CacBDD: A BDD Package with Dynamic Cache Management

Guanfeng Lv[1], Kaile Su[2,3]*, and Yanyan Xu[4,5]

[1] School of Comput. Sci. and Tech., Beijing University of Technology, Beijing, China
[2] IIIS, Griffith University, Brisbane, Australia
[3] Key laboratory of High Confidence Software Technologies, Ministry of Education, Beijing, China
[4] School of Inf. Sci. and Tech., Beijing Forestry University, China
[5] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China

Abstract. In this paper, we present CacBDD, a new efficient BDD (Binary Decision Diagrams) package. It implements a dynamic cache management algorithm, which takes account of the *hit-rate* of computed table and available memory. Experiments on the BDD benchmarks of both combinational circuits and model checking show that CacBDD is more efficient compared with the state-of-the-art BDD package CUDD.

1 Introduction

BDDs are successfully used in computer-aided verification for their efficient representation and manipulation of Boolean functions [1], and BDD packages constitute the base of some verification tools, such as NuSMV [11] and Jtlv [12]. As given in [2], classic methods of efficient implementation of BDD package include: unique table, computed table, complement edges, garbage collection and dynamic variable ordering. Besides, careful allocation of nodes can also speed up BDD packages as it reduces cache misses [4]. Even though modern BDD packages have the same cornerstone constituted by the techniques mentioned above, they may differ in a number of ways. For example, some BDD packages are pointer based, e.g. [3, 4], and some others, e.g. [5, 6, 13], use integer indices instead. As for computed table, some packages use a single computed table, and others use separate computed tables.

The unique table and computed table constitute the base data structures of modern BDD packages. The unique table is built as a hash table and contains all the BDD nodes with the hash collisions resolved by chaining. In some BDD packages, the unique table is implemented as a family of sub-tables and each of them is associated with a variable for facilitating the dynamic variable reordering [13]. The computed table (also called operation cache) is a hash-based cache to record a part of the previous BDD operation results and is usually implemented without a collision chain. Complement edge is also adopted by most

* Corresponding author:kailepku@gmail.com

modern BDD packages to reduce both space and time. Besides, garbage collection and dynamic variable reordering are important for decreasing the overall size of BDDs. Nevertheless, both of them are time consuming. The overhead of garbage collection is non-negligible and dynamic variable reordering gets the lion's share of the CPU time [9].

The size of computed table has a significant impact on BDD computations in many applications, such as model checking [10]. The management of computed table is very important for a BDD package's performance, and how to find a good dynamic cache management algorithm is the first open problem given by Yang etc. [10]. Brace etc. [2] indicated that it would be easy to control the memory and run-time tradeoff by adjusting the ratio of the number of unique-table entries to the number of computed-table entries. The ratio of the above two numbers is called *hit-rate*. However, the method is still static and preliminary, and simply controlling the *hit-rate* does not work in many cases. In the well-known BDD package CUDD, a policy called "reward-based" is adopted [9]. The policy is as follows: if a large *hit-rate* of a computed table is observed, then it is worthwhile to increase the size of the computed table. Obviously, the power of the policy is limited because the algorithm considers only the *hit-rate* of a computed table, and the so-called *large* value of hit rate is not dynamically adjusted in the process of computation.

In this paper, we present CacBDD[6], a new integer-indices based BDD package. Besides careful implementations of routine techniques for efficient modern BDD packages, we adopt a new dynamic cache management method, which significantly accelerates BDD operations as indicated by the experimental results. This novel method provides a promising solution to an important open problem raised by Yang etc. [10]. Also, CacBDD has some other novel features including a new garbage collection technique.

2 Implementation

2.1 Cache Management Algorithm

CacBDD is developed in C++, which is an index-based package similar to IBM's BDD [5] and TiniBDD [6]. It supports usual operations and those useful for model checking purpose, including multiple-operand ones like AndExist. In this paper we will not discuss the details of traditional techniques used in CacBDD which can be found in [2, 4–6, 9, 14]. The main novel techniques described in this section include a dynamic cache management method and a delayed garbage collection strategy.

Computed table is used as a cache to improve BDD manipulation, and its size is limited by the available memory. It is a space and time tradeoff issue. In many applications, the *hit-rate* of computed table is a function of the instance at hand. Because the *hit-rate* of computed table is dynamic, the size of computed table should be adjusted dynamically. Therefore, if the new *hit-rate* of computed

[6] Available at http://kailesu.net/CacBDD

```
ite($F, G, H$)
 1: $ccc = ccc + 1$;
 2: if ($ccc >= occ$) AND ($cts < limitedValue$) then
 3:    if ($cchr >= ochr$) OR ($cts < nc * cchr$) then
 4:       computed_table_increase_size();
 5:    end if
 6:    $ochr = cchr$;
 7:    $occ = 2 * ccc$;
 8: end if
 9: if (terminal case) then
10:    return result;
11: end if
12: if (computed-table has entry $\{F, G, H\}$) then
13:    return result;
14: else
15:    Let $v$ be the top variable of $\{F, G, H\}$;
16:    $T$ =ite($Fv, Gv, Hv$);
17:    $E$ =ite($F\overline{v}, G\overline{v}, H\overline{v}$);
18:    if ($T == E$) then
19:       return $T$;
20:    end if
21:    R = find_or_add_unque_table($v, T, E$);
22:    Insert_computed_table($F, G, H, R$);
23:    return $R$;
24: end if
```

Fig. 1. The ite operation with dynamic computed table management algorithm.

table is larger than the old one after resizing the computed table, then it should be necessary to extend the size of the computed table.

The idea above leads to our dynamic computed table management method. The algorithm for *ite* operation (the core of BDD packages) with a dynamic cache (computed table) management algorithm is give in Fig. 1.

In this algorithm, the codes from lines 1 to 8 is for the computed table management, and the remainder codes constitute the classic algorithm of *ite* operation. Number *ccc* (its initial value is 0) is the current counter of *ite* operation triggered, and number *occ* (its initial value is equal to the initial size of the computed table) serves as the threshold of *ccc* for triggering the cache management algorithm. Number *cts* is the size of computed table and *nc* is the count of the nodes. Number *cchr* is the current *hit-rate* of the computed table, and *ochr* is the last *hit-rate* of the computed table and it is initialized to 0. We also note that $limitedValue$ is the max value which can be reached by the size of the computed table, and it can be determined by the user or the BDD package according to the memory resource.

The cache management algorithm is triggered upon *occ* times running of *ite* operation. The code in line 4 is to increase the cache size, by which CacBDD doubles the size of the current cache. Accordingly, in the code of line 7, *occ* is assigned the **2** times of *ccc*. Clearly, by varying the constant **2**, we can control the frequency of triggering cache resizing. In the current version of CacBDD, the constant is set to 2 by preliminary experiments, although it can be further tuned in the future study.

Moreover, for unary operations, if the available memory is enough, then a temporary complete operation hash is utilized by allocating a continuous array of memory of integer whose count is equal to the count of nodes. When the operation is finished, the memory block is released.

Finally, if the available memory is not enough for new added nodes, then the cache size is to shrink (by half) and the memory space is released for new added nodes. Note that the information of available physical memory can be readily obtained from modern operating systems.

2.2 Garbage Collection

Garbage collection is also a space and time tradeoff issue. In the classic BDD packages, the garbage collection is usually triggered based solely on the percentage of the dead nodes. However, the high rebirth rate indicates that garbage collection should be delayed as long as possible in some applications, such as model checking. Therefore, in CacBDD, we use a simple garbage collection triggering condition: if the free physical memory is nearly used up, then garbage collection is triggered. It is easy to see that the garbage collection triggering is almost delayed to the maximal extent.

Instance	CUDD		CacBDD		CacBDD_Fix		CUDD/CacBDD		CacBDD_Fix/CacBDD	
	Time	Mem	Time	Mem	Time	Mem	TR	MR	TR	MR
c2670	14.4	309.0	**4.4**	277.1	**4.4**	277.1	3.27	1.12	1.0	1.0
c3540	11.7	527.0	**5.0**	360.9	5.3	424.9	2.34	1.46	1.06	1.18
c6288-10	0.3	38.6	**0.1**	21.6	**0.1**	22.1	3.0	1.79	1.0	1.02
c6288-11	0.9	81.5	**0.5**	49.9	**0.5**	65.9	1.8	1.63	1.0	1.32
c6288-12	3.6	209.7	**2.3**	160.2	**2.3**	160.2	1.57	1.31	1.0	1.0
c6288-13	12.9	576.8	**9.7**	418.6	9.8	546.6	1.33	1.38	1.01	1.31
c6288-14	43.9	1665	**33.2**	1331.2	33.5	1331.2	1.32	1.25	1.01	1.0
c6288-15	142.2	4806.1	**103.1**	3474.4	113.4	4498.4	1.38	1.38	1.10	1.29
c6288-16	456.7	13822.8	**337.8**	10966.7	373.5	10966.7	1.35	1.26	1.11	1.0
total (1-9)	686.6	22036.5	496.1	17060.6	542.8	18293.1	1.38	1.29	1.09	1.07
abp11	**7.9**	53.5	8.5	56.8	10.9	536.8	0.93	0.94	1.28	9.45
dartes	2.0	44.4	**1.4**	81.2	1.8	25.2	1.43	0.55	1.29	0.31
dme2-16	241.7	158.2	**51.4**	343.0	121.5	335	4.70	0.46	2.36	0.98
dpd75	565.4	187.2	35.0	647.3	**34.5**	263.3	16.15	0.29	0.99	0.41
ftp3	42.0	292.4	**18.9**	570.9	34.6	318.9	2.22	0.51	1.83	0.56
furnace17	430.4	122.7	**10.3**	376.0	1274.3	128	41.79	0.33	123.72	0.34
futurebus	**255.3**	372.7	546.7	691.6	1504.2	183.6	0.47	0.54	2.75	0.27
key10	48.2	173.0	**12.9**	400.7	37.5	148.7	3.74	0.43	2.91	0.37
mmgt20	344.8	122.1	**13.9**	357.5	14.3	165.5	24.81	0.34	1.03	0.46
motors-stuck	**4.1**	64.6	7.8	58.7	10.3	30.7	0.53	1.10	1.32	0.52
over12	101.8	303.7	**24.7**	421.2	133.2	169.2	4.12	0.72	5.39	0.40
phone-async	134.8	837.9	**60.7**	1088.1	79.5	864.1	2.22	0.77	1.31	0.79
phone-sync-CW	997.8	3895.6	**825.0**	4384.9	831.3	11552.9	1.21	0.89	1.01	2.63
tcas	334.2	4878.7	182.4	3608.9	**179.7**	3224.9	1.83	1.35	0.99	0.89
tomasulo	596.1	4403.3	**188.0**	2779.3	199.5	2779.3	3.17	1.58	1.06	1.0
valves-gates	**3.0**	68.3	5.6	45.3	6.9	33.3	0.54	1.51	1.23	0.74
total (10-25)	4109.5	15978.3	1993.2	15911.4	4474	20759.4	2.06	1.00	2.24	1.30
total	4796.1	38014.8	2489.3	32972	5016.8	39052.5	1.93	1.15	2.02	1.18

Table 1. Comparisons with CUDD and CacBDD_Fix

3 Experimental Results

In this section, we present the experimental results, in order to demonstrate the efficiency of CacBDD and the effectiveness of the dynamic cache management algorithm in CacBDD. We compare CacBDD with CUDD (version 2.5.0), a pointer-based and publicly available BDD package. CUDD is one of the most efficient BDD packages [7, 10], and perhaps the most widely used open source package. Most importantly, CUDD is constantly updated by its author over years. To demonstrate the effectiveness of the dynamic cache management algorithm in CacBDD, we also compare CacBDD with a slightly modified version of CacBDD, called CacBDD_Fix. CacBDD_Fix is the same as CacBDD, except for that it replaces the cache management method in CacBDD with the one used by CUDD.

The benchmarks used are ISCAS85 and smv-bdd-traces98, which are representative in combinational circuits and model checking, respectively [8]. Note that the main characteristic of ISCAS85 is that all the benchmarks in it have almost the same hit-rate (nearly to 0.5) of the computed table. In contrast, the benchmarks of smv-bdd-traces98 come from different models and have different hit-rates.

The experiments are carried out on a PC workstation (Linux 64, Intel Xeon 2.80GHz CPU and 16GB RAM). We do not report the cases that are either too small ($<$ 0.1 CPU seconds) or too large ($>$ 16 GB of memory requirement). For fair comparison, the initial cache size is set to the same value 2^{18}. The variable order used follows the order of appearance in the file.

Table 1 reports the runtime and memory comparisons with the CUDD and CacBDD_Fix. The runtime is measured in seconds and the memory in M Bytes. TR represents the time ratio and MR the memory ratio in the table.

The experimental results show that CacBDD is more efficient than CUDD. As shown in Table 1, for all the benchmarks of ISCAS85, CacBDD is more efficient and consumes less memory than CUDD. For the benchmarks of smv-bdd-traces98, we can see that the runtime of CUDD is about as twice as that of CacBDD, while the memory usage of CUDD is almost the same for CacBDD. Especially, for *dpd75*, *furnace17*, and *mmgt20*, CacBDD achieves tens of times speedup over CUDD.

The experiments also demonstrate that the dynamic cache management in CacBDD is effective. The overall runtime performance of CacBDD is about two times better than that of CacBDD_Fix. Table 1 indicates that CacBDD is consistently better than CacBDD_Fix in both time and memory dimensions on the benchmarks of ISCAS85. For most benchmarks of smv-bdd-traces98, the time consumption of CacBDD_Fix is several times more than that of CacBDD. In particular, for *furnace17* CacBDD is 100 times faster than CacBDD_Fix.

4 Conclusion

In this paper, we presented a new BDD package (CacBDD) with a dynamic method for managing computed table. Our experimental results show that CacBDD outperforms the state-of-the-art package CUDD.

We believe that the efficiency of BDD packages can be further improved, though it seems that the classical techniques for efficient implementation of BDD packages have been mature for many years. The proposed dynamic method for computed table management can be integrated into the other existing BDD packages as future work. Also, we would like to investigate some other techniques used in BDD packages, such as hash function.

Acknowledgement This work was Supported by ARC grants FT0991785 and DP120102489, 973 Program 2010CB328103, National Natural Science Foundation of China (61073033, 61003056 and 60903054), and Open Project SYSK-F1003 of State Key Laboratory of Computer Science. We would like to thank the anonymous reviewers for their insightful comments, and Shaowei Cai and Chuan Luo for their valuable help.

References

1. Bryant R E.: GraphBased Algorithms for Boolean Function Manipulation. IEEE Trans. Computers, 1986, 677–691.
2. Brace K S, Rudell R L, Bryant R E.: Efficient Implementation of a BDD Package. Proc. 27th DAC, 1990, 40–45, 1990.
3. Somenzi F.: CUDD: CU Decision Diagram Package Release. `http://vlsi.colorado` edu/ fabioi/.
4. Long D E.: The Design of a Cache-Friendly BDD Library. International Conference on Computer-Aided Design (ICCAD98), 1998, 639–645.
5. Geert Janssen. Design of a Pointerless BDD Package. In Workshop handouts 10th IWLS, 2001, pp. 310–315.
6. Guanfeng Lv, Kaile Su, Qingliang Chen,Yao Chen, Yachao Feng.: A Succinct and Efficient Implementation of a 2^{32} BDD Package, The Sixth International Symposium on Theoretical Aspects of Software Engineering (TASE2012), Beijing, China, 2012.
7. Geert Janssen.: A Consumer Report on BDD Packages. Proceedings of the 16th Symposium on Integrated Circuits and Systems Design (SBCCI03). Sao Paulo, Brazil, 2003.
8. The BDD benchmark. At URL: `http://www.cs.cmu.edu/~bwolen/software/`
9. Somenzi F.: Efficientmanipulation of decision diagrams. International Journal on Software Tools for Technology Transfer, 3: 17–181, 2001.
10. Bwolen Yang, Randall E. Bryant, David R., OHallaron, Armin Biere, Olivier Coudert, Geert Janssen, Rajeev Ranjan, Fabio Somenzi.: A Study of BDD Performance in Model Checking. in Proc. FMCAD 1998, pp. 255–289.
11. A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella.: NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In Proc. CAV 2002, pp. 241–268, Springer Verlag, 2002.
12. Amir Pnueli, Yaniv Saar and Lenore D. Zuck.: Jtlv: A Framework for Developing Verification Algorithms. In Proc.CAV 2010, pp.171–174, Springer.
13. Rudell, R.: Dynamic Variable Reordering for Ordered Binary Diagrams. In Proc.ICCAD 1993, pp.139–144.
14. Armin Biere: ABCD, `http://fmv.jku.at/abcd/`