CHAPTER 18

RETIMING, REPIPELINING, AND *C*-SLOW RETIMING

Nicholas Weaver
International Computer Science Institute

Although pipelining is a huge benefit in field-programmable gate array (FPGA) designs, and may be required on some FPGA fabrics [5, 10, 12], it is often difficult for a designer to manage and balance pipeline stages and to insert the necessary delays to meet design requirements.

Leiserson et al. [4] were the first to propose *retiming*, an automatic process to relocate pipeline stages to balance a design. Their algorithm, in $O(n^2 lg(n))$ time, can rebalance a design so that the critical path is optimally pipelined. In addition, two modifications, *repipelining* and *C-slow retiming*, can add additional pipeline stages to a design to further improve the critical path.

The key idea is simple: If the number of registers around every cycle in the design does not change, the end-to-end symantics do not change. Thus, retiming attempts to solve two primary constraints: All paths longer than the desired critical path are registered, and the number of registers around every cycle is unchanged.

This optimization is useful for conventional FPGAs but absolutely essential for fixed-frequency FPGA architectures, which are devices that contain large numbers of registers and are designed to operate at a fixed, but very high, frequency, often by pipelining the interconnect as well as the computation.

To meet the array's fixed frequency, a design must ensure that every path is properly registered. Repipelining or *C*-slow retiming enables a design to be transformed to meet this constraint. Without automated repipelining or *C*-slow retiming, the designer must manually ensure that all pipeline constraints are met by the design.

Retiming operates by determining an optimal placement for existing registers, while repipelining and *C*-slowing add registers before the retiming process begins. After retiming, the design should be optimally (or near-optimally) balanced, with no pipeline stage requiring significantly more time than any other stage.

Section 18.1 describes the basic retiming operation and the retiming algorithm and its semantics. Then Section 18.2 discusses repipelining and *C*-slowing: two different techniques for adding registers. Repipelining improves feedforward designs by adding additional pipelining stages, while *C*-slowing creates

an interleaved design by replacing every register with a sequence of C registers. Both of these transformations increase throughput but also increase latency.

Section 18.3 surveys the various implementations, beginning with Leiserson's original algorithm and concluding with both academic and commercial tools. Section 18.4 discusses implementing retiming for fixed-frequency arrays. Unlike general FPGAs, fixed-frequency FPGAs require retiming in order to match user designs with architectural constraints. Finally, Section 18.5 discusses an interesting side effect of *C*-slowing: the creation of interleaved, multi-threaded architectures. We conclude in Section 18.6 with a discussion of the reasons that retiming is not a ubiquitous optimization in FPGA tool flows.

18.1 RETIMING: CONCEPTS, ALGORITHM, AND RESTRICTIONS

The goal of retiming is to move the pipeline registers in a design into the optimal position. Figure 18.1 shows a trivial example. In this design, the nodes represent logic delays (a), with the inputs and outputs passing through mandatory, fixed registers. The critical path is 5, and the input and output registers cannot be moved. Figure 18.1(b) shows the same graph after retiming. The critical path is reduced from 5 to 4, but the I/O semantics have not changed, as three cycles are still required for a datum to proceed from input to output.

As can be seen, the initial design has a critical path of 5 between the internal register and the output. If the internal register could be moved forward, the critical path would be shortened to 4. However, the feedback loop would then be incorrect. Thus, in addition to moving the register forward, another register would need to be added to the feedback loop, resulting in the final design.

Additionally, even if the last node is removed, it could never have a critical path lower than 4 because of the feedback loop. There is no mechanism that can reduce the critical path of a single-cycle feedback loop by moving registers: Only additional registers can speed such a design.

Retiming's objective is to automate this process: For a graph representing a circuit, with combinational delays as nodes and integer weights on the edges, find a new assignment of edge weights that meets a targeted critical path or fail if the critical path cannot be met. Leiserson's retiming algorithm is guaranteed to find such an assignment, if it exists, that both minimizes the critical path and ensures that around every loop in the design the number of registers always remains the same. It is this second constraint, ensuring that all feedback loops

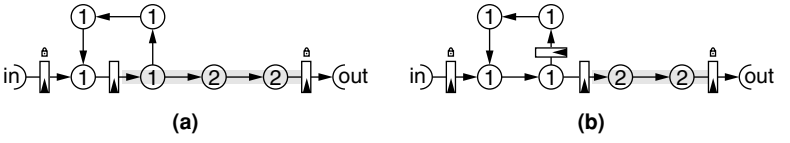

FIGURE 18.1 ■ A small graph before retiming (a) and the same graph after retiming (b).

TABLE 18.1 ■ The constraint system used by the retiming procsess

Condition normal edge from $u \rightarrow v$	Constraint $r(u) - r(v) \leq w(e)$
Edge from $u \rightarrow v$ must be registered	$r(u) - r(v) \leq w(e) - 1$
Edge from $u \rightarrow v$ can never be registered	$r(u) - r(v) \leq 0$ and $r(v) - r(u) \leq 0$
Critical paths must be registered	$r(u) - r(v) \leq W(u, v) - 1$ for all u, v such that $D(u, v) > P$

are unchanged, which ensures that retiming doesn't change the semantics of the circuit. In Table 18.1, $r(u)$ is the lag computed for each node (which is used to determine the final number of registers on each edge), $w(e)$ is the initial number of registers on an edge, $W(u,v)$ is the minimum number of registers between u and v, and $D(u,v)$ is the critical path between u and v.

Leiserson's algorithm takes the graph as input and then adds an additional node representing the external world, with appropriate edges added to account for all I/Os. This additional node is necessary to ensure that the circuit's global I/O semantics are unchanged by retiming.

Two matrices are then calculated, W and D, that represent the number of registers and critical path between every pair of nodes in the graph. These matrices are necessary because retiming operates by ensuring that at least one register exists on every path that is longer than the critical path in the design.

Each node also has a lag value r that is calculated by the algorithm and used to change the number of registers that will be placed on any given edge. Conventional retiming does not change the design semantics: All input and output timings remain unchanged while minor design constraints are imposed on the use of FPGA features. More details and formal proofs of correctness can be found in Leiserson's original paper [4].

The algorithm works as follows:

1. Start with the circuit as a directed graph. Every node represents a computational element, with each element having a computational delay. Each edge can have zero or more registers as a weight w. Add an additional dummy node with 0 delay, with an edge from every output and to every input. This additional node is to ensure that from every input to every output the number of registers is unchanged and therefore the data input to output timing is unaffected.

2. Calculate W and D. D is the critical path for every node to every other node, and W is the initial number of registers along this path. This requires solving the all-pairs shortest-path problem, of which the optimal algorithm, by Dijkstra, requires $O(n^2 lg(n))$ time. This dominates the asymptotic running time of the algorithm.

3. Choose a target critical path and create the constraints, as summarized in Table 18.1. Each node has a lag value r, which will eventially specify the *change* in the number of registers between each node. Initialize all nodes to have a lag of 0.

4. Since all constraints are pairwise integer inequalities, the Bellman–Ford constraint solver is guaranteed to find a solution if one exists or to terminate if not. The Bellman–Ford algorithm performs N iterations (N = the number of constraints to solve). In each iteration, every constraint is examined. If a constraint is already satisified, nothing happens. Otherwise, $r(u)$ or $r(v)$ is decremented to meet the particular constraint. Once an iteration occurs where no values change, the algorithm has found a solution. If there is no solution, after N iterations the algorithm terminates with a failure.

5. If the constraint solver fails to find a solution, or a tighter critical path is desired, choose a new critical path and return to step 3.

6. With the final set of constraints, a new set of registers is constructed for each edge, $w' \cdot w'(e) = w(e) - r(u) + r(v)$.

A graphical example of the algorithm's results is shown in Figure 18.1. The initial graph has a critical path of 5, which is clearly nonoptimal. After retiming, the graph has a critical path of 4, but the I/O semantics have not changed, as any input will still require three cycles to affect the output. To determine whether a critical path P can be achieved, the retiming algorithm creates a series of constraints to calculate the lag on each node (Table 18.1).

The primary constraints ensure correctness: No edge will have a negative number of registers, while every cycle will always contain the original number of registers. All I/O passes through the intermediate node, ensuring that input and output timings do not change. These constraints can be modified so that a particular line will contain no registers, or a mandatory minimum number of registers, to meet architectural constraints without changing the complexity of the equations. But it is the final constraint, that all critical paths above a predetermined delay P are registered, that gives this optimization its effectiveness.

If the constraint system has a solution, the new lag assignments for all nodes will allocate registers properly to meet the critical path P. But if there is no solution, there cannot be an assignment of registers that meets P. Thus, the common usage is to find the minimum P where the constraints are all met.

In general, multiple constraint-solving attempts are made to search for the minimum critical path P. The constraints for P are the final retimed design. There are two ways to speed up this process. First, if the Bellman–Ford algorithm can find a solution, it usually converges very quickly. Thus, if there is no solution that satisfies P, it is usually effective to abandon the Bellman–Ford algorithm early after $0.1N$ iterations rather than N iterations. This seems to have no impact on the quality of results, yet it can greatly speed up searching for the minimum P that can be satisfied in the design.

A second optimization is to use the last computed set of constraints as a starting point. In conventional retiming, the Bellman–Ford process is invoked multiple times to find the lowest satisfiable critical path. In contrast, fixed-frequency repipelining or *C*-slow retiming uses Bellman–Ford to discover the minimum number of additional registers needed to satisfy the constraints. In both cases,

keeping the last failed or successful solution in the data structure provides a starting point that can significantly speed up the process if a solution exists.

Retiming in this way imposes only minimal design limitations: Because it applies only to synchronous circuits, there can be no asynchronous resets or similar elements. A synchronous global reset imposes too many constraints to allow effective retiming. Local synchronous resets and enables only produce small, self loops that have no effect on the correct operation of the algorithm.

Most other design features can be accommodated simply by adding appropriate constraints. For example, an FPGA with a tristate bus cannot have registers placed on this bus. A constraint that says that all edges crossing the bus can never be registered ($r(u) - r(v) \leq 0$ and $r(v) - r(u) \leq 0$) ensures this. Likewise, an embedded memory with a mandatory output flip-flop can have a constraint ($r(u) - r(v) \leq w(e) - 1$) that ensures that at least one register is placed on this output.

Memories themselves can be retimed similarly to any other element in the design, with dual-ported memories treated as a single node for retiming purposes. Memories that are synthesized with a negative clock edge (to create the design illusion of asynchronicity) can be either unchanged or switched to operate on the positive edge with constraints to mandate the placement of registers.

Some FPGA designs have registers with predefined initial values. If retiming is allowed to move these registers, the proper initial values must be calculated such that the circuit still produces the same behavior.

In an ASIC model, all flip-flops start in an undefined state, and the designer must create a small state machine in order to reset the design. FPGAs, however, have all flip-flops start in a known, user-defined state, and when a dedicated global reset is applied the flip-flops are reset to it. This has serious implications in retiming.

If the decision is made to utilize the ASIC model, retiming is free to safely ignore initial conditions because explicit reset logic in state machines will still operate correctly—this is reflected in the I/O semantics. However, without the ability to violate the initial conditions with an ASIC-style model, retiming quality often suffers as additional logic is required or limits are placed on where flip-flops may be moved in a design.

In practice, performing retiming with initial conditions is NP-hard. Cong and Wu [3] have developed an algorithm that computes initial states by restricting the design to forward retiming only so that it propagates the information and registers forward throughout the computation. This is because solving initial states for all registers moved forward is straightforward, but backward movement is NP hard as it reduces to satisfiability.

Additionally, global set/reset imposes a huge constraint on retiming. An asynchronous set/reset can never be retimed (retiming cannot modify an asynchronous circut) while a synchronous set/reset just imposes too high a fanout.

An important question is how to deal with multiple clocks. If the interfaces between the clock domains are registered by clocks from both domains, it is a simple process to retime the domains separately, with mandatory registers

TABLE 18.2 ■ The results of retiming four benchmarks

Benchmark	Unretimed	Automatically retimed
AES core	48 MHz	47 MHz
Smith/Waterman	43 MHz	40 MHz
Synthetic datapath	51 MHz	54 MHz
LEON processor	23 MHz	25 MHz

on the domain crossings—the constraints placed on the I/Os ensure correct and consistent timing through the interface. Yet without this design constraint, retiming across multiple clock domains is very hard, and there does not appear to be any clean automatic solution.

Table 18.2 shows the results for a particular retiming tool [13]—the Xilinx Virtex family of FPGAs—on four benchmark circuits: an AES core, a Smith/Waterman systolic cell, a synthetic microprocessor datapath, and the LEON-I synthesized SPARC core. This tool does not use a perfectly accurate delay model and has to place registers after retiming, so it sometimes creates slightly suboptimal results.

The biggest problem with retiming is that it is of limited benefit to a well-balanced design. As mentioned earlier, if the clock cycle is defined by a single-cycle feedback loop, retiming can never improve the design, as moving the register around the feedback loop produces no effect.

Thus, for example, the Smith–Waterman example in Table 18.2 does not benefit from retiming. The Smith–Waterman benchmark design consists of a series of repeated identical systolic cells that implement the Smith–Waterman sequence alignment algorithm. The cells each contain a single-cycle feedback loop, which cannot be optimized. The AES encryption algorithm also consists of a single-cycle feedback loop. In this case, the initial design used a negative-edge Block-RAM to implement the *S*-boxes, which the retiming tool converted to a positive edge memory with a "must register" constraint.

Nevertheless, retiming can still be a benefit if the design consists of multiple feedback loops (such as the synthetic microprocessor datapath or the LEON SPARC–compatible microprocessor core) or an initially unbalanced pipeline. Still, for well-designed circuits, even complex ones, retiming is often only a slight benefit, as engineers have considerable experience designing reasonably optimized feedback loops.

The key benefit to retiming occurs when more registers can be added to the design along the critical path. We will discuss two techniques, repipelining and *C*-slow retiming, which first add a large number of registers that general retiming can then move into the optimal location.

18.2 REPIPELINING AND *C*-SLOW RETIMING

The biggest limitation of retiming is that it simply cannot improve a design beyond the design-dependent limit produced by an optimal placement of

registers along the critical path. As mentioned earlier, if the critical path is defined by a single-cycle feedback loop, retiming will completely fail as an optimization. Likewise, if a design is already well balanced, changing the register placement produces no improvement. As was seen in the four reasonably optimized benchmarks (refer to Table 18.2), this is often the case.

Repipelining and *C*-slow retiming are tranformations designed to add registers in a predictible matter that a designer can account for, which retiming can then move to optimize the design. Repipelining adds registers to the beginning or end of the design, changing the pipeline latency but no other semantics. *C*-slow retiming creates an interleaved design by replacing every register with a sequence of *C* registers.

18.2.1 Repipelining

Repipelining is a minor extension to retiming that can increase the clock frequency for feedforward computations at the cost of additional latency through more pipeline registers. Unlike *C*-slow retiming, repipelining is only beneficial when a computation's critical path contains no feedback loops.

Feedforward computations, those that contain no feedback loops, are commonly seen in DSP kernels and other tasks. For example, the discrete cosine transform (DCT), the fast Fourier transform (FFT), and finite impulse response filters (FIRs) can all be constructed as feedforward pipelines.

Repipelining is derived from retiming in one of two ways, both of which create semantically equivalent results. The first involves adding additional pipeline stages to the start of the computation and allowing retiming to rebalance the delays and create an absolute number of additional stages. The second involves decoupling the inputs and outputs to allow the retimer to add additional pipelining. Although these techniques operate in slightly different ways, they both provide extra registers for the retimer to then move and they produce roughly equivalent results.

If the designer wishes to add *P* pipeline stages to a design, all inputs simply have *P* delays added before retiming proceeds. Because retiming will develop an optimum placement for the resulting design, the new design contains *P* additional pipeline stages that are scattered throughout the computation. If a CAD tool supports retiming but not repipelining, the designer can simply add the registers to the input of the design manually and let the tool determine the optimum placement.

Another option is to simply remove the cycle between all outputs and inputs, with additional constraints to ensure that all outputs share an output lag, with all inputs sharing a different input lag. This way, the inputs and outputs are all synchronized but retiming can add an arbitrary number of additional pipeline registers between them. To place a limit on these registers, an additional constraint must be added to ensure that for a single I/O pair no more than *P* pipeline registers are added. Depending on the other constraints in the retiming process, this may add fewer than *P* additional pipeline stages, but will never add more than *P*.

Repipelining adds additional cycles of latency to the design, but otherwise retains the rest of the circuit's behavoir. Thus, it produces the same results and the same relative timing on the outputs (e.g., if input *B* is supposed to be presented three cycles after input *A*, or output *C* is produced two cycles after output *D*, these relative timings remain unchanged). It is only the data-in to data-out timing that is affected.

Unfortunately, repipelining can only improve feedforward designs or designs where the feedback loop is not on the critical path. If performance is limited by a feedback loop, repipelining offers no benefit over normal retiming.

Repipelining is designed to improve throughput, but will almost always make overall latency worse. Although the increased pipelining will boost the clock rate (and thus reduce some of the delay from unbalanced clocked paths), the delay from additional flip-flops on the input-to-output paths typically overwhelms this improvement and the resulting design will take longer to produce a result for an individual input.

This is a fundamental trade-off in repipelining and *C*-slow retiming. While ordinary retiming improves both latency and throughput, repipelining and *C*-slow retiming generally improve throughput at the cost of additional latency due to the additional pipeline stages required.

18.2.2 *C*-slow Retiming

Unlike repipelining, *C*-slow retiming can enhance designs that contain feedback loops. *C*-slowing enhances retiming simply by replacing every register with a sequence of *C* separate registers before retiming occurs; the resulting design operates on *C* distinct execution tasks. Because all registers are duplicated, the computation proceeds in a round-robin fashion, as illustrated in Figure 18.2.

In this example, which is 2-slow, the design interleaves between two computations. On the first clock cycle, it accepts the first input for the first stream of execution. On the second clock cycle, it accepts the first input for the second stream, and on the third it accepts the second input for the first stream. Because of the interleaved nature of the design, the two streams of execution will *never* interfere. On odd clock cycles, the first stream of execution accepts input; on even clock cycles, the second stream accepts input.

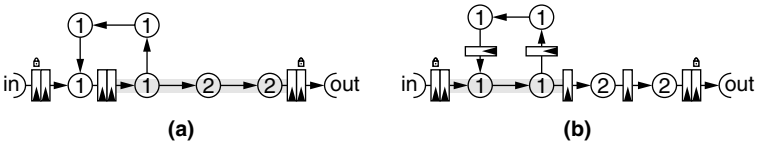

FIGURE 18.2 ■ The example from Figure 18.1, converted to 2-slow operation (a). The critical path remains unchanged, but the design now operates on two independent streams in a round-robin fashion. The design retimed (b). By taking advantage of the extra flip-flops, the critical path has been reduced from 5 to 2.

The easiest way to utilize a *C*-slowed block is to simply multiplex and de-multiplex *C* separate datastreams. However, a more sophisticated interface may be desired depending on the application (as described in Section 18.5).

One possible interface is to register all inputs and outputs of a *C*-slowed block. Because of the additional edges retiming creates to track I/Os and to ensure a consistent interface, every stream of execution presents all outputs at the same time, with all inputs registered on the next cycle. If part of the design is *C*-slowed, but all parts operate on the same clock, the result can be retimed as a complete whole and still preserve all other semantics.

One way to think of *C*-slowing is as a threaded design, with an overall system clock and with each stream having a "stream clock" of 1/*C*—each stream is completely independent. However, *C*-slowing imposes some more significant FPGA design constraints, as summarized in Table 18.3. Register clock enables and resets must be expressed as logic features, since each independent thread must have an independent reset or enable. Thus, they can remain features in the design but cannot be implemented by current FPGAs using native enables and resets. Other specialized features, such as Xilinx SRL16s (a mode where a LUT is used as a 16-bit shift register), cannot be utilized in a *C*-slow design for the same reason.

One important challenge is how to properly *C*-slow memory blocks. In cases where the *C*-slowed design is used to support *N* independent computations, one needs the illusion that each stream of execution is completely independent and unchanged. To create this illusion, the memory capacity must be increased by a factor of *C*, with additional address lines driven by a thread counter. This ensures that each stream of execution enjoys a completely separate memory space.

For dual-ported memories, this potentially enables a greater freedom in retiming: The two ports can have different lags as long as the difference in lag is less than *C*. After retiming, the difference is added to the appropriate port's thread counter, which ensures that each stream of execution will read and write to both ports in order while enabling slightly more freedom for retiming to proceed.

C-slowing normally guarantees that all streams view independent memories. However, a designer may desire shared memory common to all streams. Such

TABLE 18.3 ■ The effects of various FPGA features on retiming, repipelining, and *C*-slowing

FPGA feature	Effect on retiming	Effect on repipelining	Effect on *C*-slowing
Asynchronous global set/reset	Forbidden	Forbidden	Forbidden
Synchronous global set/reset	Effectively forbidden	Effectively forbidden	Forbidden
Asynchronous local set/reset	Forbidden	Forbidden	Forbidden
Synchronous local set/reset	Allowed	Allowed	Express as logic
Clock enables	Allowed	Allowed	Express as logic
Tristate buffers	Allowed	Allowed	Allowed
Memories	Allowed	Allowed	Increase size
SRL16	Allowed	Allowed	Express as logic
Multiple clock domains	Design restrictions	Design restrictions	Design restrictions

memories could be embedded in a design, but the designer would need to consider how multiple streams would affect the semantics and would need to notify any automatic tool to treat the memory in a special manner. Beyond this, there are no other semantic effects imposed by *C*-slow retiming.

C-slowing significantly improves throughput, but it can only apply to tasks where there are at least *C* independent threads of execution and where throughput is the primary goal. The reason is that *C*-slowing makes the latency substantially worse. This trade-off brings up a fundimental observation: Latency is a property of the design and computational fabric whereas throughput is a property derived from cost. Both repipelining and *C*-slow retiming can be applied only when there is sufficient task-level parallelism, in the form of either a feed-forward pipeline (repipelining) or independent tasks (*C*-slowing).

Table 18.4 shows the difference that *C*-slowing can make in four designs. While the retiming tool alone was unable to improve the AES or Smith Waterman designs, *C*-slowing substantially increased throughput, improving the clock rate by 80–95 percent! However, latency for individual tasks was made worse, resulting in significantly slower clock rates for individual tasks.

Latency can be improved only up to a given point for a design through conventional retiming. Once the latency limit is met, no amount of optimization, save a major redesign or an improvement in the FPGA fabric, has any effect. This often appears in cryptographic contexts, where feedback mode–based encryption (such as CFB) requires the complete processing of each block before the next can be processed.

In contrast, throughput is actually a part of a throughput/cost metric: throughput/area, throughput/dollar, or throughput/joule. This is because independent task throughput can be added via replication, creating independent modules that perform the same function, as well as *C*-slowing. When sufficient parallelism exists, and costs are not constrained, simply throwing more resources at the problem is sufficient to improve the design to meet desired goals.

One open question on *C*-slowing is its effect in a low-power environment. Higher throughput, achieved through high-speed clocking, naturally increases the power consumption of a design, just as replicating units for higher throughput increases power consumption. In both cases, if lower power is desired, the higher-throughput design can be modified to save power by reducing the clock rate and operating voltage.

Unlike the replicated case, the question of whether a *C*-slowed design would offer power savings if both frequency and voltage were reduced is highly design

TABLE 18.4 ■ The effect of *C*-slowing on four benchmarks

Benchmark	Initial clock	*C*-factor	*C*-slow clock	Stream clock
AES encryption	48 MHz	4-slow	87 MHz	21 MHz
Smith/Waterman	43 MHz	3-slow	84 MHz	28 MHz
Synthetic datapath	51 MHz	3-slow	91 MHz	30 MHz
LEON processor core	23 MHz	2-slow	46 MHz	23 MHz

and usage dependent. Although the finer pipelining allows the frequency and the voltage to be scaled back to a significant degree while maintaining throughput, the activity factor of each signal may now be considerably higher. Because each of the *C* streams of execution is completely independent, it is safe to assume that every wire will probably have a significantly higher activity factor that increases power consumption.

Whether the initial design before *C*-slowing has a comparable activity factor is highly input and design dependent. If the initial design's activity factor is low, *C*-slowing will significantly increase power consumption. But if that factor is high, *C*-slowing will not increase it. Thus, although the *C*-slowing transformation may have a minor affect on worst-case power (and can even result in significant savings through voltage scaling), the impact on average-case power may be substantial.

18.3 IMPLEMENTATIONS OF RETIMING

Three significant academic retiming tools have been developed for FPGAs. The first, by Cong and Wu [3], combines retiming with technology mapping. This approach enables retiming to occur before placement without adding undue constraints on the placer, because the retimed registers are packed with their associated logic. The disadvantage is a lack of precision, as delays can only be crudely estimated before placement. This tool is unsuitable for significant *C*-slowing, which creates significantly more registers that can pose problems with logic packing and placement.

The second tool, developed by Singh and Brown [6], combines retiming with placement, operating by modifying the placement algorithm to be aware that retiming is occurring and then modifying the retiming portion to enable permutation of the placement as retiming proceeds. Singh and Brown demonstrate how the combination of placement and retiming performs significantly better than retiming either before or after placement.

The simplified FPGA model used by Singh and Brown has a logic block where the flip-flop cannot be used independently of the LUT, constraining the ability of postplacement retiming to allocate new registers. Thus, the need to permute the placement to allocate registers is significantly exacerbated in their target architecture.

The third tool, developed by Weaver et al. [13], performs retiming after placement but before routing, taking advantage of the (mostly) independent register operation available on Xilinx FPGAs. (It would not apply to most Altera FPGAs.) It too also supports *C*-slowing.

Some commercial HDL synthesis tools, notably the Synopsys FPGA compiler [9] and Synplify [8], also support retiming. Because this retiming occurs fairly early in the mapping and optimization processes, it suffers from a lack of precision regarding placement and routing delays. The Amplify tool [10] can produce a higher-quality retiming because it contains placement information. Since these

tools attempt to maintain the FPGA model of initial conditions, both on startup and in the face of a global reset signal, considerable logic is added to the design.

18.4 RETIMING ON FIXED-FREQUENCY FPGAs

Fixed-frequency FPGAs differ from conventional FPGAs in that they have an intrinsic clock rate and commonly include pipelined interconnect and other design features to enable very high-speed operations. However, this fixed frequency demands a design modification to support the pipeline stages it requires.

Retiming for fixed-frequency FPGAs, unlike that for their conventional counterparts, does not require the creation of a global critical path constraint, as simply ensuring that all local requirements are met guarantees that the final design meets the architecture's required delay constraints. Instead, retiming attempts to solve these local constraints by ensuring that every path through the interconnect meets the delay requirements inherent in the FPGA. Once these local constraints are met, the final design will operate at the FPGA's intrinsic clock frequency.

Because there are no longer any global constraints, the W and D matrices are not created. A fixed-frequency FPGA does not require the global constraints, so having only to solve a set of local constraints requires linear, not quadratic, memory and $O(n^2)$, rather than $O(n^2 lg(n))$, execution time. This speeds the process considerably.

Additionally, only a single invocation of the constraint solver is necessary to determine whether the current level of pipelining can meet the constraints imposed by the target architecture. Unfortunately, most designs do not possess sufficient pipelining to meet these constraints, instead requiring a significant level of repipelining or *C*-slow retiming to do so. The level necessary can be discovered in two ways.

The first approach is simply to allow the user to specify a desired level of repipelining or *C*-slowing. The retiming system then adds the specified number of delays and attempts to solve the system. If a solution is discovered, it is used. Otherwise, the user is notified that the design must be repipelined or retimed to a greater degree to meet the array's clock cycle. The second approach requires searching to find the minimal level of repipelining or *C*-slowing necessary to meet the constraints. Although this necessitates multiple iterations of the constraint solver, fixed-frequency retiming only requires local constraints. Without having to check the global constraints, this process proceeds quickly. The resulting level of repipelining or *C*-slowing is then reported to the user.

Fixed-frequency FPGAs require retiming considerably later in the tool flow. It is impossible to create a valid retiming until routing delays are known. Since the constraints required invariably depend on placement, the final retiming process must occur afterwards. Some arrays, such as HSRA [10], have deterministic routing structures that enable retiming to be performed either before or after routing. Other interconnect structures, such as SFRA [12], lack deterministic routing and require that retiming be performed only after routing.

Finally, the fact that fixed-frequency arrays may use considerably more pipelining than conventional arrays makes retiming registers a significant architectural feature. Because these delay chains [10], either on inputs or on outputs, are programmable, the array can implement longer ones. A common occurrence after aggressive *C*-slow retiming is a design with several signals requiring considerable delay. Therefore, dedicated resources to implement these features are effectively required to create a viable fixed-frequency FPGA.

18.5 *C*-SLOWING AS MULTI-THREADING

There have been numerous multi-threaded architecture designs, but all share a common theme: increasing system throughput by enabling multiple streams of execution, or threads, to operate simultaneously. These architectures generally fall into four classes: context switching always without bypassing (HEP [7] and Tera [2]), context switching on event (Intel IXP) [14], interleaved multi-threaded, and symmetric multi-threaded (SMT) [11]. The ideal goal of all of them is to increase system throughput by operating on multiple streams of execution.

The general concept of *C*-slow retiming can be applied to highly complex designs, including microprocessors. Unlike a simple FIR filter bank or an encryption algorithm, it is not a simple matter of inserting registers and balancing delays. Nevertheless, the changes necessary are comparatively small and the benefits substantial: producing a simple, statically scheduled, higher clock rate, multi-threaded architecture that is semantically equivalent to an interleaved-multi-threaded architecture, alternating between a fixed number of threads in a round-robin fashion to create the illusion of a multiprocessor system.

C-slowing requires three minor architectural changes: enlarging and modifying the register file and TLB, replacing the cache and memory interface, and slightly modifying the interrupt semantics. Beyond that, it is simply a matter of replacing every pipeline register in both the control logic and the datapath with *C* registers and then moving the registers to balance the delays, as is traditional in the *C*-slow retiming transformation and can be performed by an automatic tool. The resulting design, as expected, has full multi-threaded semantics and improved throughput because of a significantly higher clock rate. Figure 18.3 shows how this transformation can operate.

The biggest complications in *C*-slowing a microprocessor are selecting the implementation semantics for the various memories through the design. The first type keeps the traditional *C*-slow semantics of complete independence, where each thread sees a completely independent view, usually by duplication. This applies to the register file and most of the state registers in the system. This occurs automatically if *C*-slowing is performed by a tool, because it represents the normal semantics for *C*-slowed memory.

The second is completely shared memory, where every thread sees the same memory, such as the caches and main memory of the system. Most such memories exist in the non-*C*-slowed portion and so are unaffected by an automatic tool.

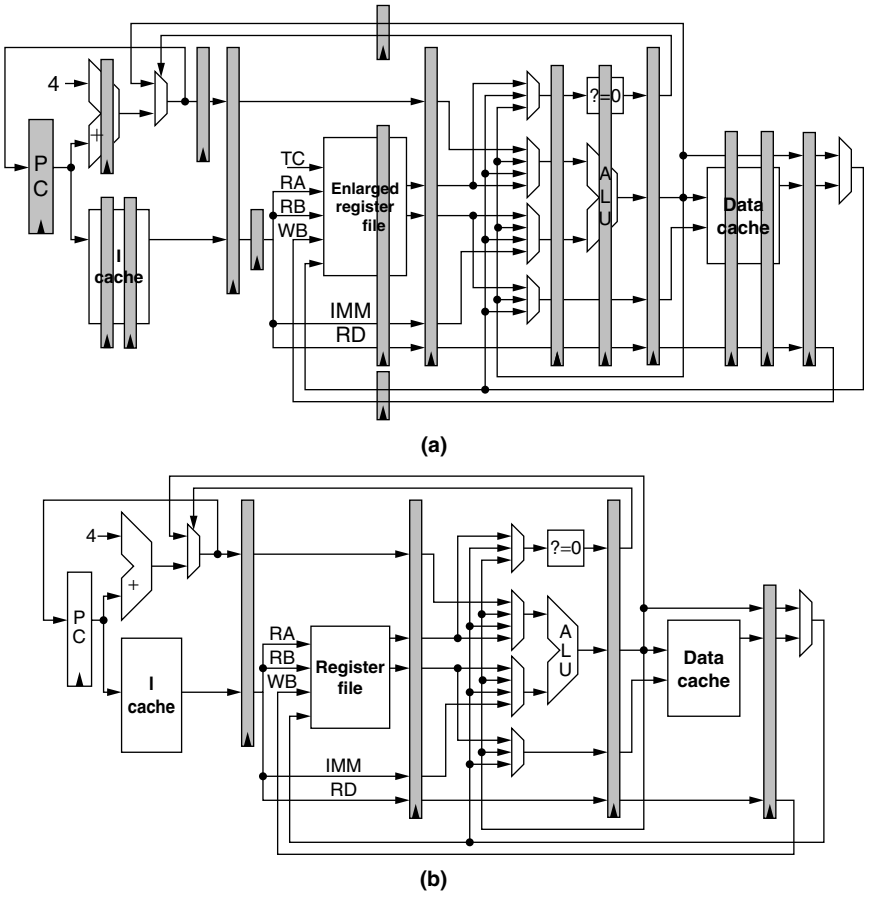

FIGURE 18.3 ■ A traditional five-stage microprocessor pipeline, and its conversion to 3-slow operation.

The third is dynamically shared, where a hardware thread ID or a software thread context ID is tagged to each entry, with only the valid tags used. This breaks the automatic *C*-slow semantics and is best employed for branch predictors and similar caches. Such memories need to be constructed manually, but offer potential efficiency advantages as they do not need to increase in size. Because they cannot be constructed automatically they may be subject to interference or synergistic effects between threads.

The biggest architectural changes are to the register file: It needs to be increased by a factor of *C*, with a hardware thread counter to select which group of registers is being accessed. Now each thread will see an independent set of registers, with all reads and writes for the different threads going to separate memory locations. Apart from the thread selection and natural enlargement, the only piece remaining is to pipeline the register access. If necessary, the

C independently accessed sections can be banked so that the register file can operate at a higher clock frequency.

Naturally, this linearly increases the size of the register file, but pipelining the new larger file is not difficult since each thread accesses a disjoint register set, allowing staggered access to the banks if desired. This matches the automatic memory transformations that *C*-slowing creates: increasing the size and ensuring that each task has an independent view of memory.

To maintain the illusion that the different threads are running on completely different processors, it is important that each thread have an independent translation of memory. The easiest solution is to apply the same transformations to the TLB that were applied to the register file: increasing the size by *C*, with each thread accessing its own set, and pipelining access. Again, this is the natural result of applying the *C*-slow semantics from an automatic tool.

The other option is to tag each TLB entry. The interference effect may be significant if the associativity or size of the TLB is low. In such a case, and considering the generally small size of most TLBs, increasing the size (although perhaps by less than a factor of *C*) is advisable. Software thread ID tags are preferable to hardware ID tags because they reduce the cost of context switching if a shared TLB is used and may also provide some synergistic effects. In either case, a shared TLB requires interlocking between TLB writes to prevent synchronization bugs.

If the caches are physically addressed, it is simply a matter of pipelining access to improve throughput without splitting memory. Because of the interlocked execution of the threads and the pipelined nature of the modified caches, no additional coherency mechanisms are required except to interlock any existing test-and-set or atomic read/write instructions between the threads to ensure that each instruction has time to be completed.

Such cache modifications occur outside the *C*-slow semantics, suggesting that the cache needs to be changed manually. This means that the cache and memory controller must be manually updated to support pipelined access from the distinct threads, and must exist outside of the *C*-slowed core itself.

Unfortunately, virtually addressed caches are significantly more complicated: They require that each tag include thread ownership (to prevent one thread from viewing another's version of memory) and that a record of virtual-to-physical mappings be maintained to ensure coherency between threads. These complications suggest that a physically addressed cache would be superior when *C*-slowing a microprocessor to produce a simple multi-threaded design. A virtually addressed cache is one of the few structures that do not have a natural *C*-slow representation or that can easily exist outside a *C*-slowed core.

The rest of the machine state registers, being both loaded and read, are automatically separated by the *C*-slow transformation. This ensures that each thread will have a completely independent set of machine registers. Combined with the distinct registers and TLB tagging, each thread will see an independent processor.

The only other portion that needs to be changed is the interrupt semantics. Just as the rest of the control logic is pipelined, with control registers duplicated,

the same transformations need to be applied to the interrupt logic. Thus, every external interrupt is interpreted by the rules corresponding to *every* virtual processor running in the pipeline. Yet, since the control registers are duplicated, the OS can enforce policies where different interrupts are handled by different execution streams. Similarly, internally driven interrupts (such as traps or watchdog timers), when *C*-slowed, are independent between threads, as *C*-slowing ensures that each thread sees only its own interrupts.

In this way, the OS can ensure that one virtual thread receives one set of externally sourced interrupts while another receives a different set. This also suggests that interrupts be presented to all threads of execution, enabling each thread (or even multiple threads) to service the appropriate interrupt.

The resulting design has full multi-threaded semantics, with each of *C* threads being independent. Because *C*-slowing can improve the clock rate (by two times in the case of the LEON benchmark), this can easily and substantially improve the throughput of a very complex design.

18.6 WHY ISN'T RETIMING UBIQUITOUS?

An interesting question is why retiming is not heavily used in FPGA tool flows. Although some FPGA vendors [1] and CAD vendors [8] support retiming, it is not universally available, and even when it is, it is usually optional.

There are three major factors that limit the general adoption of retiming: It interacts poorly with many critical FPGA features; it can only optimize poor implementations yet is not a substitute for good implementation; and it is computationally intensive.

As mentioned earlier, retiming does not work well with initial conditions or global resets—features that FPGA designers have traditionally relied on. Likewise, BlockRAMs, hardware clock eEnables, and other features can pin registers, limiting the ability of a retiming tool to move them. For these reasons, many FPGA designs *cannot* be effectively retimed.

A related observation is that retiming helps only poor designs and, moreover, only fixes one common deficiency of a poor design, not all of them. Additionally, if the designer has enough savvy to work around the limitations of retiming, he will probably produce a naturally well-balanced design.

Finally, although retiming is a polynomial time algorithm, its still superlinear. As designs continue to grow in size, $O(n^2 lg(n))$ can still be too long for many uses. This is especially problematic as the Moore's Law scaling for FPGAs is currently greater than that for single-threaded microprocessors.

References

[1] Altera Quartus II eda (*http://www.altera.com/*).

[2] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, B. Smith. The Tera computer system. *Proceedings of the 1990 International Conference on Supercomputing*, 1990.

[3] J. Cong, C. Wu. Optimal FPGA mapping and retiming with efficient initial state computation. *Design Automation Conference*, 1998.
[4] C. Leiserson, F. Rose, J. Saxe. Optimizing synchronous circuitry by retiming. *Third Caltech Conference On VLSI*, March 1993.
[5] H. Schmit. Incremental reconfiguration for pipelined applications. *Proceedings of the IEEE Symposium on Field-Programmable Gate Arrays for Custom Computing Machines*, April 1997.
[6] D. P. Singh, S. D. Brown. Integrated retiming and placement for field-programmable gate arrays. *Tenth ACM International Symposium on Field-Programmable Gate Arrays*, 2002.
[7] B. J. Smith. Architecture and applications of the HEP multiprocessor computer system. Advances in laser scanning technology. *SPIE Proceedings* 298, Society for Photo-Optical Instrumentation Engineers, 1981.
[8] Synplify pro (*http://www.synplicity.com//products//synplifypro//index.html*).
[9] Synopsys, Inc. Synopsis FPGA Compiler II (*http://www.synopsys.com*).
[10] W. Tsu, K. Macy, A. Joshi, R. Huang, N. Walker, T. Tung, O. Rowhani, V. George, J. Wawrzynek, A. DeHon. HSRA: High-speed, hierarchical synchronous reconfigurable array. *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, February 1999.
[11] D. M. Tullsen, S. J. Eggers, H. M. Levy. Simultaneous multi-threading: Maximizing on-chip parallelism. *Proceedings 22nd Annual International Symposium on Computer Architecture*, June 1995.
[12] N. Weaver, J. Hauser, J. Wawrzynek. The SFRA: A corner-turn FPGA architecture. *Twelfth International Symposium on Field-Programmable Gate Arrays*, 2004.
[13] N. Weaver, Y. Markovskiy, Y. Patel, J. Wawrzynek. Postplacement *C*-slow retiming for the Xilinx-Virtex FPGA. *Eleventh ACM International Symposium on Field-Programmable Gate Arrays*, 2003.
[14] Intel Corporation. The Intel IXP network processor. *Intel Technology Journal* 6(3), August 2002.