A Novel Concurrent Cache-friendly Binary Decision Diagram Construction For Multi-Core Platforms

Mahmoud Elbayoumi
ECE Dept., Virginia Tech
Blacksburg,VA 24061-0002
Email: mbayoumi@vt.edu

Michael S. Hsiao
ECE Dept., Virginia Tech
Blacksburg,VA 24061-0002
Email: hsiao@vt.edu

Mustafa ElNainay
CSE Dept., Alexandria University
Alexandria, Egypt 21544
Email: ymustafa@alex.edu.eg

***Abstract*—Currently, BDD packages such as CUDD depend on chained hash tables. Although they are efficient in terms of memory usage, they exhibit poor cache performance due to dynamic allocation and indirections of data. Moreover, they are less appealing for concurrent environments as they need thread-safe garbage collectors. Furthermore, to take advantage of the benefits from multi-core platforms, it is best to re-engineer the underlying algorithms, such as whether traditional depth-first search (DFS) construction, breadth-first search (BFS) construction, or a hybrid BFS with DFS would be best. In this paper, we introduce a novel BDD package friendly to multicore platforms that builds on a number of heuristics. Firstly, we re-structure the Unique Table (UT) using a concurrency-friendly Hopscotch hashing to improve caching performance. Secondly, we re-engineer the BFS Queues with hopscotch hashing. Thirdly, we propose a novel technique to utilize BFS Queues to simultaneously work as a Computed Table (CT). Finally, we propose a novel incremental Mark-Sweep Garbage Collector (GC). We report results for both BFS and hybrid BFS-DFS construction methods. With these techniques, even with a single-threaded BDD, we were able to achieve a speedup of up to 8× compared to a conventional single-threaded CUDD package. When two-threads are launched, another 1.5× speedup is obtained.**

I. Introduction

Since its introduction in 1986 [1], efficient constructions of Reduced Ordered Binary Decision Diagrams (ROBDD) have penetrated many areas of computer aided VLSI design, including fault simulation, circuit synthesis, ATPG [2], circuit verification, just to name a few. Three major approaches have been proposed to construct ROBDDs: depth-first search (DFS), breadth-first search (BFS) and hybrid BFS-DFS [3]. Although DFS has a low associated memory overheads and large potential to be optimized (i.e., by using Computed Tables (CT)), it has poor memory locality. Meanwhile, BFS construction approaches preserve memory locality and have potential to be parallelized; however, it has associated overheads in storing temporary nodes (stored in queues).

Nowadays, computing has seen a tremendous shift toward parallel computing. This shift is mainly because of three reasons [4]. First, operating frequency has hit a wall. Secondly, Instruction Level Parallelism (ILP) has reached its limit. Thirdly, the degradation due to memory access time outweighs any improvement in processor operating frequency. As a result, researchers are forced to look for alternatives to these traditional approaches for increasing performance. Increasing the number of processors (cores) on a die to improve the performance has served well [5]. As we have new multi-core architectures, we need to investigate and develop new algorithms to handle BDDs on these architectures to efficiently utilize these new low-cost off-shelf devices [6].

Recently, concurrent hashing techniques (e.g., Hopscotch hashing [7]) have been proposed to guarantee constant lookup and deletion times in the hash table. In addition, it shows a superior performance on traditional hashing techniques; i.e., chained hashing, linear hashing and Cuckoo hashing [8], even when the hash table is 90% full. It depends mainly on keeping those nodes, hashed to the same bucket, in a restricted nearby space of the memory. In doing so, it has benefits of low access time of the main memory cache, and guarantee constant worst-case lookup time. This can play a critical role in improving the performance in the access of BDD nodes, which are typically stored and indexed via Hash tables. Thus, in a sense, while construction algorithms for BDDs are extremely difficult to parallelize, we exploit concurrency in hashing algorithms, with tremendous payoffs as will be shown in the results.

The contributions of the paper are as follows: first, we propose to use Hopscotch hashing as a Unique Table (UT) and Queues. Our results show that by utilizing this hashing technique alone on a single core, we were able to achieve a speedup up to 8× compared with CUDD [9]. Moreover, we achieve another 1.5× on average with two threads on a 2-core multiprocessor. Note that the proposed Hopscotch hashing is concurrency-friendly, leading to the speedup with additional threads. Second, we propose to utilize Hopscotch hashing to be used in both queue and the computed table. This would be, as far as we know, the first introduction of a hybrid BFS-DFS approach to the construction of computed tables in Binary Decision Diagrams. We can speed up runs that utilize very small queue sizes, such that the running times are comparable to those that use very large queue size. In other words, we can trade memory consumption with running time. Finally, we propose a novel incremental Mark-Sweep Garbage collector (GC) to further enhance the performance. We implemented the multi-threaded BDD construction using both BFS and a hybrid BFS-DFS and report their results.

The rest of the paper is organized as follows: In the next section, we introduce the some of concepts in Hopscotch hashing technique. In the Third section, we describe the detailed Components of our framework, and we will present

978-3-9815370-0-0/DATE13/©2013 EDAA

our DFS-BFS hybrid approach, including the computed table, and incremental Mark-Sweep Garbage Collector. In the forth section, we present our experimental results. Finally, we conclude the work in the fifth section.

II. Hopscotch Hashing

Hopscotch hashing is a recently proposed hashing technique [7]. It is based on multi-phase probing displacement techniques. Hopscotch hashing preserves and utilizes data locality, hence it has been shown to outperform all other well-known hashing techniques, including chaining, cuckoo hashing, and linear probing. Moreover, it also guarantees a fixed worst case fetching time. We leave the details of Hopscotch hashing implementation and concurrency to [7].

The unique table (UT) is implemented as a hash table using the Hopscotch hashing technique. To preserve data locality, it is represented by an array of buckets. The initial size of the UT is selected to be a power of two, in order to improve the performance of UT resizing operation.

Our package is a pointer-free package; that is, the pointers of the BDD nodes are not the physical memory addresses of the node. We take this approach for mainly two reasons. First of all, it is to provide a platform-independent implementation. Secondly, Hopscotch hashing usually swaps nodes between entries. So, physical-address pointers will add more overhead due to the need to update pointers in each swap.

When a new node is inserted into the UT, it searches for other nodes within the bucket first, and assigns a new local pointer to the new node. When UT is not able to insert a new node within its bucket neighborhood, $resize()$ is called.

In order to save time and avoid memory explosion, we propose an incremental resizing technique for our UT. As depicted in Fig. 1, $resize()$ Method consists of three phases. The first and third phases are done by the $Master$ thread, while the second thread is processed by both $Master$ and $Slaves$ threads. When $resize()$ is called, the first thread call is considered as the $Master$ thread, and any other thread executed by the Master is called a $Slaves$ thread. In the first phase (see Fig. 1, lines 2-7), the $Master$ will allocate a continuous memory block with the size identical to the size of the original UT. Note that any other $Slave$ thread that enters while $Master$ thread is executing the first phase, will be blocked while trying to acquire the lock (line 1 of Fig. 1).

In the second phase, all $Master$ and $Slave$ threads are cooperating in table rehashing. Each thread will take a segment and rehash it until all segments are rehashed (lines 8-12 of Fig. 1). Note that each pointer of any node will not change due to UT resizing. This is because that the UT stores the hash string instead of the hash values. In addition, by restricting the size of UT to be a multiple of two, any node will be rehashed to the same location or will be shifted by the old size of the table. In the third phase (lines 14-20 in Fig. 1), all $Slave$ threads will sleep until the $Master$ finishes.

III. Package Framework

The overall package Framework is illustrated in Fig. 2. It consists mainly of 1) $Manager$, 2) $Workers$, 3) UT, and $Queues$.

```
1: resize()
2: acquire master lock.
3: if this is the master thread then
4:     acquire lock on all segments.
5:     allocates new space, updates necessary parameters variables
6: end if
7: release master lock.
8: while there is a segment doesn't rehashed yet do
9:     for all i such that i ∈ current segment do
10:        rehash node at location i.
11:    end for
12: end while
13: acquire master lock.
14: if this is the master thread then
15:    release all segments lock
16:    notify all sleeping threads
17: else
18:    sleep
19: end if
20: release master lock.
```

Fig. 1. *resize()* method.

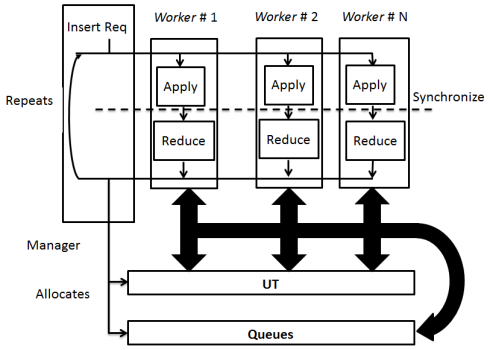

Fig. 2. $Manager$ will allocate UT and $Queues$, insert BDD requests, then creates $N - worker$ threads. $Worker$ threads will execute $apply()$ method concurrently. $Manager$ will synchronize between $worker$ threads until all threads finish executing $apply()$; Then, $worker$ threads execute $reduce()$ method. These operations repeat until all circuit gates are constructed.

A. Manager

As depicted in Fig. 3, $manager$ is responsible for allocation and initiation of all necessary components to build a BDD from a netlist circuit (lines 2-4). It schedules BDD request into queues (Line 6). In addition, it synchronizes between workers (lines 8 & 9).

B. Worker Threads

$Worker$ threads are responsible of performing BDD basic construction operations as depicted in Fig. 4. Fig. 2 depicts how $Manager$ synchronizes among $workers$. $manager$ waits until all $workers$ finish the $apply()$ method (line 4 in Fig. 4), then it allows $workers$ to begin in $reduce()$ method (lines 6 in Fig. 4).

```
1: Read circuit netlist.
2: Initialize and allocate UT, Queues, and other internal
   variables.
3: Create worker threads.
4: for all circuit levels do
5:    insert available BDD request for this level.
6:    while there is a scheduled request in this level do
7:       wait until workers finish executing apply() method.
8:       wait until workers finish executing reduce()
         method.
9:    end while
10: end for
```

Fig. 3. $Manager$ main method.

```
1: loop
2:    while there is a scheduled request in this level do
3:       apply().
4:       wait until all other workers finish apply() method.
5:       reduce().
6:       wait until all other workers finish reduce() method.
7:    end while
8: end loop
```

Fig. 4. $Worker$ main method.

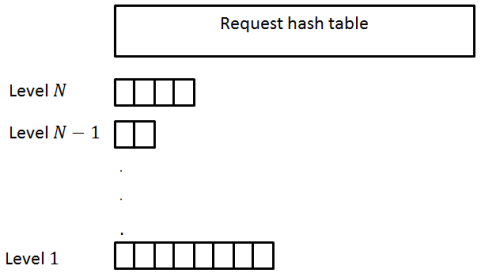

Fig. 5. Queues Structure: Queues consist of one large hash table (implemented with Hopscotch hashing), and array of N lists, where N is number of levels. Each list contains the pointers of the request in hash table corresponding to certain level.

C. Queues and Computed Tables

While $worker$ threads construct BDDs, $apply()$ and $reduce()$ methods require to access temporary request often. In order to have an efficient memory access, we propose to implement $Queues$ as a hash Table and an array of lists as depicted in Fig. 5. Array of lists is used to keep track of every request on each level.

Hash table utilization in implementing $Queues$ have another motivation. We can utilize the hash table as a CT. Dated requests remains in the hash Table as long as a new request needed to be stored in the same location. Accordingly, if a request ($R = \{F, G, H\}$) is previously evaluated and still in the hash table, Its forwarded pointer is fetch.

D. Garbage Collectors

Garbage Collector (GC) is based on a lock-free Mark-and-Sweep approach. We use two bits to represent a mark. Any node can be marked as 1) a permanent node, 2) an updated node, or 3) a dated node. Permanent nodes has its unique mark. We use two other marks to represent updated and dated marks. Before a level is constructed, mark is set and all nodes represent gates in circuit are marked as Permanent. All new nodes inserted will be marked with the Updated Mark. Any node that has a Dated Mark may be overwritten with a new nodes if it is needed to be inserted in the same location of the old node.

IV. Experimental Results

The proposed package is implemented in C++ and is tested with a $BDDtest$. $BDDtest$ creates a $Manager$, which in sequence creates $Workers$ and allocates UT, $Queues$, and other necessary data. $BDDtest$ also create BDDs for every gate in the circuit. For sequential circuits (Table II), $BDDtest$ Builds the transition relation of the circuit (which takes a longer time for construction). The transition relation for the circuit is the conjunction of all transition relations for each state element s_i, i.e., $(\bigwedge_{\forall i} s_i \oplus \delta_i)$. The experiments were run on Core 2 machine with 4 GB of RAM and Ubuntu as the Operating system. CUDD 2.4.2 [9] is used as comparison. We report results on a number of circuits from ISCAS85, ISCAS89, and ITC99 to test our proposed BDD construction. The results are reported in Tables I & II. Table I reports the results for combination circuits. Table II reports the results of creating the transition relation for sequential circuits.

In Tables I & II, for each circuit, we first report the execution time taken by CUDD, followed by our package with BR-1 (Basic Run with 1 thread). We define Basic Run as a run that does not include UT resizing, CT utilization, BFS-DFS hybrid approach, nor GC. In other words, BR-1 is a BFS without UT resizing and CT is not utilized. BR-1 is followed by BR-2 (Basic run with 2 threads). Note that starting from column three, 2 threads are used. The fourth column (Resizing) reports BR-2 with 6 resizing operations. The fifth column (BR-CT) reports the BR with computed table. The sixth column (HBR) reports the DFS-BFS hybrid approach with Queue size ranging from 1 to 40% (percentage depends on the circuit). The seventh column (HBR-CT) reports the time for DFS-BFS hybrid approach with CT (we use the same configuration as in column six). Finally, the eighth column (GC) reports the time of BR-2 with Garbage collection.

According to Tables I & II, our base approach (with BR-1) achieved a speedup ranging from 2× to 8× compared with CUDD. When we use two threads (BR-2), we achieved another 1.5× speedup on average. For example, consider circuit b12rst_1_3_new_s, the original CUDD took 714 seconds to construct the transition relation for this circuit, and our BR-1 took only 88 seconds. This is a speedup of 8.11×. BR-2 reduces the time further to 64 seconds. In a few cases, having two threads allowed us to achieve nearly 2× speedup, such as s298_1_2_new_s, where the execution time was reduced from 1585 seconds to 847 seconds. When resizing is performed for 6 times, we still obtained speedups higher than BR-1. Accordingly, we can conclude that, resizing has a little impact

TABLE I
EXPERIMENTAL RESULTS - COMBINATIONAL CIRCUITS

Circuit	CUDD	BR-1	BR-2	Resizing	BR-CT	HBR	HBR-CT	GC
c432	0.455	0.411	0.277	0.386	0.246	0.510	0.300	0.381
c1355	33.206	7.929	5.030	6.803	4.888	114.722	20.737	5.785
c1908	11.542	1.622	1.037	1.292	1.001	4.980	1.963	1.759
c5315	16.364	2.911	2.430	2.593	2.144	28.612	12.952	4.234

TABLE II
EXPERIMENTAL RESULTS - TRANSITION RELATION CONSTRUCTION OF SEQUENTIAL CIRCUITS

Circuit	CUDD	BR-1	BR-2	Resizing	BR-CT	HBR	HBR-CT	GC
s298_1_2_new_s	2822.054	1585.420	847.820	950.152	838.595	10159.394	1012.426	1044.907
s298_2_3_new_s	2202.240	646.007	437.670	525.209	365.114	875.34	529.580	660.738
s298_2_4_new_s*	2760.862	395.538	228.635	274.362	270.047	581.647	278.203	353.009
s400_1_2_new_s*	1071.225	369.387	275.662	281.214	270.169	1498.289	380.773	342.889
s444_1_2_new_s*	2430.267	335.730	183.316	247.400	167.586	786.226	266.734	268.723
b12rst_1_2_new_s*	214.134	49.568	30.872	31.180	29.972	401.723	150.084	69.135
b12rst_1_3_new_s*	714.319	88.180	64.116	76.936	64.051	268.620	67.382	100.389
b12rst_1_4_new_s*	734.405	120.394	84.605	88.836	70.514	338.461	105.426	155.959
b12rst_1_5_new_s*	680.815	94.100	65.670	81.672	53.674	248.025	84.251	82.259
b12rst_2_3_new_s*	853.914	173.047	120.226	146.386	97.805	819.007	321.896	194.499
b12rst_2_4_new_s*	864.662	108.973	78.436	114.042	76.944	231.484	151.054	110.852
b12rst_2_5_new_s*	913.505	174.565	107.600	141.180	90.129	933.179	270.986	124.164

(*) only a portion of the circuit is tested, not the whole circuit.

on the performance, because we utilize all threads to perform the resizing operation. When comparing the results under columns HBR and HBR-CT, we observed that utilizing the CT with a hybrid approach provides a large improvement in the performance when DFS-BFS hybrid approach is exploited. This is because many requests have been replicated during DFS-BFS hybrid, and hence, CT becomes very vital. When we use the DFS-BFS hybrid approach (column 6, HBR), the performance is degraded compared to BR-2 in all cases. Also, it degraded in most of cases compared to CUDD (i.e., s298_1_2_new_s), since no computed table is used (while CT is used in CUDD). However, when we use DFS-BFS hybrid approach with Computed Table (HBR-CT), the performance is enhanced. For example, in circuit b12rst_2_5_new_s, although the performance of HBR-CT is degraded by 2.52 compared with BR-2, we obtained 3.37× speed-up compared with CUDD and 3.44× over the one with CT (column HBR). Finally, garbage collection incurs some overhead, but can be useful when memory usage is high, as in b12rst_1_2.

In all circuits, our results show that the BFS based construction of the BDDs, as opposed to hybrid BFS-DFS, achieves superior performance on multi-core platforms. Of course, these depend on the type of hashing algorithms used.

V. CONCLUSION

We introduced a novel cache-friendly Multi-threaded BDD Package to construct and manipulate ROBDDs on a multicore platform. As BDD algorithms are memory intensive, maintaining locality of data is important to reduce cost of memory loads and stores. Furthermore, our algorithm offers concurrency to enhance performance on multi-core platforms. We propose the usage of concurrent Hopscotch hashing technique for both the Unique Table and the BFS Queues to improve the performance of BDD construction. Hopscotch hashing not only improves the locality of the manipulating data, but also provides a way to cache recently performed BDD operation. Moreover, it is concurrency friendly. Consequently, the time and space usage can be traded off. With our approach, even with a single-threaded implementation, we were able to achieve a speedup of up to 8× compared to a conventional single-threaded CUDD package. When two-threads are launched, another 1.5× speedup is obtained. Future work includes other concurrency-friendly hashing algorithms.

REFERENCES

[1] R. E. Bryant, Graph-based algorithms for Boolean function manipulation, IEEE Trans. Comput., August 1986, volume : 35 : 8. pages 677-691.

[2] D. Tille, S. Eggersgluss, R. Krenz-Baath, J. Schloeffel, R. Drechsler, Improving CNF representations in SAT-based ATPG for industrial circuits using BDDs, Test Symposium (ETS), 2010 15th IEEE European , vol., no., pp.176-181, 24-28 May 2010.

[3] B. Yang, Y.-A. Chen , R.E. Bryant and D.R. O'Hallaron, Space- and time-efficient BDD construction via working set control , Design Automation Conference 1998. Proceedings of the ASP-DAC '98. Asia and South Pacific , vol., no., pp.423-432, 10-13 Feb 1998.

[4] J. L. Hennessy, D. A. Patterson, Computer Architecture: A Quantitative Approach, Morgan Kaufmann, 4th Edition, 2006.

[5] J. Sanders, E. Kandrot, CUDA by Example: an Introduction to General Purpose GPU Programming, Addison Wesley, 1st Edition, 2010.

[6] S. Stergios and J. Jawahar. Novel applications of a compact binary decision diagram library to important industrial problems. Fujitsu scientific and technical journal, 46(1):111–119, 2010.

[7] M. Herlihy, N. Shavit and M. Tzafrir, Hopscotch Hashing, Lecture Notes in Computer Science, Springer Berlin / Heidelberg,p350-364,volume: 5218, 2008.

[8] R. Pagh, F. F. Rodler, Cuckoo hashing, Journal of Algorithms, Volume 51, Issue 2, May 2004, Pages 122-144, ISSN 0196-6774, 10.1016/j.jalgor.2003.12.002.

[9] http://vlsi.colorado.edu/ fabio/CUDD/ (last visited April 16, 2012).