACM/SIGDA Symposium on Field Programmable Gate Arrays, 2000: 203-13.

[7] Leiserson, C. and J. Saxe, “Retiming Synchronous Circuitry”, Algorithmica, Vol. 6, 1991: 5-35.

[8] Seidl, R., K. Eckl, and F. Johannes, “Performance-directed Retiming for FPGAs using Post-placement Delay Information”, Design, Automation and Test in Europe Conference, 2003: 770-5.

[9] Sharma, A., C. Ebeling and S. Hauck. "PipeRoute: A Pipelining-Aware Router for FPGAs", *ACM/SIGDA Symposium on Field-Programmable Gate Arrays*, 2003: 68-77.

[10] Singh, D. and S. Brown, “Integrated Retiming and Placement for Field Programmable Gate Arrays”, ACM/SIGDA Symposium on Field Programmable Gate Arrays, 2002: 67-76.

[11] Singh, D. and S. Brown, “The Case for Registered Routing Switches in Field Programmable Gate Arrays”, *ACM/SIGDA Symposium on Field-Programmable Gate Arrays*, 2001: 161-9.

[12] Tsu, W., K. Macy, A. Joshi, R. Huang, N. Walker, T. Tung, O. Rowhani, V. George, J. Wawrzynek, and A. DeHon. "HSRA: High-Speed, Hierarchical Synchronous Reconfigurable Array", ACM/SIGDA Symposium on Field Programmable Gate Arrays, 1999: 125-34.

[13] Von Herzen, B., “Signal Processing at 250MHz using High-Performance FPGA’s”, ACM International Symposium on FPGAs. 1997, 62-8.

[14] Weaver, N., J. Hauser, and J. Wawrzynek, “The SFRA: A Corner-Turn FPGA Architecture”, ACM/SIGDA International Symposium on Field Programmable Gate Arrays, 2004: 3-12.

[15] Weaver, N., Y. Markovskiy, T. Patel, and J. Wawrzynek, “Post-Placement C-slow Retiming for the Xilinx Virtex FPGA”, ACM/SIGDA Symposium on Field-Programmable Gate Arrays, 2003: 185-94.