Fast Minimum-Register Retiming via Binary Maximum-Flow

Alan Mishchenko Aaron Hurst Robert Brayton

Department of EECS, University of California, Berkeley

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

Abstract

The paper introduces a simplified version of the maximum network flow problem with application to minimum-register retiming. The simplifying assumption, which is met by min-register retiming, is that the flow takes only binary values, resulting in an elegant and scalable implementation. Experiments on industrial benchmarks show that the new algorithm is fast and effective; on a network with 100K nodes and 6K registers it took 1 second to solve; an average 10% reduction in the number of registers was achieved on a set of industrial benchmarks.

1 Introduction

Retiming [17] moves registers over combinational nodes in a logic network, preserving functionality and logic structure. Retiming can target a number of objectives: (a) minimize the delay of the circuit (*min-delay*), (b) minimize the number of registers under a delay constraint (*min-area*), and (c) minimize the number of registers (*min-register*). Numerous approaches have been proposed to achieve these goals [10][17][19][20][21][24][25][27], with most of the emphasis on the first two objectives. Objective (b) has the reputation of being the hardest to achieve in practice. Also, retiming has been integrated with logic restructuring performed during technology-independent logic synthesis [2][18][23], technology mapping [22][4], and formal verification [9][16]. These approaches can modify the circuit structure as well as the register positions. This does not reduce the value of stand-alone retiming, since in some approaches, it might be performed repeatedly [9] or as an initial/final step [16][22].

In this paper, we focus on *min-register* retiming, which has several applications in logic synthesis and verification. In synthesis, if delay is not important, the minimum number of registers can save in area and power. Some of the resulting delay degradation can be fixed by clock skewing [11]. In verification, min-register retiming minimizes the number of state variables [16], which may be critical for successful verification based on state enumeration.

Although retiming problems are traditionally translated into linear programming problems [16], it is well recognized that these are of a special network type [25][20][21][22], and can be solved with efficient network methods. Because the min-register problem does not have delay constraints, it is the same as an undirected max-flow problem. For integer flow sources and constraints, it is known that the max-flow solution is integer, making the complexity $O(ME)$ instead of $O(VE)$, where E is the number of edges, V the number of nodes, and M the value of the maximum flow. When the source flows and the edge capacities are unitary, the max flow through any edge or node is binary (all flows are 0 or 1). This binary flow problem has the same worst case complexity as the integer problem, $O(RE)$, where R is the minimum number of sources. However the binary formulation allows for a very simple implementation, which is much faster, uses less memory, and scales to larger networks. This formulation allows the implementation to circumvent more general but less scalable algorithms and software, such as [12][13].

To support these claims, we provide experimental results on large industrial benchmarks. They demonstrate the scalability of the new algorithm, e.g. a typical industrial circuit with 100K nodes and 6K registers takes less than a second to retime. The reduction in the number of registers ranges from 0% to 60%, averaging about 10%.

In the near future, we plan to apply the binary flow algorithm to min-area retiming by minimizing register count first and then greedily trading area for delay, by a combination of skewing some of the registers and incrementally retiming others, using an algorithm similar to [26], hopefully leading to a fast heuristic min-area method.

The rest of the paper is organized as follows. Section 2 describes the background. Section 3 describes the new algorithm. Section 4 reports experimental results. Section 5 concludes the paper and outlines future work.

2 Nomenclature

A *Boolean network* is a directed acyclic graph (DAG) with nodes corresponding to logic gates and directed edges corresponding to wires connecting the gates. The terms network, Boolean network, and circuit are used interchangeably in this paper.

A node has zero or more *fanins,* i.e. nodes that are driving this node, and zero or more *fanouts*, i.e. nodes driven by this node. The *primary inputs* (PIs) of the network are nodes without fanins in the current network. The *primary outputs* (POs) are a subset of nodes of the network. If the network is sequential, the register outputs/inputs are sometimes treated as additional PIs/POs. The PIs and register outputs are cumulatively called *combinational inputs* (CIs) while the POs and register inputs are called *combinational outputs* (COs).

A *fanin* (*fanout*) *cone* of a node n is a subset of all nodes of the network reachable through the fanin (fanout) edges from n. *Area* refers to the number of registers and *delay* refers to the number of logic nodes on the longest path between a CI to a CO.

3 Proposed algorithms

We present the simplified binary maximum flow algorithm and its use for retiming of sequential circuits.

3.1 Retiming as a network flow problem

The traditional formulation of retiming [17] determines the new locations of registers by computing a set of flow-like values called *register lags,* which specify how many register are retimed backward over a node. In the min-delay and min-area problems, additional delay constraints make the search for a set of feasible

register lags difficult; in the min-register problem, their absence leads to a significantly more tractable flow-only problem.

An optimal retiming may require that multiple registers are moved across nodes (i.e. the lags are unbounded integers). Instead, we propose decomposing the retiming into moves within single cycle time frames. In each iteration, either zero or one registers will be moved across a node. This formulation reduces the problem to that of computing *binary maximum-flow*. The resulting minimum cut specifies a new location for the registers between their current location and their location in the next cycle.

The partition induced by the minimum-cut is guaranteed to insert a register along every path within the current frame. This is a necessary but not sufficient condition for the retiming to be valid. A minimum cut of the directed graph may have edges that cross backward over the cut, leading to an implementation with paths that have more than one register within a single cycle.

This is illustrated in Figure 3.1. The flow from the current register positions ro_i to their positions in the next frame ri_i is 2 in the directed graph and the induced min-cut is along the outputs of nodes n_1 and n_4. There is no valid retiming of area 2. However, the flow through the undirected graph is 3, using the reverse edge n_2-n_3, and results in a valid retiming.

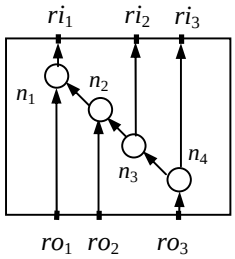

Figure 3.1. Directed versus undirected flow.

This problem can be avoided by computing the flow through the undirected version of the same network. The size of the minimum cut may grow, but it can be proven that one always exists, which has exactly one register along every path: if the min-cut is generated by partitioning the residual graph into nodes that are not reachable from the current register positions, any backward edge must have flow and therefore be forward reachable; by construction, no such edge could exist.

3.2 Binary maximum network flow algorithm

The proposed binary max-flow algorithm is a simplified version of a more general Ford-Fulkerson integer max-flow algorithm [5]. Since a node either has flow or not, in contrast to its integer-valued formulation, the residual and augmenting flows can be recorded by setting a flow-label bit, without the need to support flow counters. We discuss only the forward flow computation because the backward flow is a dual problem.

In the pseudo-code shown in Figure 3.2, the sources of the flow are the register outputs and the sinks are the COs. The reason for this selection will be explained in Section 3.3. A dual selection is associated with backward retiming.

The pseudo-code contains procedures to store and retrieve the flow successor of a node (*nodeSetFlowNext* and *nodeGetFlowNext*). Initially, *computeMaxFlowForward* resets the flow successors. Since the flow of a node is either 0 (there is no flow successor) or 1 (there is a flow successor), we look for an augmenting path originating in each register output only once. Augmenting paths are iteratively added by the recursive routine *computeMaxFlowAugmentPath*.

```
// returns the value of maximum flow from LOs to COs
int computeMaxFlowForward( network )
{
    int flow = 0;

    // clear the flow attributes of all nodes in the network
    for each node in network {
        nodeSetFlowNext( node, NULL );
    }

    // try to find forward augmenting path originating in register outputs
    for each register output node in network {
        clearNodeVisitedMarks( network );
        flow = flow + computeMaxFlowAugmentPath( node );
    }

    // find the min-cut corresponding to the max-flow computed
    min_cut = computeMaxFlowFindMinCut( network );

    return flow;
}


// returns 1 if augmenting path exists; returns 0 otherwise
int computeMaxFlowAugmentPath( node )
{
    // skip nodes visited in this traversal
    if ( nodeIsVisited( node ) )
        return 0;
    nodeMarkAsVisited( node );

    // find the node that brings flow into this node
    prev = computeMaxFlowPredecessor( node );

    // consider a node that currently does not have flow assigned
    if ( nodeGetFlowNext( node ) == NULL ) {

        // if a terminnal node is reached, an augmenting path is found
        if ( nodeIsPO( node ) || nodeIsLI( node ) ) {
            nodeSetFlowNext( node, <terminal> ); return 1;
        }

        // look for an augmenting path through the fanouts
        for each fanout/fanin next of node {
            if ( prev != node && computeMaxFlowAugmentPath( next ) ) {
                nodeSetFlowNext( node, next ); return 1;
            }
        return 0;
    }

    // if there is no fanin with flow, we reached register outputs
    // in this case, no new flow can be added
    if ( prev == NULL )
        return 0;

    // try pushing more flow through other fanouts of the fanin
    for each fanout/fanin next of prev {
        if ( computeMaxFlowAugmentPath( next ) ) {
            nodeSetFlowNext( prev, next ); return 1;
        }
    }

    // try pushing the flow through the predecessor
    // if this can be done,the predecessor's flow will be reset to zero
    if ( computeMaxFlowAugmentPath( prev ) ) {
        nodeSetFlowNext( prev, NULL ); return 1;
    }

    // an augmenting path count not be found
    return 0;
}
```

```
// returns the minimum-volume min-cut corresponding to the max-flow
// (this procedure is called when maximum-flow is assigned
// and there is no augmenting paths)
nodeset computeMaxFlowFindMinCut( network )
{
    // mark all the nodes reachable from register outputs in a flow graph
    for each register output node in network  {
        flow = computeMaxFlowAugmentPath( node );
        assert( flow == 0 ); // no augmenting path can be found
    }
    // collect nodes in the min-cut
    nodeset min_cut = ∅;
    for each node in network {
        // skip nodes without flow or not reachable from register outputs
        if ( !nodeHasFlow( node )  || !nodeIsVisited( node ) )
            continue;
        // collect terminal nodes reachable from register outputs
        if ( nodeIsPO( node ) || nodeIsLI( node ) ) ) {
            min_cut = min_cut ∪ node;
            continue;
        }
        // collect reachable nodes whose fanin with flow is unreachable
        if ( !nodeIsVisited( computeMaxFlowPredecessor(node) ) )
            min_cut = min_cut ∪ node;
    }
    return min_cut;
}

// for a node with flow, returns the fanin, which brings in the flow
node computeMaxFlowPredecessor( node )
{
    for each fanin/fanout next of node {
        if ( nodeGetFlowNext( next ) == node )
            return next;
    }
    return NULL;
}
```

Figure 3.2. Forward maximum flow computation.

An augmenting path is by definition a path from source to sink along which each edge has some remaining capacity. When the selected path hits a node *a* with flow, the existing flow to *a* is pushed back to that predecessor of *a* with flow, say node *b*, and another *unvisited* adjacent (fanout) node of *b* is chosen, from which the augmentation continues recursively. When an augmenting path to a CO is found, the flow labels of the nodes along the path are changed to reflect the modified flow, which has increased by 1. If an augmenting flow from a register can't be found, it will never be found even if the register is revisited later. Thus when the last register has been processed, the maximum flow has been found. Finally, a minimum cut (that is, the cut with the smallest number of nodes) is computed from this maximum flow by procedure *computeMaxFlowMinCut*. In general, the min-cut is not unique. We choose the min-cut with the smallest volume since the min-register retiming, based on this cut, moves registers the least distance.

3.3 Minimum-register retiming algorithm

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

The min-area retiming is performed in two steps: forward and backward, as shown in Figure 3.3. The only difference between these two retiming steps is that forward one requires computing maximum flow from the register outputs (sources) to the COs (sinks), while the backward one requires computing the maximum flow from the register inputs (sources) to the CIs (sinks).

```
// performs minimum-area retiming using maximum-flow computation
retimeMinRegister( network )
{
    // iterate forward retiming as long as there is improvement
    while ( computeMaxFlowForward(network) < registerCount(network) )
        retimeMoveRegistersToMinCut( network );
    // iterate backward retiming as long as there is improvement
    while ( retimeMaxFlowBackward(network) < registerCount(network) )
        retimeMoveRegistersToMinCut( network );
    // compute new initial state
    retimeComputeInitialState( network );
}
```

Figure 3.3. Implementation of min-register retiming.

Both the forward and backward parts of the retiming are performed on a single time frame of the network. However, general retiming may move registers across a node more than once; therefore the computation based on one time frame is iterated in Figure 3.3.[1]

It should be noted that backward retiming followed by forward retiming will also result in a solution with minimum area, but we chose to perform forward retiming first because min-register retiming in general is not unique. This reduces the amount of logic that has to be retimed backward, and although not discussed in this paper, this may lead to a simpler SAT problem when computing a new initial state after retiming.

Only forward retiming is discussed below, as backward retiming is its dual. The sources are the register outputs; the sinks are COs. The reason for this selection is that in the forward retiming, the min-cut lies between the current register positions and the COs. In particular, a register may travel from its current position all the way to an input of another register or to a PO and get stuck there, waiting for the next time frame to proceed further.

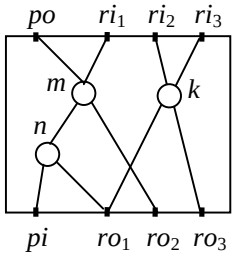

Figure 3.4. Illustration of retiming in the presence of PIs.

The PIs constrain the forward movement of registers and the location of the minimum cut. Consider a circuit with one PI, one PO, and three registers shown in Figure 3.4. Internal node *n* is fed by a register and a PI. This node cannot be retimed forward because the PI does not have a matching register. Therefore, to find the max-flow in the presence of PIs, we modify the sinks to be the nodes in the TFO of the PIs plus the COs (previously, the sinks were just the COs).

For the example shown in Figure 3.4, nodes *n* and *m* become sinks in addition to the register inputs ri_1, ri_2, and ri_3. The max-flow computed without taking PIs into account is 2 (the corresponding min-cut is {*m, k*}). When PIs are present, the max-

[1] A similar effect could be achieved by unrolling the circuit several times and computing the min-cut once using the unrolled time frames.

flow is 3, which is in correspondence with the min-register retiming using three registers. It should be noted that, in our computations, we do not add a host node and retime over it, as done implicitly in [7], since the same result can be achieved by forward and backward retiming on the circuit.

4 Experimental results

The algorithm presented in this paper was implemented in the logic synthesis and verification system ABC [1] as a new command *retime*. The current implementation can retime both SIS-like logic networks and AIGs assuming a unit-delay model (all internal logic nodes have delay 1).

The correct functionality of networks after retiming has been verified by the bounded sequential equivalence checker in ABC (command *sec*). To enable verification by comparing sequential behavior of the original and the final circuits, starting from two equivalent initial states, the circuits were preprocessed as follows. All initial values of the registers in the original circuits were set to zero and the circuit was cycled with random PI values for a fixed number of clock cycles to arrive at an initial state, for the un-retimed circuit, for which an equivalent initial state exists after retiming.[2] A corresponding equivalent initial state for the retimed circuit was computed using a SAT solver [8]. The runtime of this computation was negligible, compared to that of retiming. The bounded sequential equivalence checker then verified that the two states were equivalent up to a specified number of clock cycles.

The following notation is used in the tables below. 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 runtime in seconds measured on an IBM ThinkPad laptop with a 1.6GHz Intel CPU with 2Gb of RAM.

Two experiments were performed and are reported in the following sub-sections.

4.1 Comparison with previous retiming solutions

We compare the performance of the min-register retiming against several previous efficient retiming solutions: (a) min-delay retiming using retiming/skew equivalence implemented in ASTRA [24], (b) min-area retiming under delay constraints implemented in Minaret [20], (c) continuous min-delay retiming, called *c-retiming* [21], and (d) a heuristic incremental min-delay retiming used in an industrial setting [26].

The last algorithm was implemented by us in ABC and run on the same computer as the presented algorithm. The results for the first three algorithms are quoted from publications [24][20][21]. The benchmarks selected for this experiment were that subset of the ISCAS'89 benchmarks, for which the same files were found as used in [24][20][21]. They were judged the same by their numbers of gates reported in these references. This ensures that all retiming algorithms were applied to the same circuit structures.

Table 1 lists the benchmark names, followed by the original circuit statistics: the number of gates in the network, and the initial area/delay, and then the results of the five retiming algorithms in terms of area, delay, and runtime.

The experimental results show that the proposed algorithm for min-register retiming finds retimings with the smallest area. This is because the min-delay algorithms, such as ASTRA and c-retiming, do not constrain area while the only other area-oriented retiming method [20] works under a minimum-delay constraint. Runtime comparisons with ASTRA and Minaret are not valid without factoring in the speed of the older computers used in those papers. However, comparison with c-retiming and the incremental method indicate that the new method is very fast.

4.2 Performance on large benchmarks

We applied the proposed algorithm to a suite of gate-level circuits derived from public-domain hardware designs. Altera tools [14] were used to extract the logic networks. These were then minimally preprocessed by ABC as follows: the original hierarchical designs were (a) flattened, (b) structurally hashed and (c) algebraically balanced. The original benchmarks in BLIF and those preprocessed by ABC can be found on the web [28]. Out of the set of 63 benchmarks, we removed one combinational circuit (no registers) and 19 circuits whose initial register count was already minimum, leaving 43 circuits shown in Table 2.

The first section of Table 2 shows the gate ("Gates"), register ("A"), and delay ("D") counts. The next section shows the results produced by the proposed min-register retiming algorithm. To put these results in perspective, they are compared with the incremental heuristic min-delay retiming algorithm [26] implemented in ABC. The number of iterations was set to twice the critical delay of the original circuit instead of the fixed value (32) suggested in [26]. The last column (Pan's) shows the delay of the exact min-delay retiming derived by computing sequential arrival times [21][22].

The results confirm that the new min-register retiming algorithm is very fast; it takes only a few seconds for even the largest benchmarks. The average reduction in the number of registers is 10% while some benchmarks are reduced more than 60%.

4.3 Heuristic min-area method

The last experiment (to be conducted in the final version of the paper) will attempt to put together a heuristic min-area algorithm by combining the proposed min-register method with a type of incremental algorithm, as suggested in [26]. The idea is to start out with as few registers as possible and shift them only as little as possible to reach a desired delay.

5 Conclusions and future work

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.

Future work will include refining fast incremental retiming algorithm for delay and combining it with the proposed min-area retiming algorithm and clock skewing to achieve good delay/area trade-offs. This should open new opportunities for applying retiming in delay-driven optimization flows without excessive area penalties.

Acknowledgment

This research was supported in part by SRC contracts 1361.001 and 1444.001, by the C2S2 Focus Research center under contract 2003-CT-888, and by the California Micro program with industrial sponsors, Altera, Intel, Magma, and Synplicity.

[2] Since retiming preserves the cyclic core of a design, any state in the cyclic core of the initial design has an equivalent initial state of the retimed circuit.

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] S. Bommu, N. O'Neill, and M. Ciesielski. Retiming-based factorization for sequential logic optimization, *ACM TODAES*, vol. 5(3), July 2000, pp. 373-398.

[3] R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli, "Multilevel logic synthesis", *Proc. IEEE*, vol. 78(2), February 1990, pp. 264-300.

[4] 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.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to algorithms*, New York: McGraw-Hill, 1990.

[6] A. Dasdan, "Experimental analysis of the fastest optimum cycle ratio and mean algorithms", *ACM TODAES '04*, vol. 9(4), pp. 385-418.

[7] G. De Micheli, "Synchronous logic synthesis: Algorithms for cycle time minimization", *IEEE Trans. CAD*, vol. 10(1), January 1991, pp. 63-73.

[8] N. Een and N. Sörensson, "An extensible SAT-solver". *Proc. SAT '03*. http://www.cs.chalmers.se/~een/Satzoo/

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

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

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

[12] A. Goldberg, "An efficient implementation of a scaling minimum-cost flow algorithm", *Technical Report STAN-CS-92-1439*, Stanford University, 1992. http://ftp.cs.stanford.edu/cs/theory/goldberg/

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

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

[15] IWLS 2005 Benchmarks. http://iwls.org/iwls2005/benchmarks.html

[16] A. Kuehlmann and J. Baumgartner, "Transformation-based verification using generalized retiming", *Proc. CAV'01*, pp.

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

[18] B. Lin, "Restructuring of synchronous logic circuits", *Proc. Euro-DAC '93*, pp. 205-209.

[19] S. Malik, E. M. Sentovich, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Retiming and resynthesis: Optimizing sequential networks with combinational techniques", *IEEE Trans. CAD*, vol. 10(1), Jan 1991, pp. 74-84.

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

[21] P. Pan, "Continuous retiming: Algorithms and applications". *Proc. ICCD '97*, pp. 116-121.

[22] P. Pan and C.-C. Lin, "A new retiming-based technology mapping algorithm for LUT-based FPGAs," *Proc. FPGA '98*, pp. 35-42.

[23] P. Pan, "Performance-driven integration of retiming and resynthesis", *Proc. DAC '99*, pp. 243-246.

[24] 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.

[25] N. Shenoy and R. Rudell, "Efficient implementation of retiming", *Proc. ICCAD '94*, pp. 226-233.

[26] D.R. Singh, V. Manohararajah, and S.D. Brown, "Incremental retiming for FPGA physical synthesis", *Proc. DAC '05*, pp. 433-438.

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

[28] http://www.eecs.berkeley. edu/~alanmi /benchmarks/altera

Table 1. Comparison of the new algorithm with the previous work.

Bench	Original statistics			ASTRA [24]			Minaret [20]		
mark	Gates	A	D	A	D	T	A	D	T
s3271	1572	116	28	306	15	1.6	168	15	0.25
s3384	1685	183	60	438	27	15.5	167	27	2.44
s3330	1789	132	29	331	14	2.6	110	14	0.22
s4863	2342	104	58	201	30	1.5	138	30	5.24
s5378	2779	179	25	555	21	8.4	173	21	1.28
s6669	3080	239	93	719	29	49.3	305	29	2.20
s35932	16065	1728	29	1729	27	23.1	1729	27	7.56
Ratio		**1.00**	**1.00**	**2.37**	**0.58**		**1.11**	**0.58**	

Bench	C-retiming [21]			Incremental [26]			New min-area		
Mark	A	D	T	A	D	T	A	D	T
s3271	198	15	1.99	238	16	0.05	116	28	0.00
s3384	207	27	2.61	208	27	0.04	153	73	0.02
s3330	218	14	1.86	109	17	0.03	66	24	0.01
s4863	183	30	3.35	208	30	0.07	88	58	0.01
s5378	189	21	3.00	192	21	0.03	132	26	0.02
s6669	355	26	8.81	551	28	0.15	183	121	0.02
s35932	1729	27	40.02	1729	27	0.72	1728	29	0.06
Ratio	**1.40**	**0.58**		**1.48**	**0.60**		**0.81**	**1.06**	

Table 2. Performance of the new algorithm on industrial benchmarks.

Benchmark	Original			Min-register			Min-delay [26]			Pan's
	|AIG|	A	D	A	D	T	A	D	T	D
barrel16a	397	37	11	32	11	0.00	124	4	0.02	4
barrel16	357	37	10	32	11	0.00	85	4	0.01	4
barrel32	902	70	12	64	13	0.00	166	5	0.03	5
barrel64	2333	135	14	128	14	0.01	422	5	0.06	5
mux32_16bit	1851	533	9	505	11	0.02	873	4	0.05	4
mux64_16bit	3743	1046	13	991	13	0.01	1460	5	0.12	5
mux8_128bit	3717	1155	7	1029	8	0.07	2297	3	0.18	3
mux8_64bit	1861	579	7	517	8	0.03	1145	3	0.07	3
nut_000	1262	326	58	318	60	0.00	393	27	0.05	27
nut_001	3179	484	93	449	109	0.01	558	57	0.08	46
nut_002	873	212	24	158	25	0.01	232	10	0.02	10
nut_003	1861	265	37	238	46	0.01	304	24	0.04	24
nut_004	713	185	13	170	15	0.00	213	6	0.02	6
oc_aes_core_inv	11177	669	25	658	25	0.05	669	25	0.25	25
oc_aes_core	8732	402	24	394	24	0.02	402	24	0.14	24
oc_aquarius	23109	1477	207	1473	206	0.19	1575	200	0.81	200
oc_ata_ocidec1	1601	269	14	268	14	0.00	275	11	0.02	11
oc_ata_ocidec2	1813	303	14	299	14	0.02	310	11	0.02	11
oc_ata_ocidec3	3957	594	14	581	19	0.03	599	13	0.06	13
oc_ata_vhd_3	3933	594	14	589	14	0.01	599	13	0.06	13
oc_ata_v	838	157	14	156	14	0.00	169	10	0.02	10
oc_cfft_1024x12	9498	1051	61	712	346	0.14	1672	26	0.91	20
oc_cordic_p2r	8430	719	55	718	55	0.02	975	45	0.26	39
oc_dct_slow	879	178	32	176	32	0.01	207	14	0.03	14
oc_des_perf_opt	21281	1976	15	1088	233	1.08	4656	14	1.27	13
oc_fpu	16115	659	2661	247	2712	0.07	1578	543	30.65	543
oc_hdlc	2221	426	14	383	17	0.02	426	13	0.03	13
oc_minirisc	1918	289	36	278	39	0.01	290	33	0.03	33
oc_oc8051	10315	754	92	752	92	0.04	757	87	0.19	87
oc_pci	10426	1354	46	1326	46	0.07	1405	26	0.39	26
oc_rtc	1093	114	41	86	41	0.01	114	29	0.02	29
oc_sdram	860	112	13	109	12	0.00	109	12	0.02	12
oc_simple_fm_rec	2300	226	66	223	75	0.01	276	40	0.05	40
oc_vga_lcd	9086	1108	35	1091	35	0.05	1126	25	0.24	25
oc_video_dct	36465	3549	60	2305	73	0.72	8525	16	12.84	16
oc_video_huff_dec	1591	61	21	60	22	0.01	65	18	0.02	18
oc_video_huff_enc	1720	59	19	47	32	0.01	90	13	0.02	13
oc_wb_dma	15026	1775	19	1767	34	0.12	1794	17	0.45	17
os_blowfish	9806	891	79	827	78	0.03	906	61	0.30	42
os_sdram16	1156	147	23	144	23	0.00	162	17	0.02	17
radar12	38058	3875	110	3754	110	0.37	3991	56	3.71	56
radar20	75149	6001	110	5364	110	1.15	6363	56	6.92	56
uoft_raytracer	145960	13079	237	11645	537	6.46	16974	208	23.70	202
Ratio	1.00	1.00	1.00	0.90	1.56	1.00	1.41	0.66	1.00	0.64