Folded Logic Decomposition

Dennis Wu, Jianwen Zhu
Electrical and Computer Engineering
University of Toronto, Ontario M5S 3G4, Canada
{wudenni, jzhu}@eecg.toronto.edu

ABSTRACT

The past decade of logic synthesis research has been characterized by a constant quest for smaller circuit area and faster circuit speed. While enjoying enormous success, the runtime of classic logic synthesis algorithms cannot keep up with the exponential growth of circuit complexity predicted by Moore's law. In this paper, we propose a new technique where the regularity of the logic circuits, be it the apparent regularity in arithmetic-intensive circuits, or the hidden regularity in random logic, is automatically discovered and exploited to improve synthesis speed. We show that with this technique, the task of functional decomposition, the core engine of logic synthesis, can be made orders of magnitude faster than usual for arithmetic circuits, and on average eight times faster than usual for arbitrary random circuits.

1. INTRODUCTION

Circuit complexity increases exponentially as dictated by Moore's law. However, the classic logic synthesis algorithms are of polynomial complexity with circuit size at best. This imposes serious problems in terms of synthesis speed for modern circuits. The conventional wisdom of divide-and-conquer, where the partitions specified by designers are used as boundaries for separate synthesis, is often found to compromise optimization quality at an intolerable level. Finding new logic synthesis algorithms that scale well with circuit size, has emerged as a new priority, in addition to the traditional metric of synthesis quality, in terms of circuit area and speed. The need for fast logic synthesis algorithms is even more pronounced when combating deep sub-micron issues, where early design planning is needed.

To address the scalability problem, we propose a logic synthesis approach that exploits the regularities inherent in circuits. Regularities in array-based circuits such as arithmetic units are apparent since many logic blocks are replicated. For example, one implementation of the N-bit ripple carry adder uses N instances of the full adder logic block. Regularities in random logic often exist as well. With the crude, yet *quantitative* measure of regularity in Definition 1, we typically found the regularity of datapath circuits to be on the order of several hundreds. It is interesting to note that it is usually the datapath circuit that blows up circuit complexity.

DEFINITION 1. *The regularity of a logic network is defined to be the number of logic components divided by the number of logic component types in the network.*

The key to exploiting regularities is to determine if two logic components in a logic network implement the same logic function. This poses a problem for traditional logic synthesis systems where each function block maintains a separate copy of the Boolean function it implements. We propose a new strategy, called logic folding, where logic components with the same logic functions are easily identified.

DEFINITION 2. *Logic folding is a representation of a logic network that groups logic components that implement the same logic function together by explicitly sharing the same logic function representation.*

Algorithms to extract the structural regularity in a logic network have been attempted in the past [8][7]. While regularity has been preserved to improve layout efficiency, the synthesis runtime improvement has been modest (around 30%). Logic folding is a different strategy, as we shall later demonstrate, where the regularity of logic network can be implicitly maintained by exploiting the canonical property of Bryant's BDD data structure [2].

In this paper, we focus on how logic folding can be applied to logic decomposition. Noting that many circuits exhibit a fair amount of regularity; And noting that functional decomposition is performed with respect to a Boolean function; we propose to use logic folding to identify regularities and to apply the decomposition step on all logically equivalent components at once.

In the text that follows, a description of the traditional circuit representation and decomposition process is given and later contrasted with our folded approach. Logic folding and its application to functional decomposition is then described in detail. Finally, the runtime performance of the folded logic approach is tested against the MCNC benchmark and a set of arithmetic circuits.

2. BACKGROUND

2.1 Circuit Representation

In this paper, we only deal with completely specified, single output Boolean functions. A *Boolean function* is a mapping between the values of its n input signals to the value of its output signal $\{0,1\}^n \rightarrow \{1,0\}$.

A Boolean function can also be represented as an interconnection of simpler Boolean functions in a graph called a *Boolean Network*. A Boolean network is a directed acyclic graph $G = \langle V, E \rangle$ where the vertices represent function blocks and the edges represent signals that are produced and consumed by the function blocks.

A *function block* is characterized by two independent information, its Boolean function and its support set. A support set is the set of signals consumed by a function block. It is defined as $supp(v) = \{\langle v,w\rangle \in E\}$. The function block generates one or no output signals. The consumers of this signal is defined as $out(v) = \{\langle u,v\rangle \in E\}$. v is called a primary input if $|supp(v)| = \oslash$. v is called a primary output if $|out(v)| = \oslash$.

Efficient representation of the Boolean function has been the subject of much attention. Modern logic synthesis systems have benefited from representing Boolean functions as a Reduced Ordered Binary Decision Diagram (ROBDD or just BDD for short) data structure introduced by Bryant [2]. The BDD has several attractive properties. It is canonical: function blocks that implement the same Boolean function and share the same support set have the same BDD representation. As a result, equivalent nodes in the BDD can be identified and collapsed easily to make the BDD compact. The BDD has also been shown to have several fast decomposition algorithms including XOR and Boolean decompositions that previous to the BDD, were difficult to perform [12].

2.2 Decomposition

Often, the circuit generated by an HDL will contain gates that are too large for practical implementation. These gates must be recursively broken down in to smaller gates of lesser complexity in a process called *Decomposition*. A number of papers have been published on the subject of BDD based decomposition [6][12][1].

3. FOLDED LOGIC DECOMPOSITION

In traditional logic synthesis systems, functional decomposition is performed one function block at a time [12] [11]. This may be desirable if decomposition decisions are affected by considerations such as differences in fan-in or fan-out; this is the case in timing-driven and placement-driven decomposition. However, a wide variety of situations occur when such distinctions are not necessary, for example, in area-driven decomposition or ultra-fast algorithms used in early design planning when quality can be compromised. For these decomposition algorithms, function blocks that implement the same Boolean function will be decomposed in the same way. We can reduce the number of decompositions performed by sharing a decomposition result among function blocks with the same Boolean function. In this section we discuss an efficient method of testing Boolean functions for equivalence, called logic folding, and describe how it can be used to perform decompositions in parallel.

3.1 Logic Folding

To perform folded decomposition, we must identify which function blocks implement the same Boolean function. This requires every new function block to have its Boolean function compared to existing Boolean functions to determine if there is a match. In the traditional setup, where function blocks store their own local copy of the Boolean function, testing Boolean functions for equivalence requires $O(n)$ time BDD traversals. The overhead of comparing BDDs this way outweighs the benefits of performing the decompositions in parallel.

The folded logic structure we propose stores Boolean functions in a common repository where they can be checked for equivalence in constant time. Instead of storing a separate copy of a Boolean function with each function block, the Boolean functions are stored as BDDs in a global BDD Manager. The variables of this Manager are generic; they do not represent a specific node in the Boolean network. The N variables of the BDD are simply mapped to the bottom N variables of the BDD Manager. Because BDDs are canonical, two logic functions are equivalent if their BDD pointers are the same.

ALGORITHM 1. *Folded Decomposition*

```
forall( $BddClassesintheBooleannetwork$ )
  Push(heap, BddClass, BddClass.numVariables);
while( $f = Pop(heap)$ ) {
  $\langle g,h,op\rangle = decompose(f)$;
  if( $!bddExists(g)$ )
    create BddClass for g;
  if( $!bddExists(h)$ )
    create BddClass for h;
  update instances(f, g, h, op);
  if( $g\_has\_more\_than\_two\_nodes$ )
    Push(heap, g, g.numVariables);
  if( $h\_has\_more\_than\_two\_nodes$ )
    Push(heap, h, h.numVariables);
  }
```

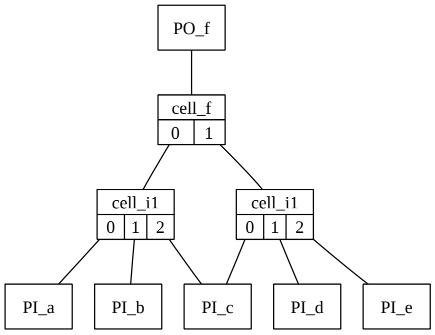
Once two Function blocks are computed to have the same Boolean function, they are grouped together into an equivalent class, called a *BddClass*. The BddClass represents a BDD in the Global BDD Manager. It stores a pointer to the BDD it represents and maintains a list of BddInstances that use it. A *BddInstance* represents a function block. It contains a pointer to it's BddClass and a support set, which is an ordered list of pointers to other BddInstances. BddClasses are the unit by which folded operations are performed.

3.2 Folded Decomposition

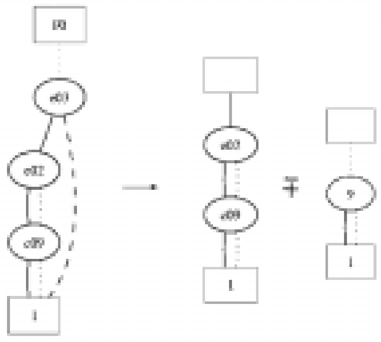
The folded decomposition algorithm is shown in Algorithm 1. Each decomposition is performed one BddClass at a time (and potentially more than one BddInstance at a time). The BDD for the BddClass is decomposed into two or more smaller BDDs. If these BDDs are not found in the Global BDD Manager, new BddClasses are created for them. Otherwise, the existing BddClasses are used. The BddInstances are updated to reflect the changes. If the new BddClasses can be decomposed further, they are added to the heap. The heap is used to decompose the BddClasses in order of non-increasing size of their support set. Decomposing BDD's in this order ensures that no decompositions are repeated.

An illustration of the traditional decomposition process is shown in Figure 1. Figure 1a shows a Boolean network before decomposition. One function block is selected for decomposition. In Figure 1b, the function block is OR-decomposed. The changes are shown in figure 1c, where the original function block has been replaced with three new function blocks of lesser complexity. Notice that the original Boolean network contains two function blocks with the same BDD. The next decomposition step will basically repeat the same BDD decomposition. We do away with this repetition in folded decomposition.

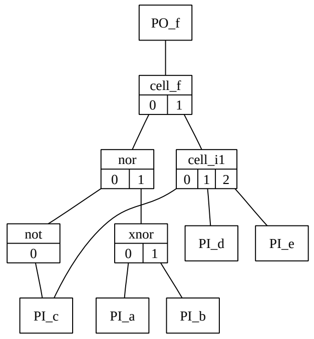
The folded decomposition process is illustrated in Figure 2. The Boolean network shown in Figure 2a is identical to the one used earlier. However, the Boolean functions for the function blocks are stored in a Global manager where two unique Boolean functions have been identified (Figure 2b). The BDD with the largest support set, cell_i1, is decomposed first. The decomposition result transforms cell_i1 into the XOR of a XNOR and NOT gate (Figure 2c). cell_i1's BDD is then removed from the Global manager and replaced with XNOR, NOR and NOT BDDs (Figure 2e). The two original cell_i1 instances are replaced with NOR, XNOR and NOT instances to reflect the changes in the Boolean network (Figure 2d).



(a) Boolean network before decomposition.

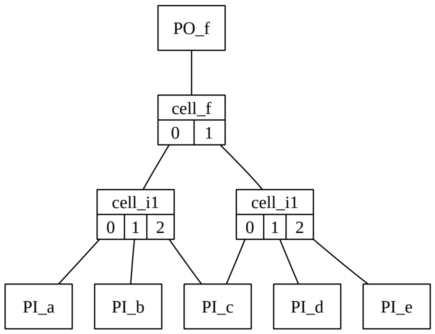


(b) Decomposition of cell_i1.

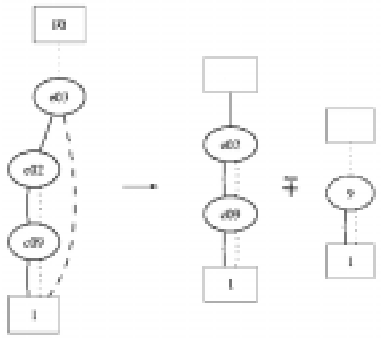


(c) Boolean network after decomposition.

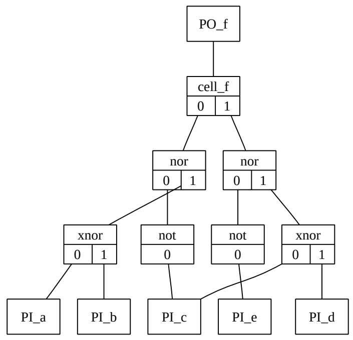
Figure 1: Decomposition using traditional approach.



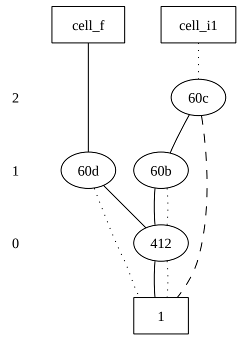
(a) Boolean network before decomposition.



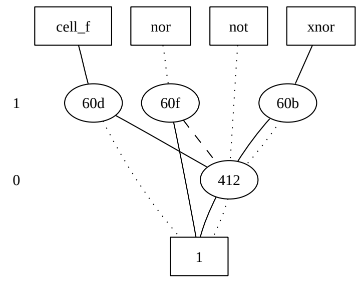
(c) Folded decomposition of cell_i1.



(d) Boolean network after decomposition.



(b) Global BDD manager before decomposition.



(e) Global BDD Manager after decomposition.

Figure 2: Decomposition using folded approach.

3.3 Boolean Matching

Folded Logic is not without its problems and limitations. Two Boolean functions can have more than one BDD representation when the variables of the BDDs are mapped differently to the generic variables of the Global BDD Manager. When this occurs, the match is missed and the BDDs are mistaken as different Boolean functions. As a result, the BDDs must be decomposed separately. Given that for an n variable function, there are up to $O(n!)$ different variable orderings, it seems unlikely for two equivalent functions to match.

The set of valid variable orders for a function can be dramatically reduced by applying variable reordering. Since BDDs are typically stored in reduced form to conserve memory, this comes at no extra cost. However, variable ordering alone is not sufficient for discovering most matches. To illustrate this point, the sifting variable ordering algorithm [10] is applied to several logically equivalent BDDs, whose starting variable orders are randomly chosen. This procedure is applied to the largest gate in each of the MCNC benchmarks. Table 1 (PMSIFT) shows the rate of matching obtained. Ideally, matching should be 100% because the logic functions compared are equivalent. However, on average, sifting can only discover 17% of the matches.

The problem described above, called permutation-independent Boolean Matching, has been investigated in the context of library cell binding and logic verification. Ercolani and De Micheli [5] propose a matching algorithm for EPGAs where BDDs are constructed for all possible input permutations of the uncommitted EPGA module, and stored together in a global manager. While practical for EPGA mapping, where logic functions are only compared against the EPGA module, the memory requirements of this approach are not practical when arbitrarily many gates are compared with each other. Debnath and Sasao [4] devise a permutation-independent, canonical form for the logic function where the rows of the truth table are represented as bits in a bit vector. Each bit indicates whether the row is part of the on-set of the function. The size of the bit vector grows exponentially with the size of the support set, and is not practical for regularity detection, where large gates exist. Ciric and Sechen [3] propose a canonical form where the function is represented as the concatenation of minterms. Their algorithm performs an exhaustive branch and bound search for the unique identifier which can be obtained through minterm reordering and variable reordering. Their algorithm also cannot handle the large gates that may exist in a logic network.

Mohnke, Molitor and Malik [9] propose a solution to the Boolean matching problem that does not suffer from the runtime and memory limitations of the algorithms described earlier. Their technique is based on computing signatures for the inputs of the logic functions that are independent of the variable order. The inputs can be sorted by their signatures to generate a variable order. Two BDDs that represent the same Boolean functions will produce the same input signatures. If each input signature is unique, then a unique variable ordering can be created from the signatures, and the resulting BDDs will be equivalent.

One example of a input signature is the *Cofactor satisfy count signature*[9]. For an input variable x, it's input signature is defined to be the number of input assignments for which the logic function is true when x is true. In BDD representation, this corresponds to the number of paths to the one terminal and can be computed in $O(n)$ time, where n is the number of nodes in the BDD. The limitation of this approach is that, unlike the Boolean matching techniques described earlier, it does not result in a canonical form. Input signatures may alias, resulting in non-unique variable orders. In folded decomposition, where the goal is to decrease runtime, performing decomposition on missed matches separately is preferable to performing a costly search for perfect matches. Using the Cofactor satisfy count signature to create an initial variable order allows 78% of the matches to be found. The match rate for individual circuits in the MCNC benchmark is shown in Table 1 (PMSIG). Mohnke et al report in [9] that for the set of signatures they implement, unique signatures are obtained in 92% of the circuits in the LGSynth91 and ESPRESSO benchmarks.

In spite of the problems described above, the potential benefit of logic folding is huge. If a match is found early on, there are savings on the immediate decomposition, as well as on all downstream decompositions. Folding is essentially free. The cost of folding is to copy BDDs to and from the global BDD manager, but this copying is required anyways when isolating a BDD for variable reordering.

4. EXPERIMENTAL RESULTS

4.1 Implementation

The folded decomposition approach is not dependent on any specific BDD based decomposition algorithm. For the purposes of implementing a complete decomposition system, the fast, disjunctive AND-OR-XOR decompositions and MUX decomposition algorithms presented in BDS[12] are used. BDS has shown to result in 40% less literals, 23% lower gate count and use 84.4% less CPU time then the popular SIS program for XOR intensive circuits. Applying folded logic synthesis to a different set of decomposition algorithms should not dramatically affect the speedup obtained.

In an effort to improve the rate of matching, BDDs are put into their semi-canonical, permutation independent form using the fast signature based approach. A number of signatures have been proposed in [9]. In our tests, we apply the *Cofactor satisfy count signature* to produce a semi-unique initial variable order.

4.2 Procedure

The experiments were conducted on a SUN UltraSPARC 5 with 320 MB memory, running SunOS version 5.8. Two sets of benchmarks were used in the test. The first set of circuits were taken from the combinational multi-level examples of the MCNC91 benchmark. Only those circuits which were reported by [13] to have an approximate gate count of 500 or more were selected for testing. To demonstrate the performance of folded decomposition on regular circuits, a second benchmark consisting of arithmetic circuits is used.

Our synthesis system does not perform all the functions described in the typical synthesis flow. Sweep and Eliminate have not been implemented. Although these stages are likely to collapse away regularity, they are essential in reducing the gate count and run time, and therefore must be included in the testing process. The sweep and eliminate capabilities of SIS are leveraged. Circuits are loaded into SIS where the “sweep;” and “eliminate -1;” commands are applied. The result is saved and folded decomposition is performed on the preprocessed circuits.

The statistics collected are defined below. *Folded Decomposition Count* (FDC) is the number of decompositions performed when decomposition results are shared among logically equivalent function blocks. *Regular Decomposition Count* (RDC) is the number of decompositions performed when decomposition is performed one function block at a time. *Expected Speedup* (ES) is equal to RDC divided by FDC.

Expected Speedup gives a ratio of the number of decompositions saved using folded decomposition over regular decomposition. Expected Speedup is roughly representative of the speed improvement that can be expected during decomposition. We also report the Actual Speedup experienced by timing the folded and regular decompositions. *Folded Decomposition Time* (FDT) is the time required to perform folded decomposition. *Regular Decomposition Time* (RDT) is the time required to perform regular decomposition. *Actual Speedup* (AS) is equal to RDT divided by FDT.

4.3 Results

4.3.1 MCNC Benchmark

The results for the MCNC benchmark are shown in Table 1. Circuit C6288.blif is an extreme case of logic folding and will be discussed later. Excluding circuit C6288.blif, the average expected and actual speedups are 4.22 and 8.33 respectively. The smallest speedup achieved is 1.01 while the greatest speedup achieved is 41.50. The runtime benefit of logic folding is quite apparent in these results.

When circuit C6288.blif is included in the results, the average expected and actual speedups become 15.07 and 8.26 respectively. Circuit C6288.blif has an expected speedup of 232 but an actual speedup of only 7. Because of the high amount of regularity in the circuit, factors other than decomposition, such as creating new gate instances, become significant. This puts a lower bound on the speed improvements that folded decomposition can achieve. In the case of regular decomposition, C6288.blif performs 464 decompositions and creates 464 new gates. In the folded case, only 2 decompositions are performed but 464 new gates are still created. The decompositions performed were also of the fast algebraic variety. This illustrates that if the expected speedup is high and the types of decompositions performed are fast, then other factors will become significant, reducing the actual speedup.

4.3.2 Array Circuit Benchmark

The speedups for arithmetic circuits are even more encouraging. An average actual speed up of 81.58 is reported. The smallest speedup of 4 is achieved for the 8-bit ripple carry adder, while the largest speedup of 299.85 is achieved for the 64-bit Wallace tree multiplier. For the circuits that have their FDT values marked by an asterisk, the decomposition times were too fast to be captured. A conservative estimate of 10ms is assigned to these circuits for the purpose of computing an “actual speedup” value. The AS values for these circuits, in turn, are also conservative.

Although our decomposition system shares some similarities with BDS, a direct comparison of the two systems may not be fair. Our system implements only a subset of BDS’s decomposition algorithms, albeit the fast ones, and does not perform sharing extraction. As a result, our decomposition system runs approximately ten times faster than BDS even with logic folding disabled and produces slightly higher gate counts. Our contribution does not run in opposition to the approaches taken by other decomposition systems. Logic folding can be applied to gain additional speedups in decomposition systems where decomposition decisions are made independently of fan-in and fan-out, as is done in BDS.

5. CONCLUSION

In this paper, a fast method for identifying logically equivalent function blocks is presented. These ideas are applied to logic decomposition to allow sharing of decomposition results among logically equivalent function blocks.

Implementing these ideas against MCNC benchmarks and arithmetic circuits, we have achieved very significant speedups in decomposition time, making the folded decomposition approach attractive for large, regular designs.

For future work, we anticipate that logic folding can also find applications in other stages of logic synthesis, such as sweep and eliminate. We are also investigating ways to increase the rate of logic matching by using matching driven decomposition and variable reordering techniques.

We expect that the theme of logic folding, to perform synthesis operations in parallel, will find many more applications and will help to meet the demands of increasing circuit sizes.

6. REFERENCES

[1] V. Bertacco and M. Damiani. The disjunctive decomposition of logic functions. In *1997 IEEE/ACM International Conference on Computer-Aided Design*, 1997.

[2] R. E. Bryant. Graph-based algorithms for boolean function manipulation. In *IEEE Transactions on Computers, Vol. C-35, No. 8*, pages 677–691, 1986.

[3] J. Ciric and C. Sechen. Efficient canonical form for boolean matching of complex functions in large libraries. In *2001 IEEE/ACM International Conference on Computer Aided Design*, 2001.

[4] D. Debnath and T. Sasao. Fast boolean matching under permutation using representative. In *Asia and South Pacific Design Automation Conference, ASP-DAC’992001 IEEE/ACM*, pages 359–362, 1999.

[5] S. Ercolani and G. D. Micheli. Technology mapping for electrically programmable gate arrays. In *28th ACM/IEEE Design Automation Conference*, 1991.

[6] K. Karplus. Using if-then-else dags for multi-level logic minimization. In *http://www.cse.ucsc.edu/ karplus/research.html*, 1988.

[7] T. Kutzschebauch. Efficient logic optimization using regularity extraction synthesis. In *Proceedings of the International Conference on Computer Design*, Austin, 2000.

[8] T. Kutzschebauch and L. Stok. Regularity driven logic synthesis. In *Proceedings of the International Conference on Computer-Aided Design*, pages 439–446, San Jose, 2000.

[9] J. Mohnke, P. Molitor, and S. Malik. Application of bdds in boolean matching techniques for formal logic combinational verification. In *International Journal on Software Tools for Technology Transfer*, pages 48–53, 2001.

[10] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of the International Conference on Computer-Aided Design*, pages 42–47, 1993.

[11] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanaha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. Sis: A system for sequential circuits synthesis. Technical Report UCB/ERL M92/41, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720, 1992.

[12] C. Yang, M. Ciesielski, and V. Singhal. BDS: A BDD-based logic optimization system. In *Proceeding of the 37th Design Automation Conference*, pages 92–97, 2000.

[13] S. Yang. Logic synthesis and optimization benchmarks user guide version 3.0. Technical report, Microelectronics Center of North Carolina, P. O. Box 12889, Research Triangle Park, NC 27709, 1991.

			Decomp. Count		Decomp. Time		Speedup			Matching	
Circuit	Time (ms)	Gate Count	FDC	RDC	FDT (ms)	RDT (ms)	ES	AS		PMSIFT (%)	PMSIG (%)
C1355.blif	180	174	6	12	20	30	2.00	1.50		10.42	100.00
C1908.blif	260	446	44	172	40	120	3.91	3.00		1.14	100.00
C2670.blif	470	653	81	275	160	430	3.40	2.68		1.00	100.00
C3540.blif	700	972	145	386	170	610	2.66	3.58		22.68	76.16
C5315.blif	520	962	106	515	20	830	4.86	41.50		2.38	100.00
C6288.blif	750	2353	2	464	10	70	232.00	7.00		50.72	100.00
C7552.blif	860	1335	85	555	60	940	6.53	15.66		1.00	100.00
alu4.blif	27480	7947	1674	4850	14840	16110	2.90	1.08		-	-
dalu.blif	920	1655	216	975	390	2210	4.51	5.66		1.06	37.56
des.blif	2130	2645	299	1442	300	2790	4.82	9.30		1.54	26.78
frg2.blif	1480	2112	321	1239	820	1970	3.86	2.40		1.00	25.94
i10.blif	1940	2062	447	994	560	1480	2.22	2.64		1.00	25.98
i8.blif	510	1480	91	1028	120	1860	11.30	15.50		1.04	100.00
i9.blif	280	551	42	295	30	130	7.02	4.33		11.04	100.00
k2.blif	3810	1823	172	976	100	360	5.67	3.60		32.74	100.00
pair.blif	8010	1076	155	477	150	5970	3.08	39.80		79.58	76.36
rot.blif	1100	706	271	389	530	540	1.44	1.01		1.00	100.00
t481.blif	2960	1949	461	989	900	1460	2.15	1.62		1.02	100.00
too_large.blif	64970	10906	2798	6782	15920	24310	2.42	1.52		-	-
vda.blif	1040	1007	81	536	20	120	6.62	6.00		100.00	100.00
x3.blif	740	885	196	590	260	1110	3.01	4.26		1.02	17.52
AVERAGE	5767	2081	366	1140	1687	3021	15.07	8.26		16.91	78.23

Table 1: Experimental results on MCNC benchmark.

			Decomp. Count		Decomp. Time		Speedup	
Circuit	Time(ms)	Gate Count	FDC	RDC	FDT (ms)	RDT (ms)	ES	AS
adder8.blif	80	40	1	8	*10	40	8.00	4.00
adder16.blif	100	80	1	16	*10	80	16.00	8.00
adder32.blif	100	160	1	32	*10	170	32.00	17.00
adder64.blif	150	320	1	64	10	360	64.00	36.00
adder128.blif	270	640	1	128	10	610	128.00	61.00
mul8.blif	110	320	1	48	*10	300	48.00	30.00
mul16.blif	340	1408	1	224	10	1140	224.00	114.00
mul32.blif	6120	5888	1	960	30	4930	960.00	164.33
mul64.blif	262930	24064	1	3968	70	20990	3968.00	299.85
AVERAGE	30022	3658	1	605	19	3180	605.33	81.57

Table 2: Experimental results on arithmetic circuits.