Maximum Throughput Logic Synthesis for Stateful Logic: A Case Study

D.B. Strukov[1*], A. Mishchenko[2], and R. Brayton[2]

[1]*Department of ECE, University of California at Santa Barbara, Santa Barbara, CA, 93106 USA*
[2] *EECS Department, UC Berkeley, Berkeley, CA, 94720 USA*
**Email: strukov@ece.ucsb.edu*

ABSTRACT

This paper investigates the design of deeply pipelined synchronous circuits with stateful logic, which can be implemented with several emerging device technologies. Stateful logic is well suited for high-throughput processing because each gate combines logic and memory. The downside of the high throughput computation is that all logic paths must have equal length. Using advanced logic synthesis techniques buffering needed to balance the path length is shown to result in an average overhead of 3x in gate count when mapped into several target libraries. The area overhead of stateful logic may be still justified given an average 7.5x improvement in throughput per gate.

Keywords

Logic synthesis, Logic optimization, Stateful logic, Material implication logic, NDR-based logic.

1. INTRODUCTION

Several prospective device technologies with combined logic and memory (i.e. flip-flop) functionality are being actively investigated. These include various spin-state-variable logic concepts [1, 2], material implication logic [3, 4], CMOS/memristor hybrids [5, 6], and quantum-dot cellular automata circuits [7]. The main promise of such "stateful" logic is very inexpensive implementation of memory and its tight integration with logic thus giving hope for implementing energy efficient high-throughput computation.

The downside of the high throughput computation is that all logic paths must have equal length. Actually, having balanced paths would be a requirement for some of these emerging device technologies with no efficient way of implementing gates without a state, so that practical (uniform) implementation of the circuits assumes that all the gates in the circuits are stateful. Because of that, the benefit of having a flip-flop in each gate also comes with a requirement of trading delay for a maximum throughput. Figure 1 explains the problem in detail. If all gates in the circuits are stateful (e.g. gates with clock input), then some buffers should be inserted for a correct implementation of the original circuit shown on Figure 1a.

Because of this, stateful logic is a good fit for computation which maps well on systolic, nearest-neighbor interconnect structures [8], such as computer arithmetic (e.g. parallel prefix adders and array multipliers), image, signal, and network processing (e.g., Fast Fourier transform, pattern matching, and various filters). The implication of balancing for arbitrary circuits, though, is less obvious.

The main goal of this paper is to investigate gate overhead associated with balancing. We further explore logic synthesis techniques to alleviate such overhead and discuss the effect of buffering on the throughput per area metric. To the best of our knowledge, this is the first attempt at such study. The only related work [9] known to us focuses on logic synthesis for material implication logic and therefore is less general.

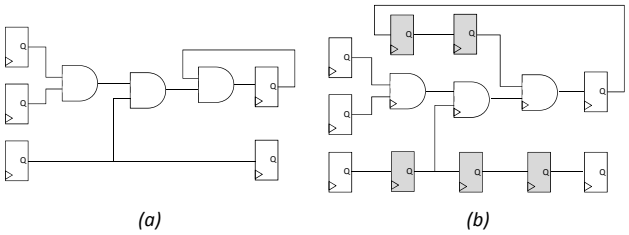

Fig. 1. Delay balancing for stateful logic: (a) original and (b) balanced circuits with added buffers (shaded grey).

2. BACKGROUND

A *Boolean network* is a directed acyclic graph (DAG) with nodes corresponding to logic gates and directed edges corresponding to wires connecting the gates.

A node *n* has zero or more *fanins*, i.e. nodes that are driving *n*, and zero or more *fanouts*, i.e. nodes driven by *n*. The *primary inputs* (PIs) are nodes without fanins in the current network. The *primary outputs* (POs) are a subset of nodes of the network.

A network is *K-bounded* if the number of fanins of any node does not exceed *K*. A *subject graph* is a *K*-bounded network used for technology mapping.

A *cut C* of node *n*, called *root*, is a set of nodes of the network, called *leaves*, such that each path from a PI to *n* passes through at least one leaf. A *trivial cut* of node *n* is the cut {*n*} composed of the node itself. A non-trivial cut *covers* all the nodes found on the paths from the root to the leaves, including the root and excluding the leaves. A trivial cut does not cover any nodes. A cut is *K-feasible* if the number of nodes in it does not exceed *K*.

A *fanin* (*fanout*) *cone* of node *n* is a subset of all nodes of the network reachable through the fanin (fanout) edges from node *n*.

The *level* of a node is the length of the longest path from any PI to the node. The node itself is counted towards the path lengths but the PIs are not. The network *depth* is the largest level of an internal node in the network. The depth and area of FPGA mapping is measured by the depth of the resulting LUT network and the number of LUTs in it.

An *And-Inverter graph* (AIG), composed of two-input ANDs and inverters represented as attributes on the edges. In the rest of the paper, the subject graph is assumed to be an AIG. This does not limit the generality of the presentation because any combinational network can be expressed as an AIG. Furthermore, the result of many AIG-based computations, in particular, technology mapping, can be back-annotated to the original network. This allows us to use the efficient software implementation of AIGs for solving various problems formulated for Boolean networks.

Technology mapping is applied to an AIG with the goal of expressing logic functions represented by the AIG using in terms of technology primitives. Each primitive is associated with a *K*-feasible cut. A *mapping* assigns one *K*-feasible cut, called *representative cut*, to each non-PI node of the subject graph. The mapping procedure computes a subset of nodes whose representative cuts cover all non-PI nodes. The nodes from this subset are *used* in the mapping.

In this paper, a starting mapping is found by assigning one "good" cut at each node in the graph. Next, the mapping is updated by modifying the representative cut of one node at a time. Each such modification may change the set of used nodes. These changes may propagate recursively from the node towards the PIs.

When unit area (delay) model is used, the *area* (*delay*) of the mapping is the number of nodes (logic levels) used by the mapping. Heuristic area recovery discussed in this paper is greedy in the sense that it modifies the representative cuts of the nodes, one at a time, in such a way that the area of the current mapping is reduced or remains the same.

3. MAPPING ALGORITHM

The primitives used for technology mapping in this work are multi-input (N)AND/(N)OR gates. Area of each gate is assumed to be unit and delay is assumed to be $\log_2(2n)$, where n ($1 \le n \le N$) is the gate fanin count and N is the maximum gate size allowed by the technology. Such assumptions, for example, are representative of CMOS/memristor hybrids [5, 6].

In many technologies, delay of the design is dominated by the delay of the wires. The intrinsic gate delay depends on the number of fanouts of a gate. However, this dependence shows only when the fanin count of a gate is relatively large (say, 10 or more). Since the networks considered in this paper have relatively few gates with large fanin count, node delay is assumed to be fanout-independent.

The pseudo-code on Figure 2 explains the algorithm used to map the subject graph represented as an AIG into (N)AND/N(OR) gates. The mapping is performed in two phases. First, each AIG node of the subject graph is annotated with the cut whose Sum-of-Products (SOP) representation has a good delay/area trade-off computed heuristically. Second, the final network is derived by converting into (N)AND/N(OR) gates the products and the sum of each SOP used in the mapping. The mapping procedure is implemented by parametrizing the priority-cut-based mapper [11].

4. BUFFERING ALGORITHM

To satisfy the requirements of the technologies presented in this paper, the network mapped into (N)AND/N(OR) gates is post-processed by inserting buffers. The goal of this buffering is to make sure that all combinational paths from PIs to POs have equal length.

Below we first present a simple buffering algorithm, which produces suboptimal results. Next, we present an improvement of this algorithm, which iteratively refines the result by removing redundant buffers.

The simple algorithm shown in Figure 3 begins by assigning logic level to the nodes. Next, it considers each node and determines the node's fanout with the highest logic level. If this fanout is one level above the node, there is no need to add buffers. Otherwise, for each level between the node and its highest fanout, a buffer is inserted. This buffer is connected to the node or to the previous buffer, depending on the level. Fanouts of the node are redirected to point to the buffers one level below their level. As a result, all wires between the nodes and their fanouts path through the intermediate buffers. The resulting network is balances because each wire spans exactly one level.

```
mapped_network DeriveMapping(
    aig A,   // A is an And-Inverter Graph
    int N )   // N is the maximum number of fanins allowed
{
  // perform mapping to find a good delay/area tradeoff
  iterate several times while there is an improvement {
    for each node n of A in a topological order {
      compute several N-feasible cuts of n;
      for each cut c {
        compute Boolean function of the cut;
        transform the function into an irredundant SOP;
        delay = log2(2*size(c));
        if ( the SOP has only one cube )
          area = 1;
        else
          area = 1 + number of cubes in the SOP;
        if ( delay and area of c are better than those of best cut of n)
          set best cut of n to be c;
      }
    }
  }
  // generate the network of (N)AND/N(OR) gates
    initialize G, a new logic network after mapping;
  for each node n in A, in a topological order {
    if ( n is used in the mapping) {
      retrieve the best cut of n;
      compute Boolean function of the cut;
      transform the function into the irredundant SOP;
      generate ANDs/ORs of the products/sum of the SOP;
      add these to G as gates;
    }
  }
  return G;
}
```

Fig. 2. Mapping into (N)AND/N(OR) gates.

```
buffered_network SimpleBuffering(
    mapped_network G ) // G is a mapped network
{
    assign logic level to all nodes in a topological order;
    for each node n of the mapped network {
      let m be a fanout of n with the highest level;
      if ( level(m) == level(n) + 1 )
          continue;
```

```
        assert( level(m) > level(n) + 1 );
        for ( lev = level(n) + 1; lev < level(m) ; lev = lev + 1) {
            create buffer b on level lev;
            if ( lev == level(n) + 1 )
                connect b to n;
            else
                connect b to the previous buffer on level lev – 1;
            if ( node n has fanout on level lev + 1 )
                update their fanins to point to b instead of n;
        }
    }
    return G;
}
```

Fig. 3. Simple buffering algorithm.

This algorithm may create redundant buffers. For example, consider levelized network in Figure 4(a). Wires between some nodes of this networks span more than one logic level. The result of simple buffering is shown in Figure 4(b). It can be observed that the middle node can be moved one level up to save one buffer. The result of this move is shown in Figure 4(c). In other cases, moving a node down can also save buffers.

An improved buffering procedure, which incrementally moves nodes up or down to reduce the total number of inserted buffers, is shown in Figure 5.

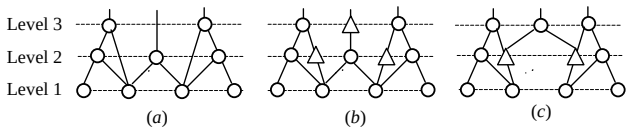

Fig. 4. Illustration of buffering.

```
buffered_network IterativelyRefineBuffering(
    buffered_network G ) // G is a buffered network
{
    iterate as long as there are changes in G {
        // try to move nodes up
        for each node n in the buffered network in some order {
            if ( there is fanout m such that level(m) == level(n) + 1 )
                continue;
            if ( each fanin of n has a fanout higher than n )
                move n up and save one buffer;
        }
        iterate as long as there are changes in G {
            // try to move nodes down
            for each node n in the buffered network in some order {
                if ( there is fanin m such that level(m) == level(n) - 1 )
                    continue;
                if ( fanins of n have other fanouts with level of n or higher )
                    move node n down and save one or more buffers;
            }
        }
        return G;
}
```

Fig. 5. Procedure for iterative refinement of buffering.

The algorithm iteratively tries to move nodes, first up, then down, as long as the number of buffers is reduced. Consider the first part of the procedure, which attempts to move node *n* up. If there exists fanout *m* on the logic level adjacent to *n*, the upward move of *n* is impossible. Otherwise, a buffer is saved if each fanin of *n* has a fanout higher than *n*. This condition is true for the middle node in Figure 4 (b). Both of its fanins have fanouts one level higher than the node. In this case, the middle node is moved up one level as shown in Figure 4 (c) and one buffer is saved.

The dual case when a node can be moved down to save one or more buffers is shown at the bottom of Figure 5.

5. EXPERIMENTAL RESULTS

The proposed algorithms are implemented in ABC [10] as a parametrized version of priority-cut-based mapper *if* [11]. Command *if –G <num>* maps a given AIG into logic nodes whose SOPs have minimal cumulative cost. Command *cubes* performs decomposition of the resulting mapped nodes into individual (N)AND/N(OR) gates. Thus, the following command sequence "*if –G <num>; cubes*" implements the mapper whose pseudo-code is shown in Figures 2 above. Command *addbuffs* implements the proposed buffering algorithm and its refinement presented in Figures 3 and 5, respectively.

Benchmarks used in the experiments are a selection of large public benchmarks from several sources. These benchmarks are available online [13]. They were used to evaluate improvements to FPGA mapping presented in several publications, i.e. [12]. The results of mapping reported in this section are verified using a combinational equivalence checker in ABC (command *cec*).

5.1 Area

The area experiments are presented in Tables I-III and shown on Figures 6 and 7. Table I shows area of the mapped networks before inserting buffers. Table II shows area after simple buffering. Table III shows area after iterative refinement.

The following ABC script is used for synthesis and mapping: *r file.blif; st; dch; if -g -K 6; st; resyn2; dch; if -G <N>; w file_map.blif*. This script combines delay optimization with structural choices (*dch; if –g –K 6; st*), delay-preserving AIG-based synthesis (*resyn2*) and technology mapping with structural choices into the chosen target technology (*dch; if -G <N>*), where *N* is the maximum allowed fanin count of (N)AND/N(OR) gates.

The runtime of experiments listed in Table I-III was dominated by computing structural choices and performing technology mapping. For large values of *N*, such as 12 and 14, the runtime of each example in the tables was approximately several minutes. For smaller *N*, the runtime was under one minute.

The tables show that area after simple buffering increases several times on average. The iterative refinement algorithm on average reduces the number of buffers 12-20%, depending on *N*. The reduction is larger for networks mapped into larger gates.

5.2 Delay

This section presents the delay results after technology mapping calculated using two delay models.

Table IV shows delay using unit-delay model. In this case, the larger is the gate size, the smaller is the delay.

Table V shows the same delay using a normalized delay model. In this model, a gate with N fanins, $1 \le N \le N_{max}$, has delay equal to $\ln(2*N)/\ln(2*N_{max})$. The reduction in delay when computed using the normalized delay model, increases with the gate size, as can be seen from the last line of Table V. This is because the larger is the gate size, the smaller is the fraction of large gates.

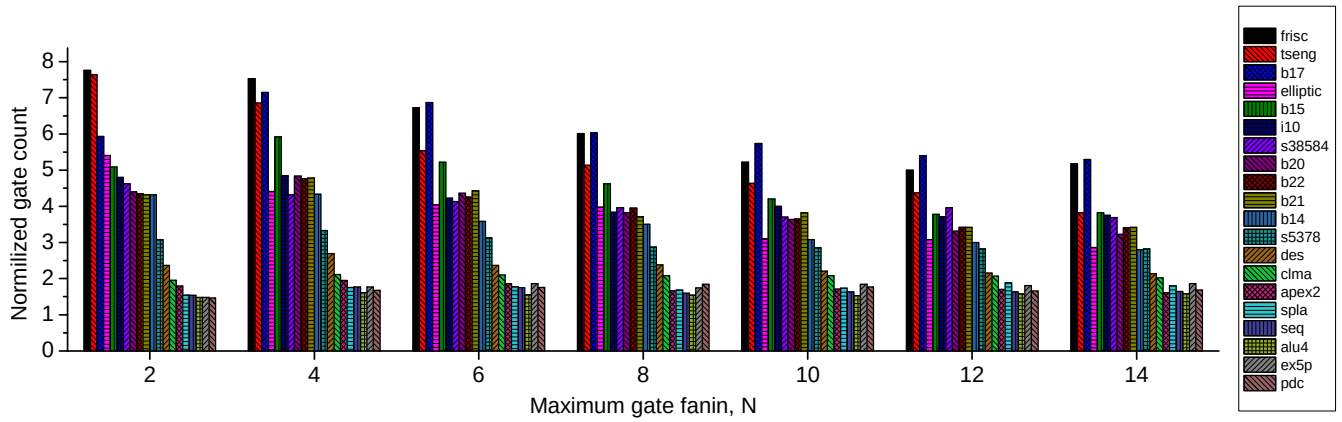

Fig. 6. Gate count after simple buffering (Table II), normalized with respect to the gate count in the original circuits (Table I).

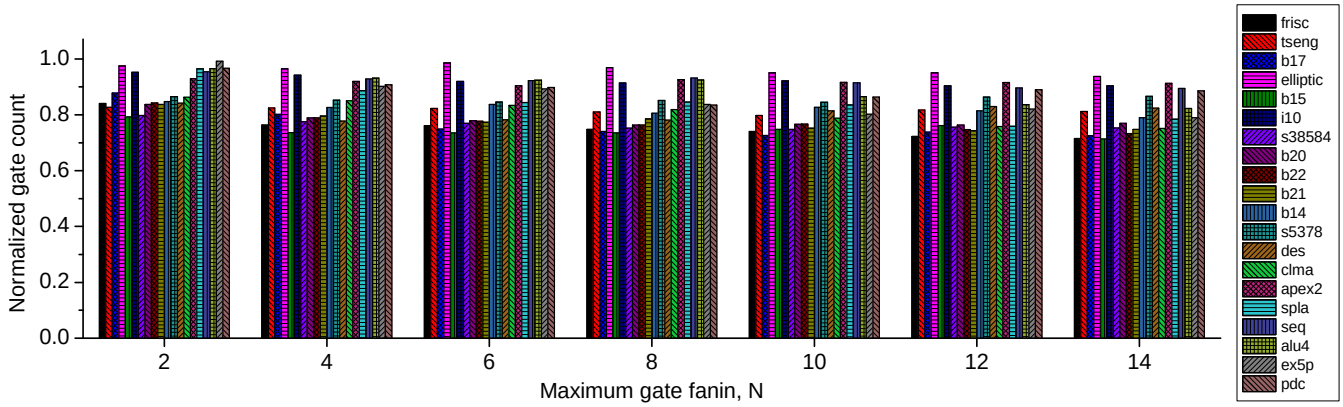

Fig. 7. Gate count after improved buffering (Table III), normalized with respect to the gate count after simple buffering (Table II).

6. DISCUSSION AND SUMMARY

The general empirical observation, which can be made from this case study, is that fewer logic levels in a circuitry results in smaller relative gate count overhead. Indeed, sequential circuits often have the smallest number of levels (Table IV) and are therefore less penalized by buffering according to Figure 6. For example, the increase in gate count due to buffering is less than the factor of 2 for the last seven circuits shown in Figure 6. Likewise, the use of larger gates in mapping reduces the number of levels (Table IV) and therefore gate count overhead (Fig. 6).

Intuitively, the correlation between the number of levels and buffering overhead is simple – fewer level of logic results in less slack and hence less need for buffering. Alternatively, circuits with the largest number of levels, such as **frisc**, are the least balanced and hence incur the largest overhead. Not surprisingly, the improved buffering technique works the best for these circuits (Fig. 7) because these circuits have the largest buffer reduction opportunities.

Balancing stateful logic circuits for the maximum processing throughput comes with a large overhead. It is therefore informative to assess the impact of such overhead by comparing throughput of the balanced circuits with the original ones. A practical metric for such comparison in the context of this study is a throughput per gate calculated as a clock cycle rate of the balanced circuits divided by its gate count normalized with respect to the corresponding ratio of the original circuit.

Due to normalization, a resulting throughput metric is insensitive to a particular number of PIs in the circuits, and therefore can be used to compare implementation of the different circuits. The values of throughput per gate computed using Tables I and V are shown in Figure 8.

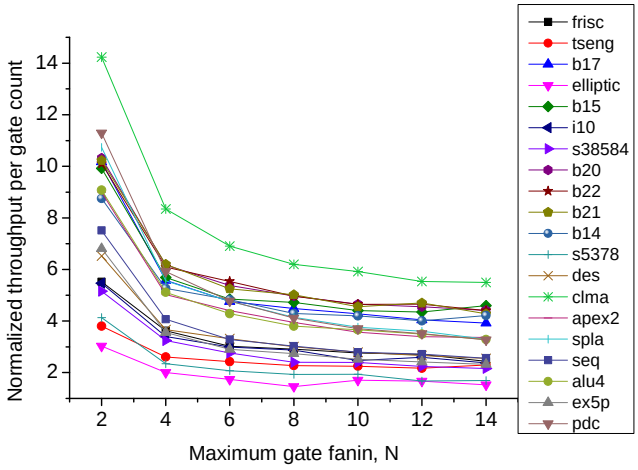

Figure 8. Throughput per gate count normalized with respect to original circuits.

Note that the numbers listed in this figure represent an ideal (upper bound) value for the throughput increase. According to Figure 8, normalized throughput is the best (7.5x on average) for 2-fanin gates. In this case, the decrease in clock rate overrides the savings in gate count due to using larger gates. These numbers do not account for the overhead of clock distribution network, placement, and routing. The former seems to be higher for the balanced stafeful logic given that each gate must be connected to the clock distribution network. However, the placement and routing overhead per gate is likely to be similar or less because additional buffers lower average Rent exponent [14].

In summary, in this paper, we investigated area overhead for implementing deeply pipelined synchronous circuits with stateful logic. Using advanced logic synthesis technique we show that buffering leads to an average factor of 2.3x (3.2x) gate count overheads for the 20 public benchmark circuits when mapped to low- (high-) fanin target library. Such high overhead of stateful logic may be still justified given the best case average 7x improvement in throughput per gate count metric.

7. ACKNOWLEDGMENT

This work is funded by NSF grant CCF-1017579.

8. REFERENCES

[1] B. Behin-Aein, D. Datta, S. Salahuddin, and S. Datta, "Proposal for an all-spin logic device with built-in memory", *Nature Nanotechnology*, vol. 5, pp. 266-270, 2010.

[2] A. Khitun and K.L. Wang, "Non-volatile magnonic logic circuits engineering", *Journal of Applied Physics*, vol. 110, art. 034306, 2011.

[3] J. Borghetti, G.S. Snider, P.J. Kuekes, J.J. Yang, D.R. Stewart, and R.S. Williams, "'Memristive' switches enable 'stateful' logic operations via material implication", *Nature*, vol. 464, pp. 873-876, 2010.

[4] S. Shin, K. Kim, and S.-M. Kang, "Reconfigurable stateful nor gate for large scale logic array integrations", *IEEE Trans. Circuits and Systems*, vol. 58 (7), pp. 442 – 446, 2011.

[5] D.B. Strukov and K.K. Likharev, "All-NDR crossbar logic", *Proc. IEEE Nanotechnology'11*, Portland, OR, Aug. 2011.

[6] F. Alibart, T. Sherwood, and D.B. Strukov, "Hybrid CMOS/nanodevice circuits for high throughput pattern matching applications", *Proc. AHS'11*, pp. 279-286, San Diego, CA, Jun. 2011.

[7] A. Orlov, A. Imre, G. Csaba, L. Ji, W. Porod, and G.H. Bernstein, "Magnetic quantum-dot cellular automata: Recent development and prospects", *Journal of Nanoelectronics and Optoelectronics*, vol. 3, pp. 1-14, 2008.

[8] H.T. Kung, "Why systolic architectures?", *Computer*, vol. 15 (1), pp. 37-46, 1982.

[9] A. Chattopadhyay and Z. Rakosi, "Combinational logic synthesis for material implication", in: *Proc. International Conference on VLSI and System-on-Chip*, pp. 200-203, October 2011.

[10] Berkeley Logic Synthesis and Verification Group. *ABC: A System for Sequential Synthesis and Verification*. http://www-cad.eecs.berkeley.edu/~alanmi/abc

[11] Mishchenko, S. Cho, S. Chatterjee, R. Brayton, "Combinational and sequential mapping with priority cuts", *Proc. ICCAD '07*, 2007.

[12] S. Ray, A. Mishchenko, N. Een, R. Brayton, S. Jang, and C. Chen, "Mapping into LUT structures", in: *Proc. DATE' 12*, 2012.

[13] A subset of ITC'99 and MCNC'89 benchmarks used in this paper: http://www.eecs.berkeley.edu/~alanmi/benchmarks/fpga/fpga_benchmarks_blif.zip

[14] P. Christie, "Rent exponent prediction methods", *IEEE Trans. VLSI*, vol. 8 (6), pp. 679-688, 2000

TABLE I: AREA AFTER TECHNOLOGY MAPPING INTO (N)AND/(N)OR WITHOUT BUFFERING.

Name	*K*=2	*K*=4	*K*=6	*K*=8	*K*=10	*K*=12	*K*=14
alu4	1384	999	941	886	853	827	757
apex2	1582	1102	1030	1066	1022	986	1048
b14	4801	4084	4309	4451	4988	5063	5941
b15	7955	6226	6251	6940	7180	7866	8749
b17	23293	18366	17754	18325	19345	19838	20296
b20	10266	8629	8874	9834	10485	11761	12429
b21	10263	8522	8660	9655	10276	11319	11865
b22	15089	12720	13372	14118	14934	16515	17310
clma	7870	6154	6319	6586	7291	7658	8438
des	3136	2461	2349	2321	2491	2588	2654
elliptic	714	612	606	616	779	783	710
ex5p	906	600	541	630	595	631	621
frisc	5949	5292	5232	5366	5548	5671	5479
i10	1689	1391	1455	1530	1498	1738	1609
pdc	3663	2789	2601	2530	2403	2575	2468
s38584	10145	8167	7314	7204	7158	7185	7230
s5378	1055	889	883	854	867	891	870
seq	1775	1214	1106	1097	1025	1029	981
spla	3432	2624	2494	2551	2337	2334	2541
tseng	1925	1903	1976	2062	2252	2391	2672
Ratio	1.000	0.797	0.781	0.807	0.830	0.866	0.884

TABLE II: AREA AFTER TECHNOLOGY MAPPING INTO (N)AND/(N)OR WITH SIMPLE BUFFERING.

Name	*K*=2	*K*=4	*K*=6	*K*=8	*K*=10	*K*=12	*K*=14
alu4	2052	1606	1456	1365	1307	1304	1200
apex2	2838	2144	1921	1770	1750	1677	1684
b14	20717	17702	15433	15593	15353	15203	16635
b15	40453	36859	32639	32067	30175	29693	33442
b17	138110	131352	121964	110481	110975	107139	107388
b20	45146	41697	38689	37588	38166	38935	40129
b21	44377	40774	38344	35829	39268	38694	40557
b22	65603	60592	56877	55805	54511	56472	58915
clma	15382	13012	13265	13719	15157	15849	17049
des	7432	6626	5559	5519	5495	5576	5673
elliptic	3863	2698	2449	2450	2416	2412	2031
ex5p	1342	1061	1008	1099	1098	1143	1156
frisc	46160	39828	35173	32263	28981	28362	28352
i10	8114	6743	6142	5867	5984	6451	6038
pdc	5370	4670	4559	4659	4244	4264	4146
s38584	46867	35273	30224	28508	26503	28410	26709
s5378	3244	2962	2758	2451	2471	2514	2454
seq	2727	2144	1933	1753	1672	1679	1612
spla	5301	4615	4436	4291	4059	4394	4588
tseng	14711	13046	10953	10589	10433	10469	10205
Ratio	1.000	0.848	0.774	0.747	0.735	0.745	0.744
Ratio Table I	3.767	3.920	3.576	3.298	3.107	2.988	2.921

TABLE III: AREA AFTER TECHNOLOGY MAPPING INTO (N)AND/(N)OR WITH IMPROVED BUFFERING.

Name	*K*=2	*K*=4	*K*=6	*K*=8	*K*=10	*K*=12	*K*=14
alu4	1981	1497	1346	1262	1131	1091	988
apex2	2636	1972	1737	1639	1604	1535	1538
b14	17551	14632	12923	12561	12693	12385	13140
b15	32067	27126	23996	23577	22582	22610	23866
b17	121312	105324	91369	81795	80539	79097	77862
b20	37806	32904	30122	28687	29247	29730	30878
b21	37142	32428	29670	28144	29533	28755	30349
b22	55264	47818	44194	42621	41815	42202	43178
clma	13270	11061	11064	11237	11949	12012	12810
des	6253	5153	4347	4310	4474	4626	4675
elliptic	3768	2603	2415	2375	2296	2293	1904
ex5p	1331	957	900	920	881	938	913
frisc	38807	30416	26760	24136	21457	20490	20277
i10	7724	6353	5647	5362	5514	5828	5457
pdc	5191	4241	4092	3891	3666	3794	3674
s38584	37384	27335	23247	21458	19838	21482	20114
s5378	2808	2525	2334	2088	2088	2171	2125
seq	2600	1990	1783	1634	1530	1505	1442
spla	5117	4093	3745	3630	3392	3337	3600
tseng	12165	10753	9018	8577	8316	8550	8286
Ratio	1.000	0.809	0.729	0.695	0.678	0.682	0.675
Ratio Table II	0.888	0.848	0.838	0.827	0.820	0.814	0.807

TABLE IV: DELAY AFTER TECHNOLOGY MAPPING INTO (N)AND/(N)OR USING UNIT-DELAY MODEL.

Name	*K*=2	*K*=4	*K*=6	*K*=8	*K*=10	*K*=12	*K*=14
alu4	13	10	9	9	9	9	9
apex2	15	12	11	10	10	9	9
b14	32	26	21	21	19	18	18
b15	40	34	29	28	26	25	25
b17	53	46	40	35	34	33	32
b20	38	33	28	25	25	24	25
b21	37	33	28	25	26	23	25
b22	37	33	27	26	24	23	23
clma	24	21	19	18	17	17	17
des	13	11	9	9	9	9	9
elliptic	16	11	10	10	9	9	7
ex5p	10	7	7	6	6	6	6
frisc	36	28	24	21	18	17	17
i10	25	20	17	15	16	16	15
pdc	16	12	11	10	9	9	9
s38584	19	14	12	11	10	11	10
s5378	11	10	9	8	8	8	8
seq	11	9	8	7	7	7	7
spla	16	11	10	9	8	9	8
tseng	24	21	17	16	15	15	14
Ratio	1.000	0.808	0.705	0.651	0.623	0.614	0.599

TABLE V: DELAY AFTER TECHNOLOGY MAPPING INTO (N)AND/(N)OR USING NORMALIZED UNIT-DELAY MODEL.

Name	*K*=2	*K*=4	*K*=6	*K*=8	*K*=10	*K*=12	*K*=14
alu4	13.00	7.67	6.13	5.41	4.85	4.61	4.29
apex2	15.00	9.00	7.43	6.07	5.60	5.29	4.94
b14	32.00	18.86	14.49	12.11	10.69	9.80	9.33
b15	40.00	24.77	18.63	16.05	13.85	12.51	12.55
b17	53.00	32.00	24.50	20.02	17.80	16.13	15.03
b20	38.00	23.53	18.25	14.51	12.97	11.52	10.96
b21	37.00	23.58	17.98	14.64	13.10	11.94	10.95
b22	37.00	22.91	18.31	14.98	13.00	11.89	11.00
clma	24.00	15.00	12.09	10.57	9.70	8.69	8.34
des	13.00	7.67	6.13	5.58	5.01	4.76	4.50
elliptic	16.00	8.53	6.93	5.62	5.03	4.88	4.08
ex5p	10.00	5.67	4.83	3.99	3.73	3.59	3.41
frisc	36.00	20.86	15.46	13.08	10.68	9.76	9.04
i10	25.00	15.53	11.60	10.07	9.11	8.70	8.05
pdc	16.00	9.00	7.52	6.32	5.66	5.18	4.87
s38584	19.00	10.86	8.77	7.16	6.63	6.71	6.02
s5378	11.00	6.67	5.49	4.72	4.67	4.09	4.14
seq	11.00	6.67	5.30	4.51	4.16	3.98	3.76
spla	16.00	8.67	7.15	5.90	5.46	5.15	4.66
tseng	24.00	14.72	11.07	9.44	8.32	7.77	7.13
Ratio	1.000	0.594	0.470	0.395	0.356	0.331	0.309
Ratio Table IV	1.000	0.736	0.667	0.607	0.571	0.539	0.517