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 $8\times$ compared to a conventional single-threaded CUDD package. When two-threads are launched, another $1.5\times$ 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 $8\times$ compared with CUDD [9]. Moreover, we achieve another $1.5\times$ 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
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 Slaves 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 Slaves 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 Slaves 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

```c
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.](image1)

**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.

**Queues.**

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: reduce().
5: while all other workers finish apply() method.
6: reduce().
7: end while
8: end loop

Fig. 4. Worker main method.

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 forward 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., \((\bigwedge_{i \in S} s_i \oplus \delta_i)\). 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 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 BFS-DFS 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 (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 \(2 \times \) to \(8 \times \) compared with CUDD. When we use two threads (BR-2), we achieved another \(1.5 \times \) 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 \(8.11 \times \). BR-2 reduces the time further to 64 seconds. In a few cases, having two threads allowed us to achieve nearly \(2 \times \) 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.
Finally, garbage collection incurs some overhead, but can be compared with BR-2, we obtained 3.37 although the performance of HBR-CT is degraded by 2.52. 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., \( \text{s}400 \_1 \_2 \_new \_s \)), 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 \( \text{b}12\text{rst} \_1 \_2 \_new \_s \_s \), 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 \( \text{b}12\text{rst} \_1 \_2 \_new \_s \).

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