

# Post-Placement C-slow Retiming for the Xilinx Virtex FPGA

Nicholas Weaver<sup>\*</sup>  
UC Berkeley  
Berkeley, CA

Yury Markovskiy  
UC Berkeley  
Berkeley, CA

Yatish Patel  
UC Berkeley  
Berkeley, CA

John Wawrzynek  
UC Berkeley  
Berkeley, CA

## ABSTRACT

*C-slow retiming is a process of automatically increasing the throughput of a design by enabling fine grained pipelining of problems with feedback loops. This transformation is especially appropriate when applied to FPGA designs because of the large number of available registers. To demonstrate and evaluate the benefits of C-slow retiming, we constructed an automatic tool which modifies designs targeting the Xilinx Virtex family of FPGAs. Applying our tool to three benchmarks: AES encryption, Smith/Waterman sequence matching, and the LEON 1 synthesized microprocessor core, we were able to substantially increase the total throughput. For some parameters, throughput is effectively doubled.*

## Categories and Subject Descriptors

B.6.3 [Logic Design]: Design Aids—*Automatic synthesis*

## General Terms

Performance

## Keywords

FPGA CAD, FPGA Optimization, Retiming, C-slow Retiming

<sup>\*</sup>Please address any correspondence to nweaver@cs.berkeley.edu

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

FPGA'03, February 23–25, 2003, Monterey, California, USA.  
Copyright 2003 ACM 1-58113-651-X/03/0002 ...\$5.00.

## 1. Introduction

Leiserson's retiming algorithm[7] offers a polynomial time algorithm to optimize the clock period on arbitrary synchronous circuits without changing circuit semantics. Although a powerful and efficient transformation that has been employed in experimental tools[10][2] and commercial synthesis tools[13][14], it offers only a minor clock period improvement for a well constructed design, as many designs have their critical path on a single cycle feedback loop and can't benefit from retiming.

Also proposed by Leiserson et al to meet the constraints of systolic computation, is *C-slow retiming*.<sup>1</sup> In *C-slow retiming*, each design register is first replaced with *C* registers before retiming. This transformation modifies the design semantics so that *C* separate streams of computation are distributed through the pipeline, greatly increasing the aggregate throughput at the cost of additional latency and flip flops. This can automatically accelerate computations containing feedback loops by adding more flip-flops that retiming can then move around the critical path.

The effect of *C-slow retiming* is to enable pipelining of the critical path, even in the presence of feedback loops. To take advantage of this increased throughput however, there needs to be sufficient task level parallelism. This process will slow any single task but the aggregate throughput will be increased by interleaving the resulting computation.

This process works very well on many FPGA architectures as these architectures tend to have a balanced ratio of logic elements to registers, while most user designs contain a considerably higher percentage of logic. Additionally, many architectures allow the registers to be used independently of the logic in a logic block.

We have constructed a prototype *C-slow retiming* tool that modifies designs targeting the Xilinx Virtex family of FPGAs. The tool operates after placement: converting every design register to *C* separate registers before applying Leiserson's retiming algorithm to minimize the clock period. New registers are allocated by scavenging unused array resources. The resulting design is then returned to Xilinx tools for routing, timing analysis, and bitfile generation.

We have selected three benchmarks: AES encryption, Smith/Waterman sequence matching, and the LEON 1

<sup>1</sup>This was originally defined to meet systolic slowdown requirements.



Condition	Constraint
normal edge from $u \rightarrow v$	$r(u) - r(v) \leq w(e)$
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 $D(u, v) > P$

Table 1: The constraint system used by the retiming process.

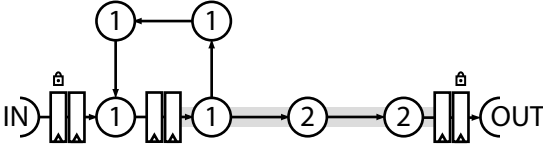


Figure 3: The example in Figure 2 2-slowed. This design now operates on 2 independent data streams.

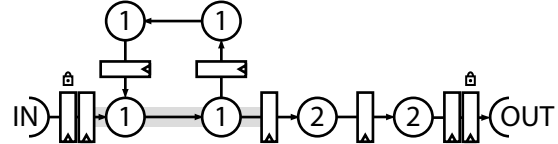


Figure 4: The example in Figure 3 after retiming. The combination of C-slowng and retiming reduced the critical path from 5 to 2.

unchanged or switch to operate on the positive edge with constraints to mandate the placement of registers.<sup>2</sup>

Initial register conditions can also be calculated if desired, but this process is NP hard in the general case. Cong and Wu[3] have an algorithm that computes initial states by restricting the design to forward retiming only, so it propagates the information and registers forward throughout the computation. This is because solving initial states for all registers moved forward is straightforward, but backwards movement is NP hard<sup>3</sup> as it reduces to satisfiability.

An important question is how to deal with multiple clocks. If the interfaces between the clock domains are registered by clocks from both domains, and with all signals being unidirectional, each clock domain can be treated as an independent block with all signals crossing the domain treated as I/O. Due to the retiming-imposed constraints on I/O, the logical view of each input will not change. However, constraints may be needed to insure that the physical registers remain in position to prevent asynchronous conditions from occurring on this interface.

### 3. C-slow retiming

C-slowng enhances retiming by simply replacing every register with a sequence of  $C$  separate registers before retiming occurs. The resulting design operates on  $C$  distinct execution tasks. Since all registers are duplicated, the computation proceeds in a round-robin fashion. The easiest way to utilize a  $C$  slowed block is to simply multi-

<sup>2</sup>For some cases, this may produce a set of unsolvable constraints, thus requiring that the memory remain a negative edge device.

<sup>3</sup>And may not possess a valid solution for nonsensical cases.

plex and demultiplex  $C$  separate data streams, but a more sophisticated interface may be desired depending on the application.

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 insure a consistent interface, every stream of execution presents all outputs at the same time, with all inputs being registered on the next cycle. If part of the design is  $C$ -slowed, but all operate on the same clock, the resulting design can be retimed as a complete whole while preserving all other semantics. We use these observations later when discussing the effects of  $C$ -slowng on a microprocessor core.

However,  $C$ -slowng imposes some more significant FPGA design constraints, as summarized in Table 2. Register clock enables and resets must be expressed as logic features, since each independent thread must see a different view of the reset or enable. Thus, they can remain features in the design but can't be implemented by current FPGAs using the native enables and resets. Other specialized features, such as Xilinx SRL16s,<sup>4</sup> can't be utilized in a  $C$ -slow design for the same reasons.

One important issue is how to properly  $C$ -slow memory blocks. In most cases, one desires the complete illusion that each stream of execution is completely independent and unchanged. To create this illusion, the memory must be increased by a factor of  $C$ , with additional address lines driven by a counter. This insures 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  $C - 1$  or less. After retiming, the difference in lag is added to the appropriate port's thread counter. This insures that each

<sup>4</sup>A mode where the LUT can act as a 16 bit shift register

FPGA Feature	Effect on Retiming	Effect on $C$ -slowing
Asynchronous Global Set/Reset	Forbidden	Forbidden
Synchronous Global Set/Reset	Effectively Forbidden	Forbidden
Asynchronous Reset	Forbidden	Forbidden
Synchronous Reset	Allowed	Express as logic
Clock Enables	Allowed	Express as logic
Tri-state Buffers	Allowed	Allowed
Memories	Allowed	Allowed, Increase Size
SRL16	Allowed	Express as logic
Multiple Clock Domains	Design Restrictions	Design Restrictions

**Table 2: The effects of various FPGA and Design Features on Retiming and C-slow Retiming**

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 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. Otherwise, there are no other semantic effects imposed by  $C$ -slow retiming.

Figures 3 and 3 demonstrates how  $C$ -slowing and retiming behave on the example used in the earlier illustrations. The act of  $C$ -slowing only increases the number of registers available, without offering any improvement in throughput. Yet as there are now more registers on the feedback loop, retiming is able to reduce the critical path from 5 to 2.

## 4. Retiming FPGAs

Most FPGA designs are especially amenable to  $C$ -slow retiming, as the FPGAs are generally register rich when compared with most designs. Thus increasing the number of registers will not necessarily create a larger design, if the registers are well allocated.

An additional benefit for many architectures is the ability to separately use the logic cells and registers under a wide variety of conditions. This enables registers to be used that would otherwise go completely unused in a design, without impacting the logic utilization.

When retiming for FPGAs, the location of retiming in the toolflow is an important consideration: it can occur before, during, or after the placement process. There are advantages and disadvantages for all three approaches.

Before placement is often the simplest but may encounter some limitations. If the placement tool only places complete cells rather than district elements, the retiming tool can't effectively scavenge unused flip-flops without distorting the placement process. Another disadvantage with pre-placement retiming is the significant factor that interconnect delays play in FPGA retiming. Thus, any retiming tools gains substantial benefits from a detailed understanding of the interconnect delays in a design, informa-

tion which can only be crudely estimated before placement occurs.

Post placement retiming is also convenient as the delays are now all completely known and flip flops can be safely scavenged without affecting the placement process. The main difficulties occur when attempting a large degree of  $C$ -slowing, as allocating large numbers of flip-flops can prove problematic unless placements are modified.

A combined placement and retiming tool is the ideal solution except that it may require a partial or complete restructuring of the placement process. Such a tool can even perform effective retiming when a large number of registers need to be allocated, as it can then permute the design's placement during the allocation process.

## 5. Automatically C-slowng for Virtex

To quantify the effects of  $C$ -slow retiming on real designs, we created a prototype  $C$ -slow retiming tool that operates after placement, targeting the Xilinx Virtex[17] family of FPGAs.<sup>5</sup> This tool is deliberately simplistic, as it is designed simply to assess the costs and benefits of  $C$ -slow retiming when applied with minimal disruption to a commercial FPGA toolflow.

The Xilinx toolflow operates in several stages: Synthesis or schematic capture, translation, mapping, placement, routing, and static timing analysis. During synthesis, the design is converted from an HDL representation to an output netlist. This netlist is translated to Xilinx `.ngd` format for use by the remaining tools use. Mapping takes the logical elements (gates, flip-flops, and other features), combining them into slices. These slices are then placed and routed. Static timing analysis calculates the critical path of the resulting design. Between mapping, placement, routing, and static timing analysis, designs are carried in an `.ncd` binary format. In order to facilitate additional tools, Xilinx defines a text-based `.xd1` format and a translator to convert between `.xd1` and `.ncd`.

By operating on the `.xd1` file format, we are able to modify designs at any point in the mapping, placement, routing, and timing analysis pathway. Significant modifications prevent timing-based backannotation from operat-

<sup>5</sup>This line includes the Xilinx Virtex, Virtex E, Spartan II and Spartan II-E lines.

ing, and designer created preplacement constraints are not included in the `.xdl`, but the other portions of the flow appear unaffected.

Because we both lacked the designer placed constraints in the `.xdl` file, the placement tool only operates on slices, and the lack of good timing information before placement, we decided to perform all our transformations after placement. If we desired pre-placement retiming, the effects of packing flip-flops with unrelated logic would significantly skew the routing process.

This is especially important because the flip-flops in the Virtex slice can be used independently of the logic through the BX and BY inputs under almost all cases.<sup>6</sup> Yet because the placement tool operates on slices, flip-flops that are packed with unrelated logic may severely impact placement quality.

Our modified toolflow, written as a series of perl scripts executing the Xilinx tools, performs synthesis, mapping, and placement, before converting the design to `.xdl`. This `.xdl` file is loaded into a Java program that performs the actual *C*-slowing and retiming.

Our tool first loads the design and insures that it contains no features that would inhibit *C*-slow retiming, such as clock enables, LUTs as RAM or SRL16s, or set/reset connections. Once the design is verified, it is converted to a directed graph that represents the retiming problem. All registers in the design are removed and replaced with edge weights, allowing them to be freely moved. BlockRAMs and I/O pad registers are likewise removed and replaced with mandatory constraints. Any negative-edge clocked memories are changed to operate on a positive clock, with constraints to insure that registers will be placed on the outputs.

Also created are estimates of logic and interconnect delay. Logic delays are based on the published datasheet numbers for LUT evaluation, Block Ram evaluation, carry chain costs, and other additions. Interconnect delays are estimated based on a Manhattan distance metric that abstracts out the different interconnect types. Once this model is complete, the design can then be *C*-slowed by simply multiplying the number of registers on any given edge by *C*. If only retiming is desired, *C* is simply set to 1.

Given this framework, the tool performs classical retiming, with no regard to limiting the number of resulting registers. This first requires a  $O(V^3)$  operation<sup>7</sup> to enumerate all possible critical paths in the design and the number of registers currently along each such path.

The next step involves determining the shortest critical path that the design can be retimed to meet. This phase involves solving the constraint system discussed in Section 2 using the Bellman/Ford shortest path algorithm, an  $O(V^2)$  operation, and using the results to perform a binary search for the smallest feasible critical path.

There are several minor techniques that speed up this process. If a solution exists, the Bellman/Ford algorithm

<sup>6</sup>The only exceptions are when the F5 or F6 muxes are used, when the LUT is used for routing, the LUTs are used as memory devices, or when the carry-in is driven from external logic.

<sup>7</sup>This could be replaced with an  $O(V^2 \lg(V))$  step, but would necessitate changing the representations used

quickly converges, allowing us to perform fewer iterations than are required to guarantee convergence. We can also detect that a solution has been found early, and halt the computation. Carrying over the attempted solution from previous runs also improves performance by providing a mostly correct starting point.

Once a solution is found, the edge weights are converted back into design flip-flops. A first pass allocates any output registers with the appropriate logic. Beyond that, all other registers are allocated by a simple heuristic. A search for an unused register begins at the central point between the source and all destinations and spirals outward. When a register is found, it is then allocated. This process iterates until all registers are instantiated. The resulting design is exported as `.xld` that is then passed through the Xilinx routing and static timing analysis tools.

The retiming process itself requires a few minutes to perform on a Pentium III 550 for the smaller designs due to the  $O(V^3)$  all-pairs shortest path implementation and the numerous Bellman/Ford processing steps. For a larger design, the initial  $V^3$  step is substantial, requiring a few hours to perform for the synthesized SPARC core described in Section 6.3.

For large designs it would be possible to partition the design before retiming. This would cost some precision as the algorithm would be unable to consider delays which cross the partition boundary while reducing running time considerably. We did not perform such an operation as we were interested in correctness, not performance.

Our tool does have some basic limitations. It only *C*-slows the entire design, only works with a single clock domain, and can't automatically increase the size of memories. This last limitation is accommodated by modifying the memories at the design level before proceeding. Yet it is suitable to evaluate the benefits of applying *C*-slow retiming to realistic designs targeting commercial FPGAs.

## 6. Benchmarks

For testing purposes, we used three separate benchmarks which can all benefit from *C*-slow retiming: AES encryption, Smith/Waterman sequence matching, and the LEON 1 synthesized SPARC core. The AES and Smith/Waterman implementations were initially hand placed and *C*-slow retimed, enabling a direct comparison of our tool's performance with those of carefully hand-crafted implementations. LEON 1 is a much larger design, allowing us to evaluate our performance on a substantial design created by conventional HDL synthesis tools.

### 6.1 AES Encryption

The AES encryption algorithm, also known as Rijndael[8], is a block cipher operating on 128-bit quantities with a 128, 192, or 256-bit key. Although generally representative of many block ciphers in design, it is particularly amenable to FPGA implementation as it relies on 8-bit table lookups, byte mixing, and bit manipulations.

Another useful feature is that many usage scenarios greatly benefit from *C*-slowing, as this represents the abil-

ity to encrypt multiple blocks simultaneously, which occurs either when there are multiple streams or the cipher is being used in Counter (CTR) mode[9].

Our initial implementation is a carefully constructed encryption core using 128-bit keys that was hand placed and manually 5-slow retimed, enabling 115 MHz operation on a Spartan 2 100, requiring 10 BlockRAMs and roughly 800 LUTs. We removed the pipelining to create a “pre-retimed” version, and also constructed versions without placement information.

## 6.2 Smith/Waterman Sequence Matching

Smith/Waterman[12] is an important application from computational biology, used to compare the edit distance between two strings. This distance represents how “close” two proteins or DNA sequences may be in terms of functionality or origin, based on a cost model associated with various substitutions, deletions, and insertions.

Smith/Waterman is a dynamic programming problem, requiring  $O(mn)$  time for a sequential implementation. Yet the nature of the dependencies allows a systolic implementation, requiring  $O(m)$  time and  $O(n)$  space. Of further benefit is the common usage: a string is compared to a large database. Thus although the comparison between any single element in the database is a sequential task, there exists abundant parallelism when searching the entire database.

Our implementation consists of a 16-bit edit distance calculation using 5-bit weights and a 5-bit (protein) alphabet. This uses an affine gap scoring, thus creating a break is more expensive than extending an existing break. This is a specialized implementation, where the sequence being compared is compiled into the design, saving a considerable amount of resources.

For testing purposes, we realized 8 systolic cells in a Spartan 2 200. This design is roughly 1700 LUTs, of which most are adders, comparators, and multiplexors.

## 6.3 LEON 1 Microprocessor Core

The Leon 1[4] synthesized microprocessor core is a SPARC-compatible design composed in portable VHDL. Although it may seem unintuitive, a microprocessor design is an excellent candidate for  $C$ -slowing, as each separate task is effectively a separate processor in a multiprocessor system. The resulting design behaves in ways similar to a Symmetric Multithreaded architecture or related designs.

The observation is that each processor in a multiprocessor or multithreaded system is really a separate task, so if one can interleave the computation between distinct tasks, one creates the illusion of 2 or more processors without increasing the logic, just the pipelining. If the memory controller and caches are separate from the processor core, it is straightforward to apply automatic techniques and some design modifications to complete this illusion.

The core itself can be automatically  $C$ -slowed, without any changes, to produce a design capable of operating at a significantly higher clock rate by interleaving the execution of  $C$  separate streams on  $C$  separate virtual processors. Because the entire core’s architectural state is au-

tomatically duplicated, including the control registers, the register file, and the TLB, and because each thread now uses isolated state, this transformation produces an illusion that  $C$ -separate cores exist and present their inputs and outputs in a round-robin fashion through a common interface.

The caches and memory controller need to be modified to account for this round-robin design. Such modifications require architectural changes, including keeping track of which virtual processor requested which datum. A non-blocking cache, at least between the separate streams, is essential to improve performance, and all memory writes and reads need to be atomic to provide synchronization primitives and to prevent problematic interactions between the virtual processors.

The resulting design is a complete  $C$ -thread processor with a programming model identical to other multiprocessor or multithreaded systems:  $C$  separate processors, with their own architectural state and a common memory. Any particular thread runs slower due to the increased pipeline latency but the overall system runs at a significantly higher clock rate, increasing throughput if programs can take advantage of the multiple virtual processors.

A  $C$ -slowed processor’s complete behavior is quite similar to a Simultaneous Multithreaded (SMT)[16] [5] architecture, although it achieves the illusion of separate processors through entirely different implementation strategies.

Similarly, the interleaved computation strategy bears a strong familial resemblance to the early multithreaded machines such as HEP[11] and Tera[1], architectures which removed all bypassing and context switched on every cycle, or the proposed interleaving[6] architectures which maintained some bypassing. Both techniques attempted to hide memory latencies through these context switches. Unlike the other architectures, the  $C$ -slowed strategy takes advantage of the longer dependencies to pipeline the structure more heavily, resulting in a substantially increased system clock.

For FPGA implementations, a 2-slow design is considerably more efficient than 2 distinct processor cores because most processors contain considerably more logic than registers. For direct ASIC implementations, a 2 slow design will still be substantially more efficient as only the number of pipeline registers needs to increase, although not to the same degree seen in an FPGA target.

## 7. Results

In order to evaluate the quality of our results, we ran our retiming tool on three application-based benchmarks discussed earlier, targeting various members of the Xilinx Virtex[18] family of FPGAs. All were placed and routed using Xilinx Foundation 4.1i with maximum effort. All timings are reported by the Xilinx static timing analysis tool.

### 7.1 AES Encryption

For testing purposes, we began with an AES implementation that was hand placed and hand retimed 5-slow to

Version	4-LUTs	LUT Associated Flip Flops	LUT Unassociated Flip Flops	Clock MHz	Stream Clock MHz
None	708		169	48 MHz	48 MHz
5-slow by hand	1012		1229	105 MHz	21 MHz
Retimed automatically	708	150	0	47 MHz	47 MHz
2-slow automatically	708	302	289	64 MHz	32 MHz
3-slow automatically	708	348	668	75 MHz	25 MHz
4-slow automatically	708	447	899	87 MHz	21 MHz
5-slow automatically	708	462	1324	88 MHz	18 MHz

**Table 3: The effects of C-slow retiming the Rijndael encryption core placed with simulated annealing on a Spartan II 100. Our tool produces highly competitive results, able to nearly double the throughput using automated transformations.**

Version	4-LUTs	LUT Associated Flip Flops	LUT Independent Flip Flops	Clock MHz	Stream Clock MHz
None	708		169	48 MHz	48 MHz
5-slow by hand	1012		1229	115 MHz	23 MHz
Retimed automatically	708	132	37	50 MHz	50 MHz
2-slow automatically	708	332	110	74 MHz	37 MHz
3-slow automatically	708	404	515	95 MHz	32 MHz
4-slow automatically	708	660	459	86 MHz	22 MHz
5-slow automatically	708	660	885	105 MHz	21 MHz

**Table 4: The effects of C-slow retiming the hand placed Rijndael encryption core targeting a Spartan II 100. For 5-slow retiming, our throughput more than doubled when compared with the original, unretimed version.**

operate at 115 MHz on a Spartan 2 100, speedgrade 5. Removing the placement information reduces the clock to 105 MHz, while removing both placement and retiming information reduces the clock to 48 MHz. This design requires 10 BlockRAMs and about 800 LUTs. We ran experiments on both the simulated annealing and hand placed versions.

For testing purposes, we started with two modification versions of the above design, one with the pipelining and placement information removed and a second with just pipeline information removed. This allows us to determine how well our tool compares with hand-retimed designs, as well as evaluating absolute speedup over the original design. Table 3 shows the effects of *C*-slow retiming on the design when automated placement is used, while Table 4 shows the benefits for a hand placed design. Lut-associated flip-flops are those which were packed with a LUT output, while LUT independant flip-flops were otherwise unused flip-flops allocated to register the signal. This benchmark has a single cycle feedback loop which defines the critical path, so retiming without *C*-slowing is of effectively no benefit.

These tables measure both the number of flip-flops that are bundled with associated logic and the number that have to be scavenged from unused elements of the FPGA for our automatic tool. In any case, this design is BlockRAM, not LUT limited, so large numbers of registers can be allocated without an area penalty. The clock rate represents the total throughput of the system, while the stream clock is the latency for any single task or stream of exe-

cution. Aggressive *C*-slowing increases overall throughput for multiple execution streams at the cost of single thread latency.

For both cases, although the results are still somewhat inferior to the hand-crafted implementation, they are still highly competitive, more than doubling the throughput for the hand placed, 5-slow version. As expected, retiming without *C*-slowing provides negligible benefit as the critical path is defined by the single cycle feedback loop.

The limitations on our tool for a high *C* factor are due to an inability to alter the logic placement in response to the numerous registers that need to be added and the simple heuristic used to allocate the flip flops. The hand-placed version contains a subkey generation module, expressing roughly 1/3 of the interconnect in 1/5 of the FPGA area. This poses significant difficulties for the tool when allocating large numbers of flip-flops. Additionally, the hand-created implementations used SRL16s to create a compact 128b wide, 3 cycle long delay chain in the subkey generation, a technique not employed by our tool.

There are other factors that affect our quality. The heuristics to calculate interconnect delay are simply linear functions of manhattan distance and the procedure for for allocating flip flops is comparatively crude. Additionally, the *C*-slow retiming process needs to allocate a considerable number of registers because there are 384 bits of registered BlockRAM output that need to be *C*-slowed.

One particular point of interest is the 4-slow case for hand placement. When *C*-slowing, the latency for each

thread is always worse, with the goal that the total throughput increases. We still don't quite understand why the change from 3-slow to 4-slow reduces the throughput, while the transition from 4-slow to 5-slow increases it. We suspect that it involves poor heuristics used to allocate new flip-flops that do not occur in the 5-slow case.

## 7.2 Smith/Waterman

For testing purposes, we began with a carefully hand-tuned specialized cell that was hand mapped, placed and hand retimed 4-slow to operate at 100 MHz on a Spartan II 200. This cell uses a 5 bit alphabet, 5 bit costs, affine gap distance, and 16 bit arithmetic quantities, with the string being matched against compiled into the configuration. Removing the placement information reduces the clock to 90 MHz, while removing both placement and retiming information reduces the clock to 44 MHz. This design requires about 1700 LUTs, of which most are adders.

The results are summarized in Table 5 and Table 6. The slight performance decrease for retiming without *C*-slowing is due to imprecision in our delay modeling that result in some slightly poor decisions. Beyond that, *C*-slow retiming shows the expected performance gains, with not quite double the throughput for the 3-slow, automatically placed version at 84 MHz.

Another observation is that the tool was deliberately conservative: never using an unrelated flip-flop when the carry chain was used, instead of being slightly more selective by monitoring the carry in state that posed a minor limitation due to the large number of adders in this design.

## 7.3 LEON 1

Unlike the other designs, LEON 1[4] is a fully synthesized design and required some minor modifications before it could be tested with our tool. The register file was implemented by hand using negative-edge clocked BlockRAMs to create the illusion that it was an asynchronous memory—needed to match the pipeline structure of the processor. The caches and memory controller were removed because they would need to be reimplemented for a *C*-slowed design to account for multiple data streams and to enable non-blocking behavior between the distinct threads. All other control lines were registered through I/O pads. Synthesis was performed using Synplify with the inferring of clock enables suppressed. The resulting design requires 5900 LUTs, 1580 Flip Flops, and 4 BlockRAMs. It runs at 23 MHz on a Virtex XCV400-5.

However, this was still insufficient for use in our tool as we were unable to prevent the synthesis of hardware resets. Hardware sets and resets were converted to logic which was placed in front of each flip flop by hand-editing the EDIF, creating a design requiring 6100 LUTs.

For this design, the critical path was initially limited by a multicycle feedback loop, enabling conventional retiming to offer a limited benefit. Conventional retiming also eliminated the fragmented cycle from the memory access by converting the BlockRAM to positive edge clocking and forcing the design to rebalance itself, offering a potential

improvement in clock rate at the cost of substantially more registers.

2-slow retiming showed a highly significant performance increase. LEON contains a higher ratio of logic to pipeline registers when compared to the other benchmarks, implying that the latency penalties for retiming would be comparatively lower. This situation the best possible case, where the combined benefits of retiming and *C*-slowing produced a design where single threaded throughput remains constant but aggregate throughput was doubled.

3-slow retiming attempted to allocate too many registers to create an efficient design, resulting in such a marginal performance increase as to be negligible. This limitation suggests that future work on better heuristics to limit flip-flop creating and improve allocation would provide considerable benefits.

## 8. Related Work

To our knowledge, there have been no published commercial or academic *C*-slow retiming tool-flows apart from flow developed by our group for the HSRA[15], a fixed frequency FPGA where designs must be retimed to match the fixed clock rate of the target architecture. The retiming process was accomplished by modifying the design and calling `sis` and having `sis` retime the resulting design.

Conventional retiming, however, is far more common. There have been two previously significant academic retiming tools targeted towards FPGAs. The first, by Cong and Wu[2] combined retiming with technology mapping. This approach enables retiming to occur before placement without adding undue constraints on the placer, since the retimed registers are packed only with their associated logic. The disadvantage is lack of precision, as delays can only be crudely estimated before placement, and it is unsuitable for significant *C*-slowing as this creates significantly more registers.

The second, by Sing and Brown[10], combined retiming with placement. This operated by modifying the placement algorithm to be aware that retiming was occurring and then modifying the retiming portion to enable permutation of the placement as retiming proceeds. They demonstrated how combining placement and retiming performs significantly better than retiming either before or after placement.

The simplified FPGA model used has a logic block where the flip flop can not be used independently of the LUT, constraining the ability of post placement retiming to allocate new registers. Nevertheless, the results still strongly suggest that combined placement and retiming offers significant benefits over pre-placement or post-placement retiming.

Some commercial HDL synthesis tools, notably Synopsys FPGA Compiler[13] and Synplify[14] also support retiming. Because this retiming occurs fairly early in the mapping and optimization processes, it suffers from a lack of precision about placement and routing delays.

C-slow Factor	4-LUTs	LUT Associated Flip Flops	LUT Unassociated Flip Flops	Clock MHz	Stream Clock MHz
None	1784	735		43 MHz	43 MHz
4-slow by hand	2040	2244		90 MHz	22 MHz
Retimed automatically	1784	628	287	40 MHz	40 MHz
2-slow automatically	1784	762	1010	69 MHz	34 MHz
3-slow automatically	1784	811	1723	84 MHz	28 MHz
4-slow automatically	1784	832	2524	76 MHz	25 MHz

**Table 5: The effects of C-slow retiming on the Smith/Waterman Implementation, for simulated annealing placement on a Spartan II 200. The 3 slow version almost doubles the throughput, while the automatic 4 slow version has passed the point of diminishing returns due to the large number of flip flops that required allocation.**

C-slow Factor	4-LUTs	LUT Associated Flip Flops	LUT Unassociated Flip Flops	Clock MHz	Stream Clock MHz
Original	1784	735		45 MHz	45 MHz
4-slow by hand	2040	2244		100 MHz	25 MHz
Retimed automatically	1784	555	321	41 MHz	41 MHz
2-slow automatically	1784	813	972	57 MHz	29 MHz
3-slow automatically	1784	752	1167	86 MHz	28 MHz

**Table 6: The effects of C-slow retiming on the Smith/Waterman Implementation for hand placement on a Spartan II 200. Again, a high C factor nearly doubles the throughput.**

## 9. Summary and Conclusions

C-slow retiming is a very powerful automatic transformation that can be applicable to a wide variety of problems. We reviewed the semantics of retiming and C-slow retiming, and showed how they affect various FPGA design features. Using this foundation, we created a prototype C-slow retiming tool that operates on designs for the Xilinx Virtex family of FPGAs.

This tool was applied to three benchmarks: AES encryption, Smith/Waterman, and the LEON 1 synthesized SPARC core. Of significant interest is that when C-slow retiming is applied to a microprocessor core, the resulting design is a multithreaded core with predictable semantics. Such a core behaves in a very similar manner to an SMT design, although it is constructed in an automatic manner.

All three benchmarks showed significant automatic gains in throughput through the additional pipelining. In some cases, the throughputs are doubled. The ability to double performance through automatic techniques is very powerful, and should be included in conventional flows.

As the C-slowness introduces major semantic changes, it is best performed early in the design process, probably during synthesis. A synthesis tool could C-slow a block, and then rely on later retiming to balance the resulting delays.

Retiming itself is best performed either before, during, or after placement as it benefits greatly from detailed timing information and the ability to scavenge unused flip-flops from elsewhere in the design. Because it requires only minor semantic restrictions, retiming should be included as part of the place and route toolflow. Additionally, retim-

ing could be biased to minimize allocation of flip-flops, a significant benefit when performing C-slow retiming.

## 10. Acknowledgments

Many thanks to Eylon Caspi for his advice on the semantics of retiming, the reviewers for their valuable comments, and Andr e DeHon for many comments about the paper. This work is partially sponsored by Xilinx and the California MICRO program.

## 11. REFERENCES

- [1] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The tera computer system. In *Proceedings of the ACM International Conference on Supercomputing*, June 1990.
- [2] J. Cong and C. Wu. An improved algorithm for performance optimal technology mapping with retiming in LUT-based FPGA design. pages 572–578.
- [3] J. Cong and C. Wu. Optimal FPGA mapping and retiming with efficient initial state computation. In *Design Automation Conference*, pages 330–335, 1998.
- [4] J. Gaisler. Leon sparc-compatible processor.
- [5] Intel. Intel hyper-threading technology, 2001. <http://developer.intel.com/technology/hyperthread/>.
- [6] J. Laudon, A. Gupta, and M. Horowitz. Interleaving: A multithreading technique targeting multiprocessors and workstations. In *Proceedings of*

C-slow Factor	4-LUTs	LUT Associated Flip Flops	LUT Unassociated Flip Flops	Clock MHz	Thread Clock MHz
None	6132	1611		23 MHz	23 MHz
Retimed automatically	6132	2398	194	25 MHz	25 MHz
2-slow automatically	6132	2150	388	46 MHz	23 MHz
3-slow automatically	6132	2438	3713	47 MHz	16 MHz

**Table 7: The effects of C-slow retiming on the LEON 1 SPARC microprocessor core targeting a Virtex 400-5. The 2 slow version doubles the throughput, while maintaining roughly the same latency, while the 3 slow version allocated too many flip flops to be effective.**

- the6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 308–318, 1994.
- [7] C. Leiserson, F. Rose, and J. Saxe. Optimizing synchronous circuitry by retiming. In *Third Caltech Conference On VLSI*, March 1993.
- [8] NIST. Federal information processing standards publication 197: Advanced encryption standard, 2001. <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>.
- [9] NIST. Recommendations for block cypher modes of operation, nist special publication 800-38a, 2001. <http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf>.
- [10] D. P. Singh and S. D. Brown. Integrated retiming and placement for field programmable gate arrays. In *Proceedings of the Tenth ACM International Symposium on Field Programmable Gate Arrays*, 2002.
- [11] B. J. Smith. Architecture and application of the hep multiprocessor computer system. In *SPIE*, pages 298:241–248, 1981.
- [12] T. Smith and M. Waterman. Identification of common molecular subsequences, 1981.
- [13] Synopsys. Synopsys fpga compiler.
- [14] Synplify. Synplify fpga synthesis solution.
- [15] W. Tsu, 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. In *Proceedings of the International Symposium on Field Programmable Gate Arrays*, pages 125–134, February 1999.
- [16] D. M. Tullsen, S. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22th Annual International Symposium on Computer Architecture*, pages 392–403, 1995.
- [17] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. *Virtex Series FPGAs*, 1999.
- [18] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. *Virtex Series FPGAs*, 1999.