Fast Minimum-Register Retiming via Binary Maximum-Flow

Aaron P. Hurst　Alan Mishchenko　Robert Brayton

Department of EECS, University of California, Berkeley

{ahurst, alanmi, brayton}@eecs.berkeley.edu

ABSTRACT

We present a formulation of retiming to minimize the number of registers in a design by iterating a maximum network flow problem. The retiming returned will be the optimum one which involves the minimum amount of register movement. Because all flows are unitary, the problem can be simplified to binary marking. Existing methods solve this problem as an instance of minimum cost network flow, an algorithmically and practically more difficult problem than maximum flow. Our algorithm has a worst-case bound of O(R^2E). We demonstrate on a set of circuits that our formulation is 5x faster than minimum cost-based methods. Delay constraints, which are problematic in the existing methods, are actually a simplifying assumption in our variant, albeit at the loss of optimality in register count.

1. INTRODUCTION

Retiming [13] moves registers over combinational nodes in a logic network, preserving functionality and logic structure. Retiming can target a number of objectives: (i) minimize the delay of the circuit (*min-delay*), (ii) minimize the number of registers under a delay constraint (*constrained min-register*), and (iii) minimize the number of registers (*unconstrained min-register*). Numerous approaches have been proposed to achieve these goals [13]-[18], with most of the emphasis on the first two objectives.

In this paper, we first focus on unconstrained min-register retiming, which has several applications in logic synthesis and verification. In synthesis, minimizing the number of registers can save area and power. Even if the delay constraints are ignored, any timing violations can be corrected with logic sizing, resynthesis, or intentional clock skewing [8]. In verification, min-register retiming minimizes the number of state variables [12], which reduces the size of the sequential verification problem and may be critical for successful completion.

Although retiming problems are traditionally expressed as general linear programs, they can be solved efficiently as minimum cost network circulation problems using suitable algorithms. Instead, we propose a retiming method that is based on iterated binary maximum network flow. This approach can be solved more efficiently than the minimum cost network formulation, because the number of iterations required appears to be quite small. Because the result of each iteration is strictly better than the previous one, the computation can be bounded and still result in an improvement. It was found experimentally that the first iteration of max-flow often accounts for 2/3 of the total gain in the number of registers. This can be used to trade the quality for runtime when a problem is large or fast computation is critical.

To support these claims, we provide experimental results on moderately-sized industrial benchmarks and a few very large artificial ones. They demonstrate the efficiency of the new algorithm: the optimum result can be generated for circuits with more than million gates in less than a minute and much faster than using existing methods. On the benchmark circuits, the reduction in the number of registers ranges from 0% to 60%, averaging about 11%.

An important feature of our algorithm is that it always returns the minimum-register retiming that is closest to the current position of the registers. If a register in the input circuit cannot be retimed to minimize the total register count, it is not touched. This simplifies the computation of the initial states and minimizes the total perturbation.

Delay constraints can be incorporated easily into the new algorithm. Instead of increasing the complexity of the problem, they actually result in a simplification, albeit at the loss of optimality in register count. Even so, the result is still guaranteed to be strictly better than the original.

The paper is organized as follows. Section 2 describes the background information and the existing approaches to minimum-area retiming. Section 3 describes the new algorithm. Section 4 describes how to incorporate delay constraints. Section 5 reports experimental results.

2. BACKGROUND

A circuit is a directed acyclic graph (DAG) $\mathcal{G} = <V,E>$ whose vertices V correspond to logic gates and directed edges E correspond to wires connecting the gates. The terms *network*, *graph*, and *circuit* are used interchangeably in this paper.

A node has zero or more fan-ins, i.e. nodes that are driving this node, and zero or more fan-outs, i.e. nodes driven by this node. The transitive fan-out cone of a vertex v is a subset of all nodes of the network reachable through the fan-out edges from v, captured by the function TFO(v): $V \rightarrow 2^V$.

For the purposes of easily illustrating the concepts in this paper, we employ the register-boundary view of the circuit. An example of this is illustrated in Figure 1. The combinational logic in the circuit is grouped together into a single directed acyclic graph. The inputs to this graph (on the left in Figure 1) consist of the registers and primary inputs. The outputs of the graph (on the right in Figure 1) are the register inputs and primary outputs. The registers are duplicated at the outputs for the ease of illustration, and the wires to the register inputs that form the loops in the sequential circuit are not shown. All references to the combinational network in this paper presume this view.

2.1 Retiming

The problem of retiming is to a find a *retiming lag function* $r(v):V \longrightarrow \mathbb{Z}$ that optimizes some objective while meeting a set of constraints. There are several such formulations, but for the purposes of this paper, we concentrate on the constrained min-register problem from [13] described by the linear program in Equations 1-3. $w_i(e)$ is the initial number of registers present on edge e, $W(u,v)$ is the minimum number of registers along a path $u \rightarrow v$, and $D(u,v)$ is the maximum delay along any path $u \rightarrow v$ with $W(u,v)$ registers. T is the delay constraint.

$$\min \sum_{\forall e=(u,v)} r(v) - r(u) \quad s.t. \tag{1}$$

$$r(u) - r(v) \le w_i(e) \qquad \forall e = (u,v) \tag{2}$$

$$r(u) - r(v) \le W(u,v) - 1 \qquad \forall u,v \text{ if } D(u,v) > T \tag{3}$$

The dual of this linear program is a minimum cost network circulation problem and can be solved efficiently using algorithms specific to this class of problems. Using the method described by Goldberg [9], the minimum cost flow can be computed in $O\left(VE\log(V^2/E)\log(VC)\right)$ worst-case time, where C is the maximum cost on any edge.

The primary source of complexity in this approach lies in the generation and representation of the $W(u,v)$ and $D(u,v)$ values and associated constraints. However, in the unconstrained version of minimum register retiming, there are no constraints of the type of Equation 3, and the problem is greatly simplified. The number of vertices and edges in the corresponding network problem is proportional to the size of the combinational circuit.

3. MIN-REG RETIMING ALGORITHM

We introduce an algorithm for optimum unconstrained minimum-register retiming that is based on an iterative maximum network flow problem. This is motivated by the observation that computing the maximum flow through a network is an algorithmically and practically easier problem than determining the minimum cost circulation. Our algorithm requires repeated iteration, but for practical circuits, the number of iterations is typically quite small.

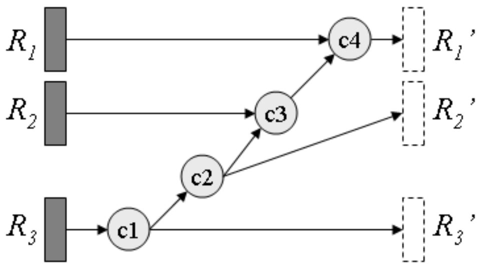

Figure 1. An example of a network where the minimum-cut in the directed graph is not a valid retiming. The combinational network is light grey; the initial positions of the registers lie to the left, and their inputs are replicated on the right. The graph can be completely cut with exactly two registers (at the outputs of *c1* and *c4*), but this results in a path ($R_3 \rightarrow R_1$') with altered sequential latency.

3.1 Single Iteration

A single iteration of the retiming algorithm involves computing the maximum flow through the combinational network, identifying the unique topologically earliest minimum cut, and moving the register boundary to the new location.

Let us consider only the paths through the combinational logic that lie between two registers (thus temporarily ignoring the primary inputs and outputs). In the register-boundary view of this circuit, the registers form a complete cut through the combinational network. Before any retiming is performed, this cut lies at its inputs. The width of the cut is the number of constituent registers.

If the registers are retimed forward over any of the combinational nodes, the corresponding cut moves forward through the network and may grow or shrink in width as registers are replicated and/or shared as needed by the graph structure. In the initial circuit, it is evident that any path in the combined graph passes through exactly one register. Any retiming must preserve this property. If this were not the case, the latency of that path would be altered and the sequential behavior of the circuit changed.

The problem of minimizing the number of registers by retiming them to new positions within the scope of the combinational network is equivalent to finding a minimum width cut. This problem is the dual of the maximum network flow problem, for which efficient solutions exist.

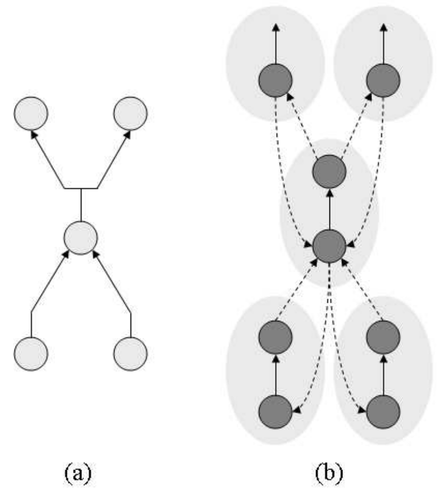

Figure 2. The network in (a) is expanded to form a network flow problem (b) resulting in a valid forward retiming with maximal fan-out sharing. All edges are illustrated for the central node. Each of the light nodes in (a) is replaced with a pair of dark grey nodes in (b) and the flow paths as illustrated. The solid edges are flow paths with unit capacity and the dashed edges are paths with infinite capacity.

Solving the general maximum flow problem using the push-relabel implementation proposed by Cherkassky and Goldberg [2] is $O\left(VE\log(V^2/E)\right)$ in the worst case. Another class of max-flow methods relies upon iteratively searching for augmenting flow paths (e.g. [4]). The runtime of these algorithms is bounded by the maximum length of an augmenting path times the maximum value of the flow, $O(|E|*maxflow)$. Because the flow constraints in our problem are of unit width, the width of the input to the graph establishes a worst-case bound of $O(ER)$, where $|R|$ is the initial number of registers. This is an asymptotically easier problem than computing minimum cost flow on an identical graph structure.

After the maximum flow has been established, the residual graph is used to generate a corresponding minimum cut. Via duality, the width of this minimum cut is exactly the volume of the maximum flow. To determine its location, the vertices in the network are partitioned by their reachability from the flow source in the residual flow graph. Generating this partition is $O(V)$ in the worst case. The partition is a complete cut, because there is no additional flow path from the source to the sink if the maximum has already been reached. The registers are removed from their current locations and placed on the graphs edges that cross the minimum cut.

There may exist multiple cuts of minimum width, but this method always generates the one that is unambiguously closest to the source node. This results in the minimal movement of the registers, simplifying the initial state computation and minimizing the design perturbation.

However, as stated, this procedure may generate an illegal retiming. A minimum cut in a directed graph only guarantees that all paths in the graph are crossed *at least* once. This is a necessary but not sufficient condition for the cut to be a valid retiming. We seek the minimum cut in the graph such that all paths are crossed *exactly* once. Figure 1 illustrates an example of this problem. The network flow problem must be altered to eliminate the possibility that a path is crossed more than once.

Reverse edges with unbounded capacity are added in the direction opposite to the constrained edges in the original network. These additional paths may increase the maximum flow (and therefore the size of the minimum cut) but guarantee that the resulting minimum cut will correspond to a legal retiming. The unlimited reverse flow prevents paths from crossing the resulting cut more than once by disallowing the flow to be constrained by original edges in the reverse direction; no path can cross the cut more than once unless there is some edge that crosses it in the reverse direction, and the cut will not contain such edges unless they constrain the flow.

It is also needed to account for sharing registers along hyper-graph fan-out edges. This requires another simple modification to the network flow problem. Each circuit node is decomposed into two vertices, a *receiver* of all of the former fan-in arcs and an *emitter* of all of the former fan-out arcs. The flow constraints are removed from these edges; instead, a single edge with a unit flow constraint is inserted from the receiver to the emitter. In this scheme, cutting a node will only contribute to the width of the cut once, regardless of its fan-out degree. Note that the reverse edges terminate the receiver.

The final network for computing the maximum flow computation is depicted in Figure 2.

The unitary flow constraints can also be used to simplify the implementation of path-based methods, so that the flow network of Figure 2 need not be explicitly built. Instead, algorithms such as Edmunds-Karp [4] can be implemented entirely with binary marking on the original circuit structure. Because the flows on the reverse edges are unconstrained, they need only be implicitly maintained with a set of flow predecessor pointers.

3.2 Primary Inputs and Outputs

The primary inputs and outputs (PIOs) can be treated in different ways, depending on the application.

In synthesis, the latency at all of the PIOs is assumed to be invariant. This restriction can be enforced in one of two ways: by creating a host node that fixes the retiming lags of all PIOs to be identical; or by excluding portions of the combinational DAG from the minimum cut computation. The first method is straightforward; we describe the second in more detail.

In forward retiming, it is the primary inputs (PIs) that constrain the movement of registers and the location of the

Algorithm 1: Unconstrained Min-Register Retiming

```
while(improvement) {  // forward
  set up forward retiming flow network
  mark restricted locations
  compute maximum flow
  convert to nearest minimum cut
  move registers to cut
}
while(improvement) { // backward
  set up backward retiming flow network
  mark restricted locations
  compute maximum flow
  convert to nearest minimum cut
  move registers to cut
}
compute initial states
```

minimum cut. All paths through the combinational network that originate from a PI have a sequential latency that must remain at *zero*. Inserting a register anywhere in the TFO({PIs}) will alter this. Therefore, to find the minimum cut in the presence of PIs, one of two methods can be used: (i) temporarily redirecting to the sink all edges $e=(u,v)$ where $v \in$ TFO({PIs}), or (ii) replacing the constrained flow arc with an unconstrained one, thus preventing that node from restricting the maximum flow and therefore participating in the minimum cut. Both methods exclude the invalid portion from participating in the retiming solution. Primary outputs are handled similarly during backward retiming.

In verification applications, it is not necessary to preserve the synchronization of the inputs and outputs. It may be desirable to borrow or loan registers to the environment individually for each PIO if the result is a net decrease in the total register count. In this case, the external connections should be left dangling. Registers will be donated to the environment if the minimum cut extends past the dangling terminals; conversely, registers will be borrowed if the minimum cut appears in the transitive fan-in/fan-out region that was excluded above. The inclusion of this region introduces additional flow paths and introduces additional possibilities for minimizing the total register count.

Because register borrowing requires the initial values of the new registers to be constrained to those reachable in the original circuit, it is necessary to construct additional combinational logic for computing the initial state. If the size of this logic grows undesirably large, register borrowing can be turned off.

3.3 Multiple Iterations

This section shows how to compute the globally optimum min-register retiming by iteratively applying the maximum-flow algorithm of Section 3.2.

Thus far, we have only considered forward retiming of registers in the circuit. It is sufficient to consider only one direction if the circuit is entirely cyclic (i.e. if a host node is used to create a loop from the primary outputs to the inputs). However, in general, the optimum minimum-register retiming requires both forward and backward moves. The procedure for a single iteration of backward retiming is nearly identical, except that it computes the maximum flow from the register inputs (sources) to the primary inputs and register outputs (sinks).

The overall algorithm consists of two iterative phases: forward and backward. In each phase, the single frame of iteration is repeated until the number of registers reaches a fix-point. The procedure is outlined in Algorithm 1.

At no point during retiming is it necessary to unroll the circuit or alter the combinational logic; only the register boundary is moved by extracting registers from their initial position and inserting them in the their final position. Therefore, each iteration is fast. In each iteration, every node's lag is either changed by one or unchanged.

The ordering of the two phases (forward and backward) doesn't affect the number of register in the result, but we chose to perform forward retiming first because in general min-register retiming is not unique. This approach reduces the amount of logic that has to be retimed backward, This may lead to a simpler SAT problem when computing a new initial state after retiming. We do not further discuss the details of the initial state computation in this paper.

3.4 Proof of Correctness

Given a retiming lag function $r(v)\colon V \longrightarrow \mathbb{Z}$, consider unrolling the sequential circuit by n cycles, where $n > \max_{\forall V} r(v) - \min_{\forall V} r(v)$. In the latch-boundary view of the circuit, this corresponds to stacking identical copies of the original network, as illustrated in Figure 3.

The positions of the registers of the reference cycle after any retiming $r(v)$ can be expressed as a cut C in the edges of this unrolled circuit. The elements of C are the register positions. The unretimed cut, $C_{init,}$ (such that $r(v)=0$) lies at the base of the unrolled circuit. The size of this cut, $|C|$, is the number registers post-retiming, or equivalently, the number of combinational nodes whose fan-outs hyper-edges cross the cut.

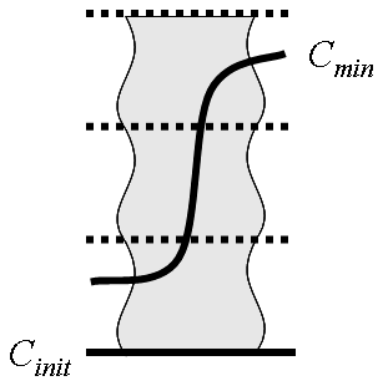

Figure 3. Unrolled Circuit with global minimum-register retiming C_{min}. C_{init} represents the initial position of the registers of the reference frame, and the dashed lines are their equivalent positions at the boundary of each unrolled frame.

A cut C is a *valid retiming* if every path through the combinational network passes through it exactly once. This implies that for any two registers R_1, $R_2 \in C$, $R_1 \cap \mathrm{TFO}(R_2)=\varnothing$ and vice versa. If this were not the case, additional latency would be introduced and functionality of the circuit would be altered.

A *combinational frame* of the cut C with retiming function $r(v)$ is the region in the unrolled circuit between C and C', where C' is generated by $r'(v)=r(v)+1$. If the circuit were retimed to C, this corresponds exactly to the register-free combinational network structure that would lie on the outputs of the register boundary.

Consider an optimal minimum register retiming and its corresponding cut C_{min}. While there exist many such cuts, assume C_{min} to be the one that lies strictly forward of the initial register positions is topologically closest to C_{init}. It can be shown with Lemma 1 that there is one unambiguously closest cut.

Theorem 1: Upon termination of our algorithm, the resulting cut is exactly C_{min}.

Proof: Our algorithm iteratively computes the nearest cut of minimum width reachable within one combinational frame and terminates when there is no change in the result. Let the resulting cut after iteration i be C_i. The cut C_i at termination will be identical to C_{min} if the following two conditions are met.

Condition 1. No register in C_i lies topologically forward of any register in C_{min}.

Condition 2. After each iteration, $|C_{i+1}|<|C_i|$ unless $C_i=C_{min}$.

Lemma 1. Let C_i and C_j be two valid retiming cuts, and $\{s_i, t_i\}$ be $\{s_j, t_j\}$ be a partitioning of each such that $s_i \subseteq TFO(s_j)$ and $t_j \subseteq TFO(t_i)$. The cuts $\{s_i, t_j\}$ and $\{s_j, t_i\}$ are also valid retimings.

First, we should point out that this partitioning is valid and that every element must fall into either s or t as defined. Because C_i and C_j are valid retiming cuts, every path must intersect them both. Given the points of intersection $R_i \in C_i$ and $R_j \in C_j$, their membership in either s or t is implied by their topological order.

Now, consider some path p from the source to the sink of the network. Because C_i and is a complete cut and a valid retiming, the path must pass through exactly one of either s_i or t_i. Similarly for C_j. If p passes through s_i, it can not pass through t_j, because t_j lies in strictly within TFO(t_i), and p would have had to intersect t_j. Also, if p does not pass through s_i, it can not intersect s_j because s_j lies strictly within TFO(s_i) and so must pass through t_j. Therefore, it must pass through exactly one of s_i and t_j, and $\{s_i, t_j\}$ is a valid retiming cut. Similarly for $\{s_j, t_i\}$.

Proof of Condition 1. Consider a cut C_i that violates Condition 1. Let $\{s_i, t_i\}$ be a partition of C_i and $\{s_{min}, t_{min}\}$ be a partition of C_{min} such that s_i is the subset of registers in C_i that lie topologically forward of the subset s_{min} of the registers in C_{min}. This is illustrated in Figure 4. By Lemma 1, we know that both $\{s_i, t_{min}\}$ and $\{s_{min}, t_i\}$ are valid cuts.

Because a single iteration returns the nearest cut of minimum width within a frame, this $C_i=\{s_i, t_i\}$ must be strictly smaller than the closer $\{s_{min},t_i\}$. This implies that $|s_i|<|s_{min}|$ and that $|\{s_i, t_{min}\}|<|\{s_{min},t_{min}\}|=C_{min}$. This is impossible by definition. Therefore, Condition 1 must be true.

Observation 1. Retiming by an entire combinational frame does not change any of the register positions in the resulting circuit and also represents a valid retiming cut. Because a register is moved over every combinational node, the retiming lag function is universally incremented. The number of registers on a particular edge is a relative quantity, the result is structurally identical to the original.

Proof of Condition 2. We can use the minimum cut to generate a cut that is strictly smaller than C_i and reachable within a combinational frame. Consider the cut C_{min}' that is generated from C_{min} via Observation 1 such that its deepest point is reachable within the combinational frame of C_i. Some of the retiming lags may be temporarily negative. Let $\{s_i, t_i\}$ be a partition of C_i and $\{s_{min}, t_{min}\}$ be a partition of C_{min}' such that s_{min} are the deepest registers in C_{min}' that lie topologically orward of the subset s_i of the registers in C_i. $s_{min}\neq\varnothing$ if $C_i \neq C_{min}$. Using the reasoning from condition 1, both $\{s_i, t_{min}\}$ and $\{s_{min}, t_i\}$ are valid cuts.

We know that $|s_{min}|<|s_i|$, otherwise there would be implied the existence of a topologically nearer cut $|\{s_i, t_{min}\}|\leq C_{min}$.

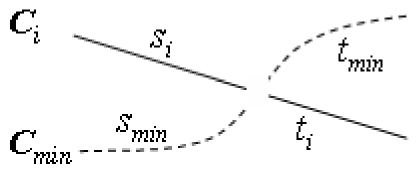

Figure 4. Definitions of cut partitions for Section 3.4.

Therefore, the cut $\{s_{min}, t_i\}$ is strictly smaller than C_i and is reachable within one combinational frame and would be returned by a single iteration of the algorithm. Note that this doesn't imply that there aren't other smaller cuts, only that there must exist at least one that is strictly smaller. Therefore, Condition 2 must also be true.

3.5 Complexity Analysis

As described in Section 3.1, the complexity of computing the minimum cut in each iteration of our algorithm is $O(RE)$. The maximum number of iterations can also be bounded by $|R|$ via Condition 2 in the above proof. The total worst-case runtime is therefore $O(R^2E)$. While this is neither strictly less nor greater than the best known bound for the equivalent minimum-cost network flow problem [9], the results in Section 5 indicate that the average runtimes are smaller for the considered circuits.

4. DELAY-CONSTRAINED MIN-REG

In the formulation of [13], enumerating and incorporating the delay constraints into the minimum cost flow problem dominates the complexity. In the worst case, the number of delay constraints is $O(V^2)$, requiring $O(V^3)$ time to enumerate them via all-pairs shortest paths. This has

Table 1: Minimum Register Retiming Results on Real Benchmarks

	Original Circuit			*Min-Delay Retiming*			*Min-Register Retiming*				
Name	|AIG |	A	D	A	D	T	F-iter	B-iter	A	D	T
barrel16a	397	37	11	124	4	0.02	1	0	32	11	0.00
barrel16	357	37	10	85	4	0.01	1	0	32	11	0.00
barrel32	902	70	12	166	5	0.03	1	0	64	13	0.00
barrel64	2333	135	14	422	5	0.06	1	0	128	14	0.00
mux32_16bit	1851	533	9	873	4	0.05	1	1	505	11	0.01
mux64_16bit	3743	1046	13	1460	5	0.12	1	0	991	13	0.01
mux8_128bit	3717	1155	7	2297	3	0.18	1	1	1029	8	0.00
mux8_64bit	1861	579	7	1145	3	0.07	1	1	517	8	0.00
nut_000	1262	326	58	393	27	0.05	1	2	312	60	0.00
nut_001	3179	484	93	558	57	0.08	2	2	435	109	0.03
nut_002	873	212	24	232	10	0.02	2	2	158	25	0.00
nut_003	1861	265	37	304	24	0.04	3	1	228	46	0.01
nut_004	713	185	13	213	6	0.02	2	2	164	15	0.00
oc_aes_core_inv	11177	669	25	669	25	0.25	1	1	658	25	0.04
oc_aes_core	8732	402	24	402	24	0.14	1	1	394	24	0.00
oc_aquarius	23109	1477	207	1575	200	0.81	1	0	1473	206	0.08
oc_ata_ocidec1	1601	269	14	275	11	0.02	1	0	268	14	0.00
oc_ata_ocidec2	1813	303	14	310	11	0.02	1	1	293	14	0.00
oc_ata_ocidec3	3957	594	14	599	13	0.06	1	1	562	19	0.00
oc_ata_vhd_3	3933	594	14	599	13	0.06	1	1	568	14	0.00
oc_ata_v	838	157	14	169	10	0.02	1	0	156	14	0.00
oc_cfft_1024x12	9498	1051	61	1672	26	0.91	12	1	704	346	0.70
oc_cordic_p2r	8430	719	55	975	45	0.26	1	0	718	55	0.01
oc_dct_slow	879	178	32	207	14	0.03	0	1	176	32	0.00
oc_des_perf_opt	21281	1976	15	4656	14	1.27	15	0	1015	233	1.18
oc_fpu	16115	659	2661	1578	543	30.65	2	0	247	2712	0.12
oc_hdlc	2221	426	14	426	13	0.03	1	3	375	17	0.00
oc_minirisc	1918	289	36	290	33	0.03	2	1	253	39	0.01
oc_oc8051	10315	754	92	757	87	0.19	1	1	743	92	0.01
oc_pci	10426	1354	46	1405	26	0.39	1	1	1308	46	0.02
oc_rtc	1093	114	41	114	29	0.02	1	0	86	41	0.00
oc_sdram	860	112	13	109	12	0.02	1	0	109	12	0.00
oc_simple_fm_rec	2300	226	66	276	40	0.05	0	1	223	75	0.00
oc_vga_lcd	9086	1108	35	1126	25	0.24	2	1	1078	35	0.02
oc_video_dct	36465	3549	60	8525	16	12.84	1	1	2305	73	0.30
oc_video_huff_dec	1591	61	21	65	18	0.02	0	1	60	22	0.00
oc_video_huff_enc	1720	59	19	90	13	0.02	1	0	47	32	0.00
oc_wb_dma	15026	1775	19	1794	17	0.45	1	1	1751	34	0.08
os_blowfish	9806	891	79	906	61	0.30	1	0	827	78	0.00
os_sdram16	1156	147	23	162	17	0.02	1	0	144	23	0.00
radar12	38058	3875	110	3991	56	3.71	2	3	3754	110	0.21
radar20	75149	6001	110	6363	56	6.92	2	1	5364	110	1.34
uoft_raytracer	145960	13079	237	16974	208	23.70	3	2	11610	537	3.76
AVERAGE		**1.0**	**1.0**	**1.41**	**0.66**				**0.89**	**1.56**	

Table 2: Maximum Flow vs. Minimum-Cost Flow on Large Benchmarks

Benchmark			Min-Delay Retiming					
			Min Cost Flow		*Iterative Maximum Flow*			
Name	|Gates||	|Regs|	|Regs|	Runtime	F. Iter.	B. Iter.	Runtime	Speedup
large1	1 006 k	72.9 k	66.9 k	147.9s	3	3	33.0s	4.48
large2	1 005 k	82.7 k	76.9 k	131.3s	3	3	24.5s	5.36
deep3	1 010 k	74.7 k	67.6 k	182.0s	3	21	34.2s	5.32
deep4	1 074 k	86.4 k	82.0 k	130.3s	3	3	17.9s	7.27
larger5	2 003 k	151.1 k	139.5 k	410.6s	3	3	67.2s	6.11
largest6	4 008 k	300.1 k	279.0 k	818.3s	3	3	139.9s	5.85

encouraged attempts to reduce the problem size by eliminating unnecessary delay constraints [14][17], as well as alternative approaches that do not use minimum cost flow [15][16].

Our method lends itself to the incorporation of delay constraints in such a way as to *reduce* the complexity of the network flow problem, albeit at the cost of losing optimality in the number of registers. However, the number of registers in the result is guaranteed to be at least as small as the initial circuit.

The delay constraints must be met by the initial circuit; this can be accomplished by a min-delay retiming. Subsequent violations are prevented by restricting the set of permissible register locations after retiming. By retiming a register forward, the setup condition at its input and/or the hold condition at a register in its transitive fan-out may be violated; retiming a register backward may lead to hold violations at its input or setup violations in its transitive fan-out. The set of potential positions for a register that result in such violations can be pre-computed by assuming that the register or input at the other end of the timing path remains fixed.

So far, the restriction on the possible destinations for a register has only considered the movement of one end of the timing path. In general, the registers at both ends of the path will be relocated, but it can be shown that the above constraints are conservative. The movement of the other timing endpoint (if there is any) will always reduce the criticality of the aforementioned timing constraints. Some of this conservatism can be reduced by ignoring timing paths that end at a register which is also present in the transitive fan-in of a register; these registers were exactly the ones that were retimed forward to the new location, resulting in no net change in the length of these timing paths.

The network structure need not be modified to incorporate the delay constraints. Similar to the technique described in Section 3.3, removing the unit flow constraint from the vertex (and replacing it with an unconstrained edge) will prevent that node from restricting the maximum flow, participating in the minimum cut, and having a register retimed to its output.

5. EXPERIMENTAL RESULTS

We applied the proposed algorithm to a suite of gate-level circuits derived from public-domain hardware designs [11]. Altera tools were used to extract and optimize the logic networks. This optimization may have included retiming. These were then minimally preprocessed by the ABC logic synthesis package [1] as follows: the original hierarchical designs were (a) flattened, (b) structurally hashed and (c) algebraically balanced. From the set of 63 benchmarks, we removed one combinational circuit and 19 circuits whose initial register count was already minimum, leaving 43 circuits shown in Table 1.

Our algorithm was implemented in C++. The maximum network flow problem was internally solved using the HIPR package available at [10] and described in [2].

Table 1 is divided into three groups of columns, each describing the characteristics of a particular retiming. The first section of Table 1 shows the statistics about the circuit with the registers in their initial positions. The second section describes the results of an incremental heuristic min-delay retiming algorithm [18] implemented in ABC to provide perspective on the area/delay tradeoffs. The third set of columns shows the results produced by the proposed min-register retiming algorithm.

The following notation is used in the table. Columns labeled “A” refer to the number of registers in the network (area). Columns labeled “D” refer to the number of nodes on the longest combinational path. Columns labeled “T” refer to the cumulative runtime of the flow computations in seconds measured on a 64-bit 2.0Mhz Pentium Xeon. For the minimum register retiming algorithms, the number of forward and backward iterations that are required before the fix-point is reached are also listed (“F-iter” and “B-iter”, respectively).

Because these benchmarks are only of moderate size, a set of larger artificial circuits was created by combining the benchmarks in Table 1. These are described in Table 2. As the number of retiming iterations required appears to be independent of the circuit size—likely as a result of the independence of the size to the maximum latency around any loop or input to output—the circuits “large1” and “large2” were constructed so that latencies were the maximum of any constituent component. The 2 and 4 million gate circuits, “larger5” and “larger6”, were constructed similarly. In contrast, the two circuits “deep3” and “deep4” were built by concatenating the components to increase the maximum latencies.

In Table 2, the results of our iterative maximum flow-based algorithm are compared against a single minimum cost flow-based implementation as described by [8]. The latest CS2 package from [9] was used as the solver. In every case, the iterative maximum flow-based implementation required less time to complete.

6. CONCLUSIONS

This paper presented an application of a simplified maximum flow computation to the problem of minimizing the number of registers after retiming. The presented method is very simple, straight-forward to implement, fast, memory efficient, and scalable for large industrial circuits. Potential applications of the method include sequential synthesis and verification.

REFERENCES

[1] Berkeley Logic Synthesis and Verification Group, *ABC: A System for Sequential Synthesis and Verification*, Release 61104. http://www.eecs.berkeley.edu/~alanmi/abc/

[2] B. V. Cherkassky and A. Goldberg, "On Implementing Push-Relabel Method for the Maximum Flow Problem," *Algorithmica* 19, 1997, pp. 390-410.

[3] J. Cong and C. Wu, “Optimal FPGA mapping and retiming with efficient initial state computation”, *IEEE Trans. CAD*, vol. 18(11), Nov. 1999, pp. 1595-1607.

[4] J. Edmonds and R. Karp, "Theoretical improvements in algorithmic efficiency for network flow problems", Journal of the ACM 19 (2), 1972, pp. 248-264.

[5] N. Een and N. Sörensson, “An extensible SAT-solver”. *Proc. SAT ‘03.* http://www.cs.chalmers.se/~een/Satzoo/

[6] C. A. J. van Eijk. “Sequential equivalence checking based on structural similarities”, *IEEE Trans. CAD*, vol. 19(7), July 2000, pp. 814-819.

[7] G. Even, I. Y. Spillinger, and L. Stok, “Retiming revisited and reversed”, *IEEE Trans. CAD*, vol. 15(3), March 1996, pp. 348-357.

[8] J. P. Fishburn, “Clock skew optimization”, *IEEE Trans. Comp.*, vol. 39(7), July 1990, pp. 945-951.

[9] A. Goldberg, “An efficient implementation of a scaling minimum-cost flow algorithm”, *J. Algorithms* 22, 1997, pp. 1-29.

[10] A. Goldberg, *Network optimization library.* (Software tools) http://www.avglab.com/andrew/soft.html

[11] M. Hutton and J. Pistorius, *Altera QUIP benchmarks.* http://www.altera.com/education/univ/research/unv-quip.html

[12] A. Kuehlmann and J. Baumgartner, "Transformation-based verification using generalized retiming”, *Proc. CAV’01*.

[13] C. E. Leiserson and J. B. Saxe. “Retiming synchronous circuitry“, *Algorithmica*, 1991, vol. 6, pp. 5-35.

[14] N. Maheshwari and S. Sapatnekar, “Efficient retiming of large circuits”, *IEEE Trans VLSI*, 6(1), March 1998, pp. 74-83.

[15] P. Pan, “Continuous retiming: Algorithms and applications”. *Proc. ICCD ‘97*, pp. 116-121.

[16] S. S. Sapatnekar and R. B. Deokar, “Utilizing the retiming-skew equivalence in a practical algorithms for retiming large circuits”, *IEEE Trans. CAD*, vol. 15(10), Oct.1996, pp. 1237-1248.

[17] N. Shenoy and R. Rudell, “Efficient implementation of retiming”, *Proc. ICCAD ’94*, pp. 226-233.

[18] D.R. Singh, V. Manohararajah, and S.D. Brown, “Incremental retiming for FPGA physical synthesis”, *Proc. DAC ’05*, pp. 433-438.

[19] H. J. Touati and R. K. Brayton, “Computing the initial states of retimed circuits“, *IEEE Trans. CAD*, vol. 12(1), Jan 1993, pp. 157-162.