Copyright © 1996, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

BINARY DECISION DIAGRAMS ON NETWORK OF WORKSTATIONS

by

Jagesh V. Sanghavi, Rajeev K. Ranjan, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M96/9

6 February 1996

BINARY DECISION DIAGRAMS ON NETWORK OF WORKSTATIONS

by

Jagesh V. Sanghavi, Rajeev K. Ranjan, Robert K. Brayton,
and Alberto Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M96/9

6 February 1996

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Binary Decision Diagrams on Network of Workstations

Jagesh V. Sanghavi[*] Rajeev K. Ranjan[†] Robert K. Brayton Alberto Sangiovanni-Vincentelli
Department of Electrical Engg. and Computer Science
University of California at Berkeley
Berkeley, CA 94720

Abstract

The success of all binary decision diagram (BDD) based synthesis and verification algorithms depend on the ability to efficiently manipulate very large BDDs. We present algorithms for manipulation of very large Binary Decision Diagrams (BDDs) on a network of workstations (NOW). A NOW provides a collection of main memories and disks which can be used effectively to create and manipulate very large BDDs. To make efficient use of memory resources of a NOW, while completing execution in a reasonable amount of wall clock time, extension of breadth-first technique is used to manipulate BDDs. BDDs are partitioned such that nodes for a set of consecutive variables are assigned to the same workstation. We present experimental results to demonstrate the capability of such an approach and point towards the potential impact for manipulating very large BDDs.

[*]Supported by Micro Grant
[†]Supported by Motorola and SRC Grants

1 Introduction

The manipulation of boolean functions is one of the most important operations in several areas of computer-aided design such as logic synthesis, testing, checking sequential equivalence, design verification, etc. The efficiency of the logic function manipulations depends on the data structure used for representing boolean functions. The *reduced ordered binary decision diagram (ROBDD)* [1, 5] is a canonical, directed acyclic graph representation of boolean functions. ROBDD (henceforth referred to as BDD) representation is compact for many functions encountered in practice. The canonicity and compactness properties of the BDD led to its widespread usage in the area of logic synthesis and testing. The application of BDD is further extended with its use in symbolic computation, which include symbolic simulation [6], reachability analysis [8, 15], and BDD based formal design verification [4, 7, 11].

However the BDD representation suffers from the drawback that the size of a BDD required to represent a complex logic circuit is very large. This results in large computation and memory requirements. These problems have been tackled on both the fronts.

Reducing the computation time: Kimura *et al.* [10] have presented a parallel algorithm to construct BDDs that uses a shared memory multiprocessor to divide the tasks that can be performed in parallel on several processors. Shared memory machine allows the use of a single global hash table to maintain canonicity. Ochi *et al.* [12] have proposed a breadth-first manipulation approach that uses a vector processor to exploit the high vectorization ratio and long vector lengths by performing a BDD operation on a level-by-level basis.

Increasing the available memory: When the size of a BDD exceeds the main memory, BDD nodes are swapped to the hard disk. The conventional depth-first BDD manipulation algorithm results in random accesses to the memory leading to a large number of page faults. Since a page access time is of the order of tens of milliseconds, a large number of page faults lead to impractical amount of wall clock time, even though the time spent by processor doing useful work is quite small. Ochi *et al.* [13] have proposed the breadth-first implementation approach to regularize the memory accesses, which leads to fewer page faults. As a result, BDDs of very large size (up to 12 million nodes) can be handled. Ashar *et al.* [3] have presented an improved breadth-first algorithm, which enables manipulation of BDDs with up to 100 million nodes.

In this work, we propose a technique to manipulate BDDs on a network of workstations (NOW). A NOW provides a large amount of collective memory resources, both main memories and disks. The collective memory resources of NOW provide a potential to manipulate very large BDDs.

The advantage of our approach as compared to existing ones is two fold. Unlike the approaches in [10, 12], which require special computing hardware (shared multiprocessor or vector processor), a NOW is a part of the existing infrastructure. Secondly, the approaches in [3, 13] are limited by the memory available on a particular machine. When using a network of workstations, the available memory increases significantly.

The rest of the paper is organized as follows. We explain the relevant attributes of the network of workstations in Section 2 and that of the BDD algorithm in Section 3. After explaining the characteristics of the available resources and the algorithmic requirements, we propose a new BDD algorithm on a network of workstations in Section 4. We present the implementation details in Section 5. We present experimental results in Section 6. Finally, we draw some conclusions and outline the direction of the future work in Section 7.

2 Network of Workstations

A network of workstations is a computing resource that uses as its building block, an entire workstation. These building blocks are interconnected by a local area network such as ethernet, FDDI, switched ethernet, or ATM. Using a network of workstations as a large computer system to solve large scale problems is attractive, since it uses the existing infrastructure as opposed to buying a dedicated scalable parallel computer, a server, or a shared memory multiprocessor machine. Further, when the system is upgraded to use faster processors, faster network, larger capacity DRAMs, or larger capacity disks, a network of workstations leverage each of the enhancements.

Let us first understand the nature of NOW computing resource to exploit it fully to match the requirements of BDD algorithms. An existing computing infrastructure with two year old technology may consist of a network of workstations, each with 50 MHz processor, 64KB cache, 64MB main memory, and 200MB of disk space. It takes about 0.1–0.6 microseconds to access data from the main memory and about 6 milliseconds to move a page of memory from the disk to the main memory. The software overhead and latency for a local area network is of the order of about 10 milliseconds and bandwidths are 10 Mbits per second for ethernet and 100 Mbits per second for FDDI network.

It is clear from the above discussion that the time taken to access the data from the disk or from across the network is about 10000–50000 times more than the time to access the data from the main memory. Over the next few years, the networks are expected to become faster [2] in terms of the latency, the software overhead, and the bandwidth. However, the ratio of time to access the remote memory which involves a network transaction vs. the time to access the main memory is still expected to be the order of 1000. This qualitative analysis has important implication when distributing the BDD nodes across the several workstation memories.

For developing distributed BDD algorithms on NOW, the message passing model of computation is assumed for the following reasons: 1) it closely resembles the underlying NOW architecture and 2) easy availability of robust message passing software in the public domain. The message passing programming model makes the cost of communication explicit, however, the programmer has to worry about resource management, sending and receiving messages, and overall orchestration of the collection of processes spread across several workstations.

3 BDD Algorithms

To implement BDD algorithms using the message passing model over a NOW, we need to design distributed BDD data structures. However, it is important to understand the requirements of BDD algorithms on a uniprocessor to help guide our design decision about distributing the data and scheduling the interprocessor communication.

The conventional depth-first recursive BDD manipulation algorithm performs a boolean operation by traversing the operand BDDs on a path-by-path basis (see Figure 1), which results in extremely disorderly memory access pattern. The random memory access pattern with no spatial locality of reference translates into severe page faulting behavior when the BDD does not fit the available main memory.

```
df_op(op,$F$,$G$)
    if (terminal case(op, $F$, $G$)) return result;
    else if (computed table has entry(op, $F$, $G$)) return result;
    else
        let $x$ be the top variable of $F$, $G$;
        $T$ = df_op (op, $F_x$, $G_x$);
        $E$ = df_op (op, $F_{x'}$, $G_{x'}$);
        if ($T$ equals $E$) return $T$;
        result = find or add in the unique table ($x$, $T$, $E$);
        insert in the computed table ((op, $F$, $G$), result);
    endif
return result;
```

Figure 1: Depth-First BDD Manipulation Algorithm

Since the access to the main memory of an another workstation involves a network transaction, the aforementioned disk access behavior of the depth-first algorithm translates to a large number of network transactions for any distribution of the BDD nodes among main memories of a NOW. Since it is very expensive to access the data across the network compared to the workstation main memory, any attempt to use depth-first manipulation algorithm on a NOW will meet limited success.

The breadth-first iterative algorithm [3, 13] (see Figures 2, 3, and 4) attempts to regularize the memory access pattern by traversing the operand BDDs on a level-by-level basis and by using a customized memory allocator that allocates the BDD nodes for a specific variable id from the same page. However, since the result BDD is constructed level-by-level, it is not possible to perform certain isomorphism checks while constructing the BDD. The redundant nodes created during the APPLY phase (see figure 3) in the result BDD have to be eliminated by a bottom-up REDUCE phase (see figure 4). In [3], Ashar *et al.* obviate the need to access the BDD node to determine its variable id by using a lookup table that returns the variable id from the BDD node pointer. The memory access pattern for servicing the REQUEST in APPLY and REDUCE phases is also regularized by processing them in a sorted order.

```
bf_op(op, F, G)
    if terminal case (op, F, G) return result;
    min_id = minimum variable id of (F, G)
    max_id = number of variables
    create a REQUEST (F, G) and insert in REQUEST QUEUE[min_id];
    /* Top down APPLY phase */
    bf_apply(op, min_id, max_id);
    /* Bottom up REDUCE phase */
    bf_reduce(max_id, min_id);
    return REQUEST or the node to which it is forwarded;
```

Figure 2: Breadth-First BDD manipulation algorithm

We make the following observations to guide the implementation of breadth-first search algorithm for a NOW.

1. We need a mechanism to determine the variable id from the BDD node pointer without accessing the BDD node.
2. While processing the REQUEST for a specific variable id during the APPLY phase, we need to access only those BDD nodes that have the same variable id.
3. The forwarding mechanism, which allows temporary creation of redundant nodes can facilitate the creation of a REQUEST on one workstation and servicing of that REQUEST on an another workstation.

4 Binary Decision Diagrams on Network of Workstations

4.1 Issues:

The following issues need to be resoved before we can implement the breadth-first BDD manipulation algorithm on a NOW.

Node Distribution How to distribute the BDD nodes among the workstations on a network? The number of nodes assigned per workstation should be proportional to the memory resources available on the workstation. The high

```
bf_apply(op, min_id, max_id)
    for (id = min_id; id $\leq$ max_id; id++)
        /* process each request queue */
        $x$ is variable with id "id";
        while (REQUEST QUEUE[id] not empty)
            REQUEST ($F$, $G$) = unprocessed request from REQUEST QUEUE[id];
            /* process REQUEST by determining its THEN and ELSE */
            if (NOT terminal case ((op, $F_x$, $G_x$), result))
                            next_id = minimum variable id of ($F_x$, $G_x$)
                            result = find or add ($F_x$, $G_x$) in REQUEST QUEUE[next_id]
            REQUEST $\rightarrow$ THEN = result;
            if(NOT terminal case ((op, $F_{x'}$, $G_{x'}$), result))
                            next_id = minimum variable id of ($F_{x'}$, $G_{x'}$)
                            result = find or add ($F_{x'}$, $G_{x'}$) in REQUEST QUEUE[next_id]
        REQUEST $\rightarrow$ ELSE = result;
```

Figure 3: Breadth-First BDD manipulation algorithm - APPLY

overhead and latency of accessing a remote memory by performing network transaction implies that performing a large number of communications which involve small messages would result in unacceptably high performance penalty. Therefore, a distribution that results in exchanging information at the level of a BDD node would not be satisfactory.

Naming BDD Nodes How to uniquely identify each BDD node regardless of where it resides on the network, i.e., regardless of workstation address space it belongs to? For a single address space, each BDD node is uniquely identified by its pointer; we need to extend the pointer mechanism to have a generalized address for a BDD node.

Variable Id Determination How to determine the variable id of a BDD node given its generalized address? In the breadth-first algorithm, we need to determine the variable id from the BDD "pointer" to avoid random access to the BDD node. However, the BDD node to index lookup table solution proposed by Ashar et al., is unattractive for NOW case for three reasons: 1) each workstation will need to maintain a private copy of the lookup table to determine the variable id from a generalized address for *all* the nodes in the BDD, 2) this private copy will have to be updated every time any workstation allocates a page of memory, and 3) since generalized address would augment 32-bit address space, it may be necessary to implement the node to index lookup table as hash table instead of an array.

We have designed a generalized addressing scheme that works in conjunction with a partitioning scheme to solve the aforementioned problems, while resulting in a very compact representation for the BDD nodes.

4.2 Solutions:

Node Distribution The breadth-first algorithm constructs the result BDD one level at a time by accessing the operand BDD nodes on a level-by-level basis, the natural choice for the decomposition of the BDD is to partition it by levels. To make number of nodes in a partition (BDD section) proportional to the amount of memory resources per workstation, we can use the flexibility of determining the location of and the number of levels in the partition. For example, a BDD section closer to the root nodes can have more levels than a BDD section at the halfway between root and leaf nodes.

Naming BDD Nodes By assigning nodes for a set of consecutive variables to the same workstation, it is possible to determine the workstation on which a BDD node resides by knowing its variable id. Hence a (variable id, memory

```
bf_reduce(max_id, min_id)
    for (id = max_id; id $\geq$ min_id; id- -)
        $x$ is variable with id "id";
        /* process each request queue */
        while (REQUEST QUEUE[id] not empty)
            /* process each request */
            REQUEST ($F$, $G$) = unprocessed REQUEST from REQUEST QUEUE[id];
            if (REQUEST→THEN is forwarded to $T$) REQUEST→ THEN = $T$;
            if (REQUEST→ELSE is forwarded to $E$) REQUEST→ ELSE = $E$;
            if (REQUEST→THEN equals REQUEST→ELSE) forward REQUEST to REQUEST → THEN ;
            else if (BDD node with (REQUEST→ THEN , REQUEST → ELSE ) found in UNIQUE TABLE[id])
                forward REQUEST to that BDD node;
            else
                insert REQUEST to the UNIQUE TABLE[id] with key (REQUEST THEN , REQUEST ELSE )
```

Figure 4: Breadth-First BDD manipulation algorithm - REDUCE

address) tuple can serve as a generalized address that uniquely identifies each node in the BDD.
Variable Id Determination We could have used the pair (workstation number, memory address) to represent a generalized address that uniquely identifies each node in the BDD. However, the reason for choosing (variable id, memory address) tuple to represent the generalized address under the constraint of specific levelized partitioning scheme is to solve the variable id determination problem for free. Further, this choice of generalized address results in very compact representation for a BDD node.

Given the partitioning scheme and the mechanism to determine the variable id, we need to address one more issue before we can perform computations related to the BDD sections assigned to a workstation. Servicing a REQUEST in the APPLY phase may result in creation of an another REQUEST , top variable id for which belongs to another workstation. The newly created REQUEST with a specific top variable id should now be serviced on the workstation that owns the BDD section containing that variable id. REQUEST can be generated on a source workstation and processed on a destination workstation, as long as the source workstation receives a correct generalized address that should result from processing the REQUEST . It is an easy matter to use forwarding mechanism in the APPLY phase for the source workstation by forwarding the generated REQUEST to the generalized address. Since the REQUEST node that gets generated on the source workstation is a *shadow* of the REQUEST node that gets processed on the destination workstation, we call this as *shadow node forwarding*. By using *shadow* nodes, a node which creates the *shadow* node can now be processed in the APPLY phase without accessing the remote memory. Using the same *shadow* node forwarding concept, a set of REQUEST , which belong to the set of consecutive variables assigned to the processor, can be processed without accessing remote memories. The mechanism of *shadow node forwarding* also helps to separate the computation and the communication for the collection of sequential processes. The separation helps simplify the development of the NOW BDD package. The algorithm for manipulation of BDDs on a NOW is presented in Figure 5.

The breadth-first BDD manipulation algorithm on a NOW is obtained by suitable modifications of APPLY and REDUCE phase of the breadth-first algorithm for a single address space. The assignment of BDD sections imposes a total order on the workstations. Each workstation receives a set of REQUEST from all its predecessor workstations before the beginning of the APPLY phase. The APPLY phase is now modified to process only those REQUEST , the variable ids for which belong to the workstation. The set of generated *shadow* requests are sent to appropriate successor workstations for processing. The workstation then waits to receive from the successor workstation, the

```
now_bdd_op(op, $F$, $G$)
    if(NOT a terminal case (op, $F$, $G$))
        if(processor id = 0)
            min_id = minimum variable id of ($F$, $G$)
            create a REQUEST ($F$, $G$) and insert in request_queue[min_id];
        for(proc_id = 0; proc_id < processor id; proc_id++)
            bf_apply_recv(proc_id, set of requests);
        bf_apply(op, first_var_id, last_var_id);
        for(proc_id = processor id + 1; proc_id < num processors; proc_id++)
            bf_apply_send(proc_id);
        for(proc_id = num processors - 1; proc_id > processor id; proc_id--)
            bf_reduce_recv(proc_id);
        result = bf_reduce(first_var_id, last_var_id);
        for(proc_id = processor id - 1; proc_id >= 0 id; proc_id--)
            bf_reduce_send(proc_id);
    return result;
```

Figure 5: BF BDD Algorithm on NOW

generalized address to which each shadow REQUEST gets forwarded to. It then performs modified REQUEST , the variable ids for which belong to the workstation. After the REDUCE phase, the workstation sends a set of generalized addresses to each of its predecessor workstations. The overall procedure can be viewed as top-down APPLY phase followed with bottom-up REDUCE phase for a distributed BDD which is partitioned into set of sections each of which is made up of set of consecutive levels. A graphical representation of this concept which also illustrates the algorithm in Figure 5 has been given in Figure 6. The communication serves as a glue to hold together the computations performed in different memories by using *shadow node forwarding* concept.

5 Implementation

5.1 Data Structures

The BDD generalized address represents a tuple (variable id, memory address).The BDD node represents two generalized addresses, one each for the THEN and the ELSE BDD nodes. The BDD node has a NEXT pointer to link the next BDD node in a hash chain. It is of utmost importance to have a compact representation for the BDD node. The minimum requirement for the size of a BDD node with 32-bit memory pointers is 16 bytes: 4 bytes for the NEXT pointer and 6 bytes x 2 = 12 bytes for representing the THEN and the ELSE generalized addresses. We use a custom memory manager so that each page fits 4096/16 = 256 BDD nodes, all of which belong to the same variable id. The custom memory manager aligns each bdd node on a quad word boundary, which makes it possible to tag last 4 bits each of the THEN, the ELSE, and the next memory pointers (see Figure 7). The THEN and the ELSE memory pointers require one complement bit each, leaving a total of 10 bits out of which 8 are used for the reference count and 2 are used internally to mark the status during the course of a BDD operation.

At the end of the REDUCE phase, a BDD node is obtained from a REQUEST node that is not forwarded. To overload the use of REQUEST data structure with the BDD node data structure, the REQUEST data structure is limited to 16 bytes. Before APPLY phase each REQUEST represents operand BDD nodes, a generalized address for each of which requires 6 bytes. Therefore, we allow only two operand operations. Three operand operations such as

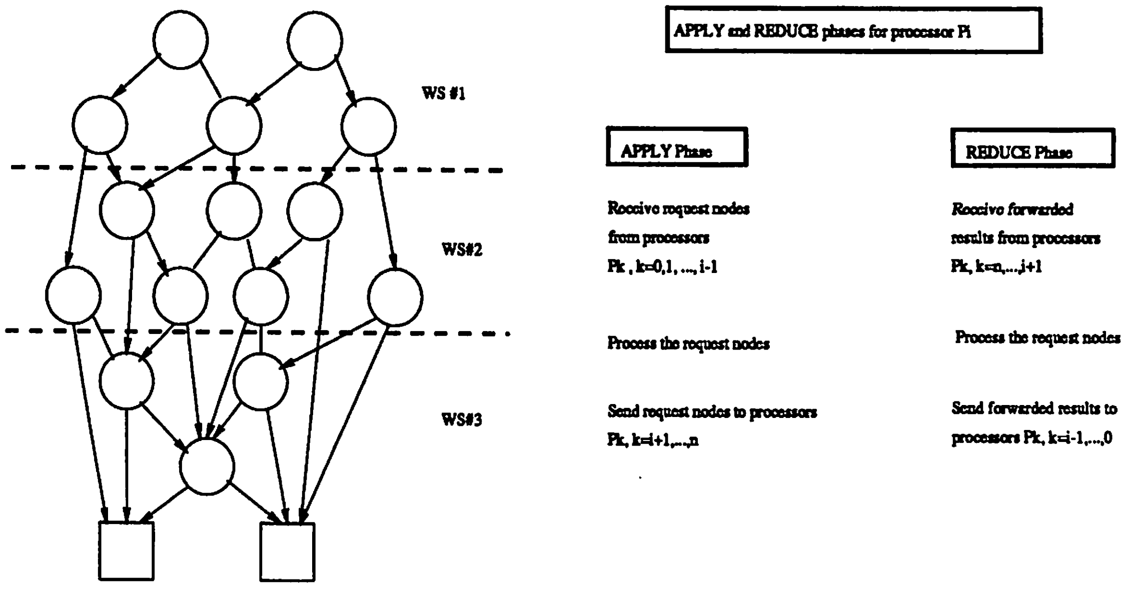
WS #1
WS#2
WS#3
APPLY and REDUCE phases for processor Pi
APPLY Phase
Receive request nodes from processors Pk , k=0,1, ..., i-1
Process the request nodes
Send request nodes to processors Pk, k=i+1,...,n
REDUCE Phase
Receive forwarded results from processors Pk, k=n,...,j+1
Process the request nodes
Send forwarded results to processors Pk, k=i-1,...,0

Figure 6: BDD Manipulation Algorithm on a NOW

$ITE(f, g, h)$ can be simulated by combination of two operand operations if necessary.

5.2 Implementation Issues

The following issues are unique to the breadth-first implementation on NOW.

1. **Shadow REQUEST duplication:** Shadow REQUEST may have multiple shadow REQUEST on different workstations. However, the multiple shadow REQUEST are identified before the REQUEST is processed, hence, only a single REQUEST gets processed and the resulting generalized address is sent to all the workstations with its shadow REQUEST.

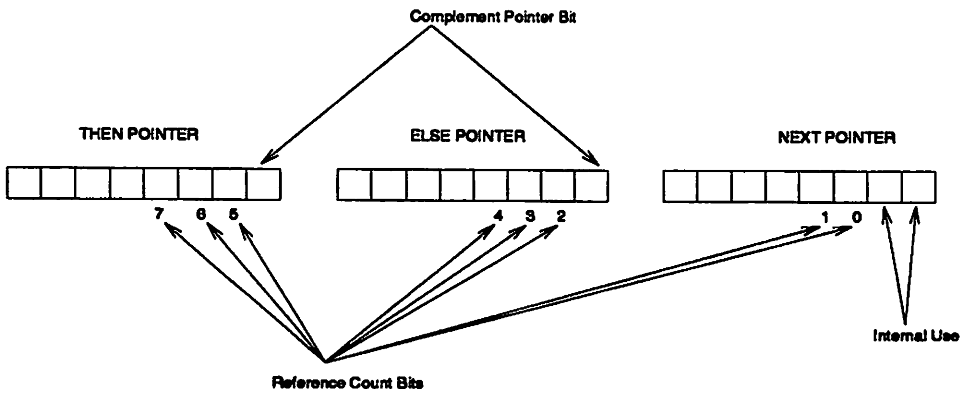

Figure 7: BDD Node Data Structure

2. **Reference count management for nonlocal BDD nodes:**
   (a) Even if a REQUEST can be simplified without accessing the remote memory (e.g. F AND F), it is important to create a new shadow request and process it on appropriate workstation so that the reference count of the node in the unique table is maintained correctly.
   (b) During the REDUCE phase if a redundant node is found for which one of the THEN and or the ELSE generalized addresses point to a node on a successor workstation, we need to adjust reference count of that remote BDD node. This can be achieved by delayed evaluation to avoid communication to all successor workstations after completion of REQUEST phase on a workstation. The delayed evaluation can be performed during the garbage collection step when reference count for the remote nodes can be adjusted appropriately.
3. **Caching shadow** REQUEST **vs. on-line issue of remote requests:** If the shadow request are not cached, we need a network transaction for every shadow REQUEST created during the APPLY phase. Given the high network overhead and latencies this may not be acceptable. However, this may change if communication can be overlapped with computation and low latency, low overhead networks, which can pipeline several small messages, become available.

6 Experimental Results

We have used a heterogenous network of workstations as the computing environment to perform our experiments. This environment contains approximately 60 workstations with 64MB (about 40MB available) main memory and 256MB (about 200MB available) disk space and MIPS-R4000 processor.

We have used PVM [9] (Parallel Virtual Machine) software to provide thc communication between thc workstations in the cluster during a BDD operation. This softwarc permits a network of heterogeneous UNIX computers to be used as a single large parallel computer by providing user level routines to send and receive messages among clusters of workstations.

To evaluate the performance we integrated our BDD package with SIS [14]. In order to systematically analyse the performancc of our algorithms with increase in the BDD size, we have used a series of sub-networks of the ISCAS benchmark C6288. We have suitably taken sub-networks of this benchmark such that the shared BDD sizes of the outputs are roughly multiple of one million. For instance, C6288_3M is one of such examples, for which creating the BDD's of all its outputs will involve creating about three million nodes.

In the following subsections we describe the experiments that highlight the salient features of our approach.

6.1 To Exploit Collective Main Memories

The main emphasis of our approach is to exploit the collective main memory available across all the workstations. This would lead to less page faults and hence reduced wall clock time to complete the computation. To observe this phenomenon, we have used a virtual machine consisting of 4 workstations. The results have been given in Table 1.

From Table 1, we observe that for small BDDs, performance on uniprocessor outperforms that on multiprocessor by about a factor of 2-3. The reason being small examples did not result in significant number of page faults for a single processor and network transaction overhead incurred in the multiprocessor approach resulted in large elapsed time. However, as the number of BDD nodes increase, causing the uniprocessor implementation to page fault enormously, the multiprocessor scheme outperforms uniprocessor scheme.

Examples	# Nodes	Uniprocessor Scheme		NOW Scheme	
		# Page Faults	Elapsed Time (in secs)	# Maximum Page Faults	Elapsed Time (in secs)
C6288_1M	1×10^6	0	877	0	2589
C6288_2M	2×10^6	0	1918	0	3743
C6288_3M	3×10^6	4392	3587	280	4818
C6288_4M	4×10^6	70184	7234	450	5530
C6288_5M	5×10^6	187843	10676	3060	6454
C6288_6M	6×10^6	780361	15844	8090	10397

Table 1: Exploiting Collective Main Memories

6.2 To Exploit Collective Disk Space

Table 6.2 indicates the potential of a NOW in manipulating large BDDs. In this experiment, we increased the number

C6288 subckts	Elapsed Time		
# Nodes	One WS	Two WS	Four WS
9×10^6	26098	24074	*n.p.*
10×10^6	*s.o.*	24853	21617
11×10^6	*s.o.*	36802	*n.p.*
12×10^6	*s.o.*	49801	35652
13×10^6	*s.o.*	47521	*n.p.*
14×10^6	*s.o.*	58383	*n.p.*
15×10^6	*s.o.*	60139	*n.p.*

Table 2: BDDs on Multiple Workstations.

s.o.: could not complete due to disk space limitation
n.p.: data could not be collected due to time constraint

of BDD nodes to be manipulated increased to the extent that it did not fit the disk space of a single workstation. We observe that we are able to manipulate BDDs of much larger size using the collective disks of many workstations.

6.3 Analysis of Experiments

In previous two subsections we have presented results which demonstrate the two key advantages of manipulating BDDs on a NOW, namely, exploiting collective main memory for improved performance and using collective disk space to build large BDDs. However, we note that the time taken to manipulate BDDs on a NOW is large. We monitored the elapsed time in our algorithm and found that a large part of the elapsed time is due to the network transaction. Hence, the performance of our approach is significantly dominated by the penalty incurred during message transfers. The hope is that with the ongoing research in NOW community [2] which includes using asynchronous transfer mode, parallel file server, and active message passing will result in low network latency and overhead. Our approach will take advantage of performance enhancements achieved by NOW research community.

7 Conclusions and Future Work

We have presented algorithm for manipulation of binary decision diagrams (BDD) on a network of workstations (NOW). A NOW provides a collection of main memories and disks which can be used effectively to create and manipulate very large BDDs. We use breadth first manipulation technique to exploit the memory resources of a NOW efficiently. The prototype implementation points to the potential impact this approach can have in manipulating very large BDDs.

The effectiveness of our approach was demonstrated with experiments. This paper serves as a proof of concept for our approach. The work is still in the development stage and we need to add many features to support the assortment of BDD operations. Furthermore, we expect to carry out a multitude of optimizations at various steps to improve the computational efficiency.

Apart from adding necessary features of a BDD package (garbage collection, quantification routines etc), we plan to extend the current package with following features:

1. Utilizing the computation power of NOW: In the current approach, the computations are carried out one processor at a time. Hence, we have only exploited the memory resources of workstations in the network. We plan to extend our approach to utilize the parallel computation power offered by NOW. This will be achieved by *pipelined* processing of REQUEST's during the APPLY and REDUCE phase. In pipelined scheme, REQUEST's are processed on more than one processor concurrently. Hence, a processor need not wait to collect REQUEST's from predecessor processors during APPLY phase and from successor processors during REDUCE phase. This will result in improved computation time of the processing of the REQUESTs. However two drawbacks of this scheme are : i) On-line issue of remote requests, will result in significant increase in the network transaction. We observed in Section 6 that the performance of our approach was signigicantly hamperted by the network latency and overhead. ii) Further, it will result in duplication of effort due to inability to recognize a REQUEST after it is processed in the APPLY phase. Consequently, it will also increase the working memory requirements and amount of work during the REDUCE operation.

   We need to investigate these the benefits of this approach in view of these two drawbacks. A plausible solution could be to adopt a scheme in between two extremes and issue remote requests in a group only.

2. Dynamic load balancing: In the current scheme the variable indices are statically distributed over several processors. This has the disadvantage that if the number of nodes in certain levels grow very large then it leads to uneven distribution of BDD nodes. A better approach would be to dynamically change the distribution of set of variables among the processors to balance the number of nodes on each processor.

References

[1] S. B. Akers. Binary Decision Diagrams. *IEEE Trans. Comput.*, C-37:509–516, June 1978.

[2] T. E. Anderson, D. E. Culler, and D. A. Patterson. A Case for NOW: Network of Workstations. Technical Report UCB/ERL M94/58, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, Nov. 1994.

[3] P. Ashar and M. Cheong. Efficient Breadth-First Manipulation of Binary Decision Diagrams. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 622–627, Nov. 1994.

[4] A. Aziz, F. Balarin, S.-T. Cheng, R. Hojati, T. Kam, S. C. Krishnan, R. K. Ranjan, T. R. Shiple, V. Singhal, S. Tasiran, H.-Y. Wang, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. HSIS: A BDD-Based Environment for Formal Verification. In *Proc. of the Design Automation Conf.*, pages 454–459, June 1994.

[5] R. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.*, C-35:677–691, Aug. 1986.

[6] R. Bryant. A methodology for hardware verification based on logic simulation. *Journal of the Association for Computing Machinery*, 38(2):299–328, Apr. 1991.

[7] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential Circuit Verification Using Symbolic Model Checking. In *Proc. of the Design Automation Conf.*, June 1990.

[8] O. Coudert, C. Berthet, and J. C. Madre. Verification of Sequential Machines Based on Symbolic Execution. In J. Sifakis, editor, *Proc. of the Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 365–373, June 1989.

[9] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM 3 User's Guide and Reference Manual*. Oak Ridge National Laboratory, Sept. 1994.

[10] S. Kimura and E. M. Clarke. A Parallel Algorithm for Constructing Binary Decision Diagrams. In *Proc. Intl. Conf. on Computer Design*, pages 220–223, Nov. 1990.

[11] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[12] H. Ochi, N. Ishiura, and S. Yajima. Breadth-First Manipulation of SBDD of Boolean Functions for Vector Processing. In *Proc. of the Design Automation Conf.*, pages 413–416, June 1991.

[13] H. Ochi, K. Yasuoka, and S. Yajima. Breadth-First Manipulation of Very Large Binary-Decision Diagrams. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 48–55, Nov. 1993.

[14] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. SIS: A System for Sequential Circuit Synthesis. Technical Report UCB/ERL M92/41, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, May 1992.

[15] H. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Implicit State Enumeration of Finite State Machines using BDD's. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 130–133, Nov. 1990.