Multi-threaded Reachability

Debashis Sahoo
Stanford University
sahoo@stanford.edu

Jawahar Jain
Fujitsu Labs of America, Inc.
jawahar@fla.fujitsu.com

Subramanian K. Iyer
University of Texas at Austin
subbuk@cs.utexas.edu

David L. Dill
Stanford University
dill@cs.stanford.edu

E. Allen Emerson
University of Texas at Austin
emerson@cs.utexas.edu

ABSTRACT

Partitioned BDD-based algorithms have been proposed in the literature to solve the memory explosion problem in BDD-based verification. Such algorithms can be at times ineffective as they suffer from the problem of scheduling the relative order in which the partitions are processed. In this paper we present a novel multi-threaded reachability algorithm that avoids this scheduling problem while increasing the latent parallelism in partitioned state space traversal. We show that in most cases our method is significantly faster than both the standard reachability algorithm as well as the existing partitioned approaches. The gains are further magnified when our threaded implementation is evaluated in the context of a parallel framework.

Categories and Subject Descriptors

B.6.3 [**Logic Design**]: Design Aids—*Verification*

General Terms

Algorithms, Measurement, Verification

Keywords

Reachability Analysis, Parallel, Multi-threaded

1. INTRODUCTION

Reachability analysis is typically done using *Reduced Ordered Binary Decision Diagrams* (OBDDs) [1, 3]. A more compact representation of boolean functions, *Partitioned-OBDDs* (POBDDs) [7] leads to further improvement in reachability analysis [9]. However, in standard POBDD-based reachability analysis, the complexity of BDD-based image computation can vary significantly between different partitions. Thus, the relative order in which the partitions are analyzed plays a critical role in the overall performance. Finding an optimal schedule appears to be a hard problem. Therefore, any heuristic to find a good schedule is likely to not perform well in all cases. In a few cases, the approach can get stuck in some difficult partition and, hence, many remaining states which otherwise could have been easily computed are not reached at all. We present a solution to this *scheduling problem* by developing a novel multi-threaded reachability approach. In a multi-threaded environment, using our techniques of *Early Communication* and *Partial Communication*, state space traversal in some partitions can continue even while remaining partitions are proving to be difficult.

We show that our results are better in most cases than OBDDs as well as POBDDs even when the method is running on a single processor. Using our approach, we can locate error states significantly faster than other BDD based methods. We can also show that our results are much better than the standard reachability algorithms in many passing cases as well. Finally, we show that our method is more robust than the standard sequential POBDD-based reachability algorithm as it is able to solve various easy reachability instances which prove to be problematic for current POBDD approaches.

Our approach also improves the parallelism achieved by a naive parallelization of standard POBDD-based algorithm. We estimate the time it takes when all the threads are run in parallel in a multi-processor and shared-memory architecture. Our results show that in a large number of cases we get super linear improvements with respect to the number of processors used, especially for checking failing invariants.

2. PRELIMINARIES

The set of reachable states is obtained by repeatedly performing image computations until a fixed point is reached [3, 8]. This is termed as the *Least Fixed Point* computation. Verification based on reachability can often be improved by the use of POBDDs [6, 9, 11]. Essentially, the POBDD based-reachability algorithm performs as many steps as possible of image computation within each partition i in a step of *least fixed point* within the partition. When no more images can be thus computed, it synchronizes between partitions by considering the transitions that originate in partition i and lead out from there. The term *Communication* refers to these cross-partition image computations that are followed by transferring the computed BDDs to other partitions. Notice that the POBDD-based reachability algorithm performs a BFS which is local to individual partitions, and then synchronizes to add states that result from transitions crossing over from one partition to another. We may characterize this as a region-based BFS, where individual regions

DAC 2005, June 13–17, 2005, Anaheim, California, USA.

of the state space, *i.e,* the partitions, are traversed independently in a breadth first manner. We term the computation within individual partitions as a *local Least Fixed Point* computation or a local LFP computation in short.

Related Work

Several methods have been proposed to do parallel verification. Stern and Dill [13] parallelize an explicit model checker. In [14], parallelized BDDs are used for reachability analysis. Verification using parallel reachability analysis has been studied in [4, 5, 16]. A scalable parallel reachability analysis is presented in [5]. They perform distributed reachability using the classical BFS traversal of the state space in a parallel environment, using distributed memory. A different disjunctive partitioning approach based on iterative squaring is explored in [2]. Thread-based approach has been applied to Constraint-Based Verification in [10].

The focus of this paper is not on designing a good parallel reachability algorithm for a *distributed* computational framework. Rather we assume a *shared* memory model, and the algorithm uses the fast communication between threads, to improve the parallelism. Therefore, we do not compare our method with other parallel reachability algorithms that are designed to run in a *distributed* framework.

3. IMPROVING PARALLELISM IN MULTI-THREADED REACHABILITY

The POBDD-based algorithm given in [9] is naturally parallelizable. The local LFP computation of each partition combined with their *Communication* can be processed in parallel. We have to wait for all the partitions to finish their local LFP computation and the *Communication* to begin transferring the communicated states to the appropriate partition. However, empirically we find that this simple parallelization of the algorithm in [9] doesn't have much parallelism. This may be due to following reasons

High variation of BDD computations

The performance of the image computations inside each partition depend on the BDD variable order. We call a partition an easy partition if the BDDs inside the partition are compact and a hard partition otherwise. For a majority of circuits, the complexity of the BDD computations can have significant variations between different partitions. In such cases, all easy partitions wait for the hard partitions to finish their image computation, which reduces the parallelism significantly.

Depth of the local LFP computation:

Another reason for the reduced parallelism may be due to the fact that the depth of the local LFP computation can vary a lot between partitions. In this case the partition with smaller depth finish faster whereas the partitions with larger depth take longer time. This results in many idle threads which reduces the parallelism.

In practice we find that a large number of partitions wait for a few hard partitions. To address this issue we give following heuristics to improve the parallelism.

Early Communication: Communicate states to other partition after the least fixed point.

Partial Communication: Initiate a partial communication in an idle thread.

3.1 Early Communication

After a partition finish its local LFP computation, we allow the partition to immediately communicate its states to the other partitions. Each partition accepts this communicated states asynchronously during their local LFP computation. This would enable the easy partitions to make progress with their subsequent local LFP computation without waiting for the hard partitions to finish. Therefore, the early communication from easy partitions to other easy partitions enables all such partitions to reach a fixed point. This is very difficult to achieve in sequential partitioned reachability analysis because such scheduling information is difficult to obtain.

If new states are *communicated* during early communication, then we restart the current image computation after adding these states. Such augmentation can make a harder image computation significantly easier in some cases. This may be due to the reason that some of the communicated states correspond to what were hard states to compute in the receiving partition using the local LFP.

3.2 Partial Communication

Even after applying the above technique, we found that many partitions are still waiting for other partitions to communicate some states, so that they can continue their local LFP computation. This case arises when all the easy partition finish their local LFP and need communication from a hard partition to make further progress. To improve parallelism, we initiate a *Communication* in an idle thread using a small subset of the state space of the hard partition. The *Communication* introduces new states in the easy partitions. This enables easy partitions to make progress further with their collective least fixed point from the communicated states. Intuitively this tries to accelerate the activity among easy partitions. We use a small subset of state space instead of the full state space of the hard partition in order to reduce the computational effort in *Communication*. This heuristic tries to keep all the threads busy there by improving the parallelism. Further, this heuristic can increase the number of early communication instances. Thus, the combined effect of the partial communication and early communication improves the parallelism significantly.

```
Parallel-Reachability($n$, $TR$, $InitStates$) {
    Create $n$ partitions for $InitStates$
    Run in parallel for each partition $i${
        After every microsteps runs
                ImproveParallelism($i$) {
            Get all the communicated states
            Calculate LeastFixedPoint($Rch$) in partition $i$
            Compute cross-over states from $i$ to all parts
        }
    } until (No new state is found in any partition);
}
ImproveParallelism($n$: Partition Number) {
    check and add all the communicated states
        if new states are added
            restart current image computation
    request a waiting partition to initiate
        partial communication procedure
}
```

Figure 1: Parallel Reachability Algorithm

3.3 Multi-threaded Reachability Algorithm

We present our complete parallel POBDD-based reachability algorithm as shown in Figure 1 using the techniques discussed in last section.

We run the local LFP computation combined with the *Communication* in parallel. All computation inside a partition is managed by a dedicated thread. Each thread polls for the communicated states from the other threads. After every micro-step of the image computation, each thread calls a function *ImproveParallelism* that implements two heuristics for improving parallelism. The first heuristic is to do early communication. As a part of the first heuristic, the function checks whether other threads have communicated some states to the current thread. If it finds any threads, then it transfer all the communicated states from those threads to the current thread. This simple check and update subroutine performed by each thread implements the early communication heuristic. The second heuristic is to do partial communication. As a part of this heuristic, every active thread checks for an idle thread. If an idle thread is found, then it gives a small subset of the state space from the current partition to the idle thread. The idle thread start a *Communication* from this subset of states to the partition associated with the idle thread.

3.4 Termination Condition

In our multi-threaded approach, each thread manages a partition. The threads goes back to idle state if no new states are communicated to the partition associated with that thread. The thread manager asserts a global termination flag if all the threads are idle.

4. EXPERIMENTAL RESULTS

Our implementation of the POBDD-data structure and algorithms uses VIS-2.0 package. We modified the CUDD [12] package to support multi-threading. Our experiments were run using default cluster size of 5000, lazy sift reordering, MLP image method on a Linux box with 2.20 GHz Intel XEON CPU and 2 GB RAM. We report results only on VIS [15] and industrial circuits. In keeping with the typical timeout limits set in our in-house verification tools, we set a timeout of 5000 seconds on all circuits. For sake of brevity, we present our results only on those circuits where VIS requires more than 100 seconds. Results are omitted for the circuits where all the methods timeout.

4.1 Overview of Table

We compare the total time taken by the following approaches: the standard approach of VIS, the simple partitioning approach and our multi-threading of POBDD-based reachability algorithm. We compare the naive multi-threading with the successive introduction of the two heuristics for communication – early communication and partial communication. The columns in the table are arranged in the same order. The first column is the circuit name, followed by *vis*, *seq*uential POBDDs, *naive* parallelization, the multi-threaded approach with just early communication and finally both techniques. The final column has two parts – *seq* and *est par*, which report, respectively, the total work time for the method under a sequential implementation and the estimated runtime for a parallel shared memory architecture. Note that the *seq* time is only the *work* time (total CPU time), whereas the estimated parallel time also takes into account the idle processor time in an actual parallel implementation. The details of the processor utilization are presented in Section 6.3 using Gantt charts.

In the case of multi-threading, the interleaving of threads is handled by the operating system, and accordingly, there may be a system dependent variation in runtime of the program. In most cases the variation in runtime is quite small. However, for a small number of cases we do observe significant runtime variations. Therefore, the development of a deterministic algorithm is kept as a future work. We report the results in the table by running all the circuits in a single batch.

			our multi-threaded approaches			
					early comm + partial comm	
ckts	vis	seq pobdd	naive	early comm	seq	est par
(a) Industrial Circuits using 16 threads						
c1	371	T/O	T/O	1888	88	14
c2	3346	1728	1738	1417	284	26
c3	2540	T/O	T/O	T/O	1260	93
c4	2236	T/O	T/O	T/O	1955	142
(b) Failing VIS benchmark Circuits using 16 threads						
b12abs2	111	4169	873	311	4333	318
ball7	2076	T/O	T/O	T/O	768	88
blkjack3	T/O	4715	2521	2675	T/O*	3316
palu	273	1098	169	157	44	7
vsa16a6	3897	T/O	T/O	31	41	10
vsa16a7	4212	608	685	31	16	9
vsa16a8	3907	T/O	T/O	31	38	10
vsaR	T/O	75	413	809	866	58
(c) Passing VIS benchmark Circuits using 16 threads						
FIFOs	2259	T/O	T/O	T/O	T/O	T/O
s1269b-1	3635	471	1255	216	184	20
s1269b-5	2287	471	1362	128	133	12
sp_prod	891	143	233	45	144	42
(d) Simple Industrial Circuits using 4 threads						
d1	6.3	T/O	T/O	T/O	8.3	6.6
d2	9.7	T/O	T/O	T/O	6.3	5.8
d3	14.7	T/O	T/O	T/O	14	9.6
d4	10.6	T/O	T/O	T/O	6.4	5.8
d5	11.5	13.1	9.3	9.8	14.6	8.1

(T/O = Timeout of 5000 sec)
(* = Finishes in 5590 seconds)

Table 1: Time (in sec) for Invariant Checking on VIS benchmarks and a few Industrial Circuits

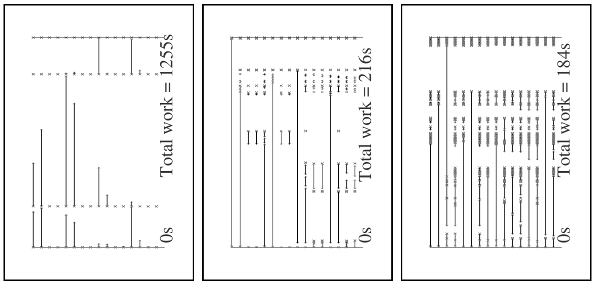
4.2 Efficiency Issues

In the above tables, for almost all entries, the resulting multi-threaded run times (even while using a single CPU) become much faster than standard OBDDs as well. They are also clearly superior to classical Partitioned-reachability. Our proposed multi-threaded approach is also usually superior to the lesser sophisticated multi-threading techniques. If we consider the results in context of a 16-way shared memory architecture, then in an overwhelming majority of difficult cases they offered a super-linear improvement, i.e., greater than 16X, as compared to OBDD based reachability.

Scheduling is a Problem Even on Easy Functions (Table 1(d)) : Consider results of some properties from an industrial design whose OBDDs are fairly small. It is known that creating a large number of partitions on an easy function (BDD) leads to a large amount of overhead [7]. This makes partitioning based verification artificially harder. Thus for properly evaluating the impact of scheduling we make fewer partitions; we choose to make 4 partitions instead of 16. Note that even for such simple cases classical sequential POBDD-based analysis can get trapped in an inefficient schedule. Note, even for an easy function, in POBDD-based reachability a partition can have its local fix-point at a depth larger than the actual fix-point of the function. Good schedule should avoid exploring such partitions early. Due to early communication and round-robin nature of our approach, we avoid unnecessarily deep image calculations in these functions.

4.3 Improving Parallelism

Consider the reachability analysis of *s1269b* from the VIS Verilog benchmark suite. As shown in Table 1(c), we perform reachability analysis using 16 partitions, each of which runs in a separate thread.



(a) (b) (c)

Figure 2: Multi-threaded Reachability with successive addition of each heuristics

Figure 2 shows the Gantt charts of three multi-threaded reachability analysis on *s1269b* circuit. We use the three charts to show the effect of the two heuristics added successively to the reachability algorithm. Figure 2(a) shows Gantt chart of the naive multi-threaded reachability. Figure 2(b) shows the Gantt chart of reachability analysis when early communication is allowed. Figure 2(b) shows the Gantt chart of reachability analysis when both early communication and partial communication are allowed. Each partition is represented by a vertical broken line. The filled segment represents the *work* time for the partition to perform a computation. At the end of each such stage, a small cross indicates the communication of states to other partitions. A break in the line indicates that the corresponding thread is idle. In a multi-processor environment, this corresponds to the idling of the processor. However, in a multi-threaded environment, the processor can immediately schedule another thread for execution. The total time is the *work* time for executing all threads on a single processor. As we can see from the figure, more gaps are being filled with the addition of each heuristic. This shows a clear trend of improved parallelism in each case.

However, in a very deep circuits where there is not much parallelism, our heuristics make unnecessary attempts to improve parallelism, therefore the reachability analysis gets worse as compared to the standard VIS or the standard POBDD-based reachability analysis. A fix of the above limitation is to dynamically recognize deep reachability instances and suitably change the communication strategy in such cases.

5. CONCLUSION

In this paper we show that by avoiding inefficiencies due to the scheduling bottleneck, the use of multi-threading provides significant gains over both POBDD as well as OBDD approaches. Our heuristics is also able to improve the parallelism over a naive multi-threaded approach because it avoids unnecessary waiting for the communication of states. It also uses the communicated states in the active local LFP computation. This enables all the easy partitions to reach a collective local least fixed point among themselves. Therefor, if an error is present in this fixed point, then the algorithm detects it very fast. Further, if the error state is present in a hard partition, then it may detect it fast using the early communication. The greedy nature of multi-threaded reachability allows it to find easy to reach paths to the error states. The above arguments indicate that error detection may happen dramatically faster in such an approach.

For some functions, POBDDs can often be much smaller than OBDDs. Hence, if the instability of scheduling can be ameliorated by a multi-threaded approach, then even though the threaded approach will have an overhead, it may complete the full state space traversal faster than other BDD-based methods for passing cases also. Thus, in most cases (passing or failing) we are significantly faster than both the standard reachability algorithm as well as the partitioned approaches. The gains are dramatically magnified when evaluated in context of a shared-memory parallel architecture.

6. REFERENCES

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

[2] G. Cabodi, P. Camurati, L. Lavagno, and S. Quer. Disjunctive partitioning and partial iterative squaring: An effective approach for symbolic traversal of large circuits. In *DAC*, pages 728–733, 1997.

[3] O. Coudert, C. Berthet, and J. C. Madre. Verification of sequential machines based on symbolic execution. In *Proc. of the Workshop on Automatic Verification Methods for Finite State Systems*, 1989.

[4] H. Garavel, R. Mateescu, and I. Smarandache. Parallel state space construction for model-checking. In *SPIN workshop on Model checking of software*, pages 217–234. Springer-Verlag New York, Inc., 2001.

[5] T. Heyman, D. Geist, O. Grumberg, and A. Schuster. Achieving scalability in parallel reachability analysis of very large circuits. In *CAV*, 2000.

[6] S. Iyer, D. Sahoo, C. Stangier, A. Narayan, and J. Jain. Improved symbolic Verification Using Partitioning Techniques. In *Proc. of CHARME 2003*, volume 2860 of *Lecture Notes in Computer Science*, 2003.

[7] J. Jain. et. al., Functional Partitioning for Verification and Related Problems. *Brown/MIT VLSI Conference*, 1992.

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

[9] A. Narayan. et. al., Reachability Analysis Using Partitioned-ROBDDs. In *ICCAD*, pages 388–393, 1997.

[10] C. Pixley and J. Havlicek. A verification synergy: Constraint-based verification. In *Electronic Design Processes*, 2003.

[11] D. Sahoo and S. Iyer. et. al., A Partitioning Methodology for BDD-based Verification. In *FMCAD*, 2004.

[12] F. Somenzi. CUDD: CU Decision Diagram Package ftp://vlsi.colorado.edu/pub, 2001.

[13] U. Stern and D. L. Dill. Parallelizing the murphy verifier. In *CAV*, 1997.

[14] T. Stornetta and F. Brewer. Implementation of an efficient parallel BDD package. In *DAC*, pages 641–644, 1996.

[15] VIS. Verilog Benchmarks http://vlsi.colorado.edu/˜ vis/.

[16] B. Yang and D. R. O'Hallaron. Parallel breadth-first bdd construction. In *symposium on Principles and practice of parallel programming*, pages 145–156. ACM Press, 1997.