Caching in Parallel BDD Package

Igor Grudenić, Nikola Bogunović
Faculty of Electrical Engineering and Computing, University of Zagreb
Unska 3, 10000 Zagreb, Croatia
{igor.grudenic, nikola.bogunovic}@fer.hr

Abstract. *Fast computation of Binary Decision Diagram (BDD) operations is essential for efficient symbolic model checking. In order to improve symbolic model checking we parallelized BDD operations on network of workstations (NOW). Sequential BDD packages have always been sensitive to cache design, which is even more true in the parallel environment. Distribution of cache to the NOW made larger cache possible, but it implicated problems with references in distributed environment and posed limitations on ordering of computation. In this paper we address these issues with our distributed cache architecture.*

Keywords. Distributed BDD computation, cache management, formal methods.

1. Introduction

Representation of logical functions and their efficient manipulation is the heart of many formal verification methods, especially model checking [2]. BDDs are commonly used to represent logical functions because of its two basic properties: canonicity and compactness. Since formal verification is essential for complex systems, the uncontrolled growth of its BDD encoding is an open problem.

Parallelization of BDD representation and management is used in [5][8][6][3] to address space and time complexity of formal verification. BDD management parallelization is realized for multiprocessor systems and for networks of workstations (NOWs). Introducing cache system in the BDD management delivers performance improvement that justifies its use [1]. In this paper we address cache issues in parallel BDD implementation [3], which runs on NOW.

In Section 2 we outline an algorithm for BDD manipulation. Cache distribution and data organization are presented in section 3. In section 4 we describe different cache issues such as hashing function, number of caches and cache size. The results, which contain speedups and cache hit rates are given in section 5.

2. Parallel BDD algorithm

BDD is a directed acyclic graph, where vertices represent Boolean variables and arcs represent different variable interpretations. Sinks in the graph denote Boolean function values for the given interpretation. The basics of BDD management are in the following recursive definition of operations:

$$f\langle op\rangle g = x\cdot\left(f_x\langle op\rangle g_x\right)+\bar{x}\cdot\left(f_{\bar{x}}\langle op\rangle g_{\bar{x}}\right) \qquad (1)$$

where f and g are BDD representations of the corresponding Boolean functions, and *op* is the Boolean operation. Cofactors f_x and $f_{\bar{x}}$ refer to the BDDs of f with variable x evaluated to *true* and *false* respectively. The calculation of equation (1) is terminated when one of the terminal cases occur.

Distribution of BDD nodes in the parallel algorithm [3] is based on the variable indexes. Every workstation holds a set of consecutive variable indexes. Nodes are allocated to the workstations in order to maintain equal memory consumption.

A workload for every workstation is defined in terms of a context. Every context holds a number of BDD operations that need to be expanded and reduced. Operations are encoded into the *operationNode* data structures. The expansion is defined in equation (1) and the reduction is the following phase, where the new BDD is checked for isomorphism with the existing BDDs to prevent duplicate BDDs to occur.

During the expansion phase, if the number of newly generated operations is beyond the given threshold, new contexts are created. All the operations for which expansion needs to continue on the other workstations are dispatched after the expansion. The reduction of each of the contexts

can begin only if all operations are expanded, and results came in for all the dispatched operations. The reduction will be delayed if there is any context that needs to be expanded. The order of expansions and reductions is made in first-come first-served manner.

3. Cache system organization

Cache system is very important in BDD management. During the expansion phase, every operation is looked up in the cache before expanding by equation (1). If it is found the result is returned, and if not, the operation is added into the cache. Using this and presuming that an infinite cache is available, the upper bound for the time complexity of the BDD operation *f*<*op*>*g* is given by $|f| \cdot |g|$, where $|f|$ and $|g|$ represent number of vertices in the BDDs.

In the subsection 3.1., cache structure is described. Limitations on the order of the computation are presented in subsection 3.2.

3.1. Cache structure

Cache is internally organized as a number of hash tables without collision chains. We initially designed collision chains which consumed more memory because of the links in the chain. Finding elements in such a chain is a bit slower, especially for cache misses, since all the elements must be visited. Avoiding the use of collision chains results in overwriting the elements with the same hash key and it cancels theoretical bound on the time complexity. In the practice however, the cache without the collision chains is faster because the hashing functions distributes entries nicely for all our benchmarks which include the ISCAS 85 [4].

The structure of the each cache entry is presented in Fig. 1. For the operation *f* <*op*> *g*, *index1* and *index2* represent the lowest variable index in BDDs *f* and *g*. Indexes are needed since

index1	operand1
index2	operand2
opCode	resultP

Figure 1. Cache entry

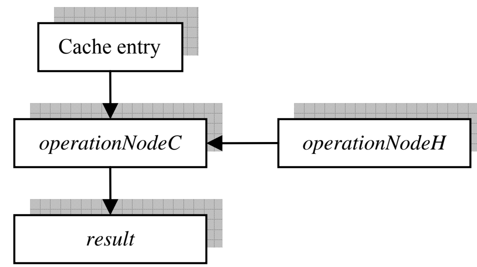

Figure 2. Operation Node forwarding

it's the only mechanism for locating the workstation in which the BDD node is stored. Pointers to the graph structure that correspond to BDDs *f* and *g* are *operand1* and *operand2* (Fig. 1.). The result of the operation is denoted by *resultP*. The lowest variable index of the result is calculated as the minimum of *index1* and *index2* and it is not stored. Since the parallel package [3] is based on calculation of the binary operations, it is necessary to store the code of the Boolean operation (*opCode*) in order to minimize network traffic. The other feasible solution would be to have a different hash table for every Boolean operation.

3.2. The order of computation

Every cache entry is written twice in order to store one operation into the hash table. After the operation's expansion the operands and the pointer to the *operationNode* (as *resultP* in Fig. 1.) are stored into the cache entry. When the reduction of the operation is done, *resultP* is updated with the result of the reduction, which is either a newly created or an existing BDD node.

It is possible that the cache hit occurs before the cache entry is updated with the final result (after reduction). Cache hit in that scenario can be used if the *operationNode* that generated cache entry is reduced before the *operationNode* for which the hit occurred. After the cache entry update, new result must be propagated to all the *operationNodes* that accessed the entry before the update. Such propagation would involve list of such *operationNodes* for each of the non-updated cache entries.

The use of such lists is avoided by node forwarding. After each of the operation nodes is reduced and updated in the cache, it is not freed but forwarded to the result. In Fig.2. *operationNodeC* is the node stored in the cache, which can be reached through the cache entry. After the *operationNodeH* (which is semantically the same as *operationNodeC* and results in a cache hit) is created, it is pointed to the

operationNodeC. When the *operationNodeH* is reduced, it is forwarded to the result, and it is possible to reach that result from the *operationNodeH*. The reduction of the *operationNodeH* is simply the redirection to the result.

In order to achieve a correct order on *operationNode* reduction, it is necessary to care about ordering of the context computation. As noted above, the first-come first-served method is used. After all the expansions are computed, reductions are done in the same context order. Strict ordering of the context reductions may theoretically slow the parallel algorithm. It is possible that the context, which is scheduled for the reduction, can't be reduced because the results from the other workstations are not available yet. In such a scenario algorithm stops and waits for the results, because reduction of other contexts would harm the caching system. In practice, it is rarely a case that other contexts could be reduced, because almost always their reduction depends on the results that hadn't arrived. It is also possible that, while waiting for the other workstations results, new context that needs to be expanded arrives for the computation.

Operations that reside in each of the contexts are handled in the breadth-first manner and the correct reduction ordering within context is preserved.

4. Cache implementation issues

Implementation of hash accessed tables as computed caches, involves choosing the appropriate hash function. This issue is discussed in subsection 4.1. Determination of the cache size is a problem that can drastically affect system performance and is addressed in subsection 4.2. Specifically for BDD management, different number of caches can be introduced. This aspect is outlined in subsection 4.3.

4.1. Hash function

Hash function is used to transform data that is going to be cached into a key, which will be used to determine cache address of the operation.

Input data for the hash function are 32-bit pointers to the BDD operands. Prevalent BDD packages, such as CUDD [7] and PPBF [9], use two different hash functions:

$$f_1(op1, op2) = \left[(op1 \cdot p1 + op2) \cdot p2\right] >> \left[32 - \log_2(CS)\right] \quad (2)$$

$$f_2(op1, op2) = (op1 + op2 >> 4)\ \&\ (CS - 1) \quad (3)$$

The first one, noted by the equation (2), is based on the prime number multiplication, where $p1$ and $p2$ are primes, and $op1$,$op2$ are pointers to the operands. The *CS* stands for cache size which is a power of two, while operator >> is the right shift operator. In order to fit the hash key to the cache address space, operator >> was used to select proper amount of the register's upper bits for addressing. The registers size is 32 bits.

The function f_2, defined by the equation (3), was proposed because the multiplication of large primes takes significant processor time. Given that the computer architecture evolved, multipliers in modern processors are designed using fast look-up tables, which renders the function f_1 more time effective.

Both functions were tested in order to determine "randomness" of the hash function and the time consumption. The hash function f_1 distributes operations more evenly to the cache. As the CPU time is concerned, almost no difference is observed, which shows that the effect of the slower multiplication is neutralized by the better hashing properties of the function f_1.

4.2 Cache size

Cache size in every software system can be fixed or can be determined dynamically during the runtime. Both policies were employed and tested. The fixed size for the cache is taken as the 10% of the systems memory. While employing cache with variable capacity, the size is doubled when certain percentage of the cache is filled.

For large BDD management we discovered that both policies result in the same execution time, since cache size reaches the limit exponentially during the beginning of execution. For moderate size BDDs, the recommended policy is to determine cache size in advance.

If the amount of system memory is not available, or exposed to frequent changes, it is reasonable to determine the cache size dynamically. The limit in that case may be the percentage of memory used by the BDD package. The percentage threshold for cache resize should be chosen to minimize execution time. In Fig. 3. execution times for different thresholds are presented. It is evident that the resize threshold should be set at about 70%.

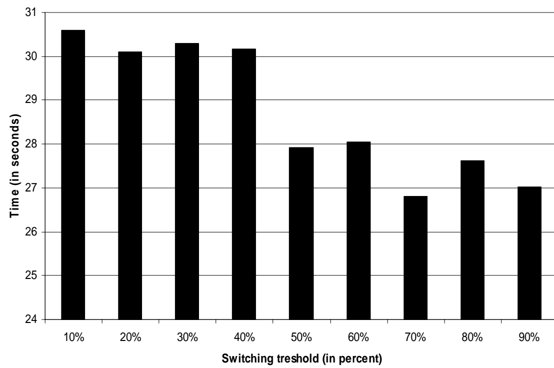

Figure 3. Switching treshold

4.3. The number of caches

BDD operations can be grouped by the type of operation and by the minimum variable index of the involved operands.

It is possible to use different cache for each type of operations, and to set the cache size dynamically as required. This would eliminate the need for the *opCode* in the cache entry structure (Fig. 1.), but would mean a need for maintaining separate caches. Another issue is the distribution of memory capacity among the caches for different operations. We decided to keep all the operations in a single cache because we believe that resizing caches would consume too much of the CPU time. The overhead of memorizing the operation type is solved by the usage of lower operand pointer bits. These bits are not used for addressing because of the implemented custom memory management.

The creation of BDD diagrams, which are exponentially large in the number of variables, revealed the fact that the majority of BDD operations tend to cluster over few variable indexes. Except this few variable indexes, operations involving other indexes occur in a much smaller rate. Opposite to the number of unique operations at each variable index, cache hit rate is the highest for variables with the highest variable index for the most of hardware designs. The number of unique operations, for which the cache hits never occur, is the greatest for the lowest variable indexes. It is possible that the unique operations replace operations with the high hit rates in the cache which could result in the lower global hit rates. Since the uniqueness of the operation usually depends on the variable index, it may be feasible to have a separate cache system for the each variable index.

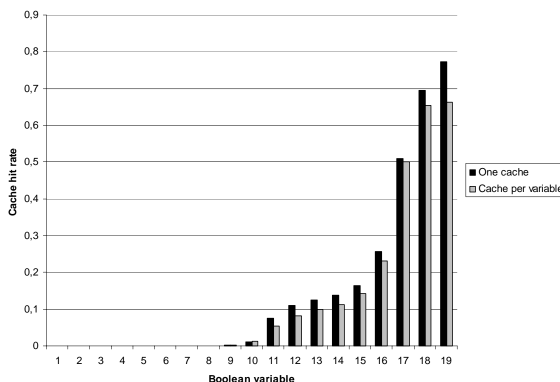

Figure 4. One cache vs. Per-variable cache

Measurements of the difference in the cache hit rate for each variable using one cache system or separate cache systems for each variable are shown in Fig. 4. The BDD built is ISCAS 85 10-bit multiplier and the execution was made on the single workstation.

It can be observed that the cache hit rates for the single cache system are slightly higher for almost all indexes. The reason for this could be in more frequent cache resizes as the result of higher number of caches. Since the cache is resized when it is almost filled up, the number of rewritings and cache misses is higher at that point. The use of different cache for each BDD variable is therefore unnecessary, except in the case of the fixed cache size.

5. Experimental results

Performance speed-up and hit-rates are the best indicators of the cache performance. The performance was tested on multipliers of size 8 to 14 bits (Fig. 5.) with execution time limit of 10 minutes for the 14 bit multiplier. For the purpose of testing, a computing cluster with 2.8 GHz Xeon processors and 2GB of RAM per workstation was used. The computation was always done on different workstations, and 2 CPUs in the same machine were never used.

Fig. 5. presents the ratio of two execution times for a single CPU and a cluster of 2 to 5 CPUs. The nominator is the execution time of the system without cache and the denominator is the execution time of the system with cache. For the 8-10 bit multipliers there is no visible performance gain, since the running times are below 5 seconds and data are statistically irrelevant. The use of the cache system is beneficial when the computation time exceeds 30 seconds (11 bit multiplier and up).

When comparing speedups on the single workstation and on several workstations in parallel, one workstation scenario discloses

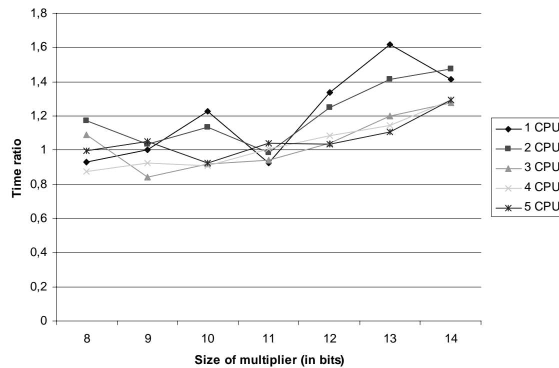

Figure 5. System speedup

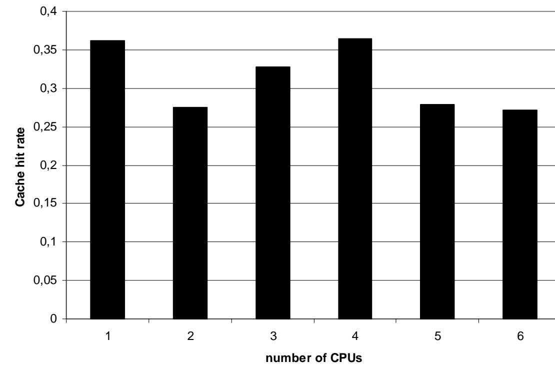

Figure 6. Cache hit rates

better relative performance. The performance is degraded when multiple workstations are used due to the load balancing. When BDD nodes are moved between the workstations the cache entries are not moved but erased to minimize the overhead. This didn't have large effect on the cumulative cache hit rate, as shown in Fig. 6.

The results in Fig. 6. depict cache hit rates for a 13 bit multiplier with cache size preset to 4 million entries. The cumulative cache hit rate is almost constant because the cache size is proportional to the number of processors.

It is expected that, while presuming the same cache hit rate, the speedup for higher number of workstations will be more noticeable. The results in Fig. 5. are in favor of the single workstation computation. The reason for this lies in the processor power needed to keep distributed cache consistent while the parts of BDDs are moving among the workstations.

6. Conclusion

In this paper we addressed cache system issues for parallel BDD package. We analyzed the order of computation that must be followed to keep cache system consistent. Two different cache functions were discussed as well as two cache size policies.

It is shown that cache system significantly improves performance of the parallel BDD package. The improvement is most evident when employed on the single workstation. Performance gain with multiple workstations, compared to the single workstation, falls for about 10% which is the result of the employed load balancing strategy.

In order to make cache more feasible in the distributed environment, further investigations should be performed. Detailed cost analysis of the cache entries transfer between the workstations should be studied in more detail.

References

[1] Brace KS, Rudell RL, Bryant RE. Efficient implementation of a BDD package. Proceedings of the 27th Design Automation Conference; June 1990; Orlando, USA. p. 40-45.

[2] Burch JR, Clarke EM, Long DE, MacMillan KL, Dill DL. Symbolic Model Checking for Sequential Circuit Verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 1994; 4(13): 401-24.

[3] Grudenic I, Bogunovic N. Parallel approaches to BDD manipulation. Proceedings of MIPRO, Computers in technical systems and intelligent systems. 2004; Opatija, Croatia.

[4] Harlow JE. Overview of Popular Benchmark Sets. IEEE Design and Test 2000; 17(3):15-17.

[5] Milvang-Jensen K, Hu AJ. BDDNow: A Parallel BDD Package. Proceedings of FMCAD. 1998 Nov 4-6; Palo Alto, California, USA. p. 501-07.

[6] Sanghavi JV, Ranjan RK, Brayton RK, Sangiovanni-Vincentelli A, High performance BDD package by exploiting memory hierarchy, Proceedings of Design Automaton Conference. 1996 Jun 3-7; Las Vegas, Nevada, USA. p. 635-40.

[7] Somenzi F, CUDD: "CU Decision Diagram Package Release 2.3.0.", University of Colorado at Boulder, 1998.

[8] Stornetta T, Brewer F. Implementation of an Efficient Parallel BDD Package. Proceedings of Design Automaton Conference. 1996 Jun 3-7; Las Vegas, Nevada, USA. p. 641-44.

[9] Yang B, "PPBF parallel BDD package", http://www-2.cs.cmu.edu/~bwolen/software/ppbf/