Achieving Scalability in Parallel Reachability Analysis of Very Large Circuits

Tamir Heyman[1], Danny Geist[2], Orna Grumberg[1], and Assaf Schuster[1]

[1] Computer Science Department, Technion, Haifa, Israel
[2] IBM Haifa Research Laboratories, Haifa, Israel

Abstract. This paper presents a scalable method for parallel symbolic reachability analysis on a distributed-memory environment of workstations. Our method makes use of an adaptive partitioning algorithm which achieves high reduction of space requirements. The memory balance is maintained by dynamically repartitioning the state space throughout the computation. A compact BDD representation allows coordination by shipping BDDs from one machine to another, where different variable orders are allowed. The algorithm uses a distributed termination protocol with none of the memory modules preserving a complete image of the set of reachable states. No external storage is used on the disk; rather, we make use of the network which is much faster.
We implemented our method on a standard, loosely-connected environment of workstations, using a high-performance model checker. Our initial performance evaluation using several large circuits shows that our method can handle models that are too large to fit in the memory of a single node. The efficiency of the partitioning algorithm is linear in the number of workstations employed, with a 40-60% efficiency. A corresponding decrease of space requirements is measured throughout the reachability analysis. Our results show that the relatively-slow network does not become a bottleneck, and that computation time is kept reasonably small.

1 Introduction

This paper presents a scalable parallel algorithm for reachability analysis that can handle very large circuits.

Reachability analysis is known to be a key component, and a dominant one, in model checking. In fact, for large classes of properties, model checking is reducible to reachability analysis [3]; most safety properties can be converted into state invariant properties (ones that do not contain any temporal operators) by adding a small state machine (satellite) that keeps track of the temporal changes from the original property. The model checking can be performed "on-the-fly" during reachability analysis. Thus, for safety properties verification is possible if reachability analysis is, and an efficient method for model checking these kind of properties is one where the memory bottleneck is the reachable state space.

There is constant work on finding algorithmic and data structure methods to reduce the memory requirement bottlenecks that arise in reachability analysis.

One of the main approaches to reducing memory requirements is symbolic model checking. This approach uses Binary Decision Diagrams (BDDs) [4] to represent the verified model. However, circuits of a few hundreds of state variables may require many millions of BDD nodes, which often exceeds the computer memory capacity.

There are several proposed solutions to deal with the large memory requirements by using parallel computation. Several papers suggest to replace the BDD with parallelized data structure [11, 1]. Stern and Dill [10] show how to parallelize an explicit model checker that does not use symbolic methods. Other papers suggest to reduce the space requirements by partitioning the work to several tasks [5, 9, 8]. However, these methods do not perform any parallel computation. Rather, they use a single computer to sequentially handle one task at a time, while the other tasks are kept in an external memory. Our work is similar, but we have devised a method to parallelize the computation of the different tasks. Section 7 includes a more detailed comparison with [5] and [8].

Our method parallelizes symbolic reachability analysis on a network of processes with disjoint memory, that communicate via message passing. The state space on which the reachability analysis is performed, is partitioned into *slices*, where each slice is *owned* by one process. The processes perform a standard Breadth First Search (BFS) algorithm on their owned slices. However, the BFS algorithm used by a process can discover states that do not belong to the slice that it owns (called *non-owned* states). When non-owned states are discovered, they are sent to the process that owns them. As a result, a process only requires memory for storing the reachable states it owns, and computing the set of immediate successors for them. As can be seen by the experimental results in Section 6, Communication is not the bottleneck. We can thus conclude that usually, the number of non-owned states found by a process is small.

Computation on a single slice usually requires less memory than computation on the whole set. Thus, this method enables the reachability analysis of bigger models than those possible by regular non-parallel reachability analysis. Furthermore, applying computation in parallel reduces execution time (in practice, doing computation sequentially makes partitioning useless because of the large execution time).

Effective slicing should significantly increase the size of the overall state space that can be handled. This is not trivial since low memory requirements of BDDs are based on *sharing* among their parts. Our slicing procedure is therefore designed to avoid as much redundancy as possible in the partitioned slices. This is achieved by using *adaptive* cost function that for each partitioning chooses slices with small redundancy, while making sure that the partition is not trivial. Experimental results show that our slicing procedure results in significantly better slicing than those obtained by fixed cost functions (e.g. [5, 8]).

Memory balance is another important factor in making parallel computation effective. Balance obtained by the initial slicing may be destroyed as more reachable states are found by each process. Therefore, balancing is dynamically applied during the computation. Our memory balance procedure maintains approximately equal memory requirement between the processes, during the entire computation.

Our method requires passing BDDs between processes, both for sending non-owned states to their owners and for balancing. For that, we developed a compact and efficient BDD representation as a buffer of bytes. This representation enables different variable orders in the sending and receiving processes.

We implemented our technique on a loosely-connected distributed environment of workstations, embedded it in a powerful engine, and tested it by performing reachability analysis on a set of large benchmark circuits. Compared to execution on a single machine with 512MB memory, the parallel execution on 32 machines with 512MB memory uses much less space, and reaches farther when the analysis eventually overflows. Our slicing algorithm achieves linear memory reduction factor with 40-60% efficiency, which is maintained throughout the analysis by the memory balancing protocol. The timing breakdown shows that the communication is not a bottleneck of our approach, even using a relatively slow network.

The rest of the paper is organized as follows: Section 2 describes the main algorithm. Section 3 discusses the slicing procedure and Section 4 suggests several possible optimizations. The way to communicate BDD functions is described in Section 5. Experimental results prove the efficiency of our method in Section 6. Finally, Section 7 concludes with comparison with related works.

2 Parallel reachability analysis

Computing the set of reachable states is usually done by applying a Breadth First Search (BFS) starting from the set of initial states. In general, two sets of nodes have to be maintained during the reachability analysis:

1. The set of nodes already reached, called `reachable`. This is the set of reachable states, discovered so far, which becomes the set of reachable state when the exploration ends.
2. The set of reached but not yet developed nodes, called `new`.

The right-hand-side of Figure 1 gives the pseudo-code of the BFS algorithm.

The parallel algorithm is composed of an initial sequential stage, and a parallel stage. In the sequential stage, the reachable states are computed on a single node as long as memory requirements are below a certain threshold. When the threshold is reached, the algorithm described in Section 3 slices the state space into k slices. Then it initiates k processes. Each process is informed of the slice it owns, and of the slices owned by each of the other processes. The process receives its own slice and proceeds to compute the reachable states for that slice in iterative BFS steps.

During a single step each process computes the set `next` of states that are directly reached from the states in its `new` set. The `next` set contains owned as well as non-owned states. Each process splits its `next` set according to the

```
1 mySlice = receive(fromSingle);
2 reachable = receive(fromSingle);
3 new = receive(fromSingle);
4 while (Termination(new)==0) {
5     next = nextStateImage(new);
6     next = sendRecieveAll(next)
7     next = next $\cap$ mySlice
8     new = next $\setminus$ reachable;
9     reachable = reachable $\cup$ next;
  }
```

(a) BFS by one process

```
reachable = new = initialStates;
while (new $\neq \phi$) {
  next = nextStateImage(new);
  new = next $\setminus$ reachable;
  reachable = reachable $\cup$ next;
}
```

(b) Sequential BFS

Fig. 1. Breadth First Search

k slices and sends the non-owned states to their corresponding owners. At the same time, it receives states it owns from other processes.

The reachability analysis procedure for one process is presented on the left-hand-side of Figure 1. Lines 1-3 describe the setup stage: the process receives the slice it owns, and the initial sets of states it needs to compute from. The rest of the procedure is a repeated iterative computation until distributed termination detection is reached. Notice that, the main difference between the two procedures in Figure 1 is the modification of the set `next` in lines 6-7 as the result of communication with the other processes.

The parallel stage requires an extra process called the *coordinator*. This process coordinates the communication between the processes, including exchange of states, dynamic memory balance, and distributed termination detection. However, the information does not go through the coordinator and is exchanged directly between the processes.

In order to exchange non-owned states, each process sends to the coordinator the list of processes it needs to communicate with. The coordinator matches pairs of processes and instructs them to communicate. The pairs exchange states in parallel and then wait for the coordinator, that may match them with other processes. Matching continues until all communication requests are fulfilled. A process which ends its interaction may continue to the next step without waiting for the rest of the processes to complete their interaction.

2.1 Balancing the memory requirement

One of the objectives of slicing is to distribute an equal memory requirement amongst the nodes. Initial slicing of the state space is based on the known reachable set at the beginning of the parallel stage. This slicing may become inadequate as more states are discovered during reachability analysis. Therefore the memory requirements of the processes are monitored at each step, and whenever they become unbalanced, a balance procedure is executed. The coordinator matches processes that have a large memory requirement with processes that have a small one. Each pair re-slices the union of their two slices, resulting in a better balanced slicing. The pair uses the same procedure that is used to slice the whole state space (described in Section 3) with $k = 2$. After the balance procedure is completed, the pair informs the new slicing to the other processes.

2.2 Termination detection

In the sequential algorithm, termination is detected when there are no more undeveloped states i.e., `new` is empty. In the parallel algorithm, each process can only detect when `new` is empty in its slice. However, a process may eventually receive new states even if at some step its `new` set is temporarily empty.

The parallel termination detection procedure starts after the processes exchange all non-owned states. Each process reports to the coordinator whether its `new` set is non-empty. If all the processes report an empty `new` set, the coordinator concludes that termination has been reached and reports this to all the processes.

3 Boolean function slicing

Symbolic computation represents all the state sets, and the transition relation as Boolean functions. This representation becomes large when the sets are big. To reduce memory requirements we can partition a set into smaller subsets whose union is the whole set. This partition, or slicing should have smaller memory requirements. Furthermore, the subsets should be disjoint in order to avoid duplication of work when doing reachability analysis. Since sets are represented as Boolean functions, slicing is defined for those functions.

Definition 1. *[Boolean function slicing] [9] Given a Boolean function $f : B^n \rightarrow B$, and an integer k, a Boolean function slicing $\chi(f, k)$ of f is a set of k function pairs, $\chi(f, k) = \{(S_1, f_1), \ldots, (S_k, f_k)\}$ that satisfy the following conditions:*

1. *S_i and f_i are Boolean functions, for $1 \leq i \leq k$.*
2. *$S_1 \vee S_2 \vee \ldots \vee S_k \equiv 1$*
3. *$S_i \wedge S_j \equiv 0$, for $i \neq j$*
4. *$f_i \equiv S_i \wedge f$, for $1 \leq i \leq k$.*

The S_i functions define the slices of the state space and we refer to them as *slices*.

Reducing memory requirements depends on the choice of the slices $S_1, \ldots, S_k$. Specifically, when representing functions as BDDs, the memory requirement of a function f, denoted $|f|$, is defined as the number of BDD nodes of f. Thus slicing a function f into two functions f_1 and f_2 may not necessarily reduce the requirements. BDDs are compressed decision trees where pointers are joined if they refer to the same subtree (see Figure 2 for a BDD example). This causes significant sharing of nodes in the Boolean function (for example node 4 in

Figure 2). As a result of this sharing, a poor choice of S_1, S_2 may result in $|f_1| \approx |f|$, and also $|f_2| \approx |f|$.

Finding a good set of slices is a difficult problem. A possible heuristic approach to solving this problem is to find a slicing that minimizes $|f_1|, \ldots, |f_k|$. However as the results of our experiments in Section 6 show, a better approach is to find a slicing that additionally minimizes the sharing of BDD nodes amongst the k functions $f_1, \ldots, f_k$.

3.1 Slicing a function in two: `SelectVar`

Our slicing algorithm, `SelectVar`, slices a Boolean function (a BDD) into two, using assignment of a BDD variable. The algorithm receives a BDD f, and a threshold δ. It selects one of the BDD variables v and slices f into $f_v = f \wedge v$, and $f_{\overline{v}} = f \wedge \overline{v}$. Figure 2 shows an example of such a slicing where the function f is sliced using variable v_1 into f_1 and f_2.

The cost of such a slicing is defined as:

Definition 2. *[Cost(f, v, α):]* $\alpha * \frac{MAX(|f_v|,|f_{\overline{v}}|)}{|f|} + (1-\alpha) * \frac{|f_v|+|f_{\overline{v}}|}{|f|}$

The $\frac{MAX(|f_v|,|f_{\overline{v}}|)}{|f|}$ factor gives an approximate measure to the reduction achieved by the partition. The $\frac{|f_v|+|f_{\overline{v}}|}{|f|}$ factor gives an approximate measure of the amount of sharing of BDD nodes between f_v and $f_{\overline{v}}$ (e.g., node 4 in Figure 2), and therefore reflects the *redundancy* in the partition.

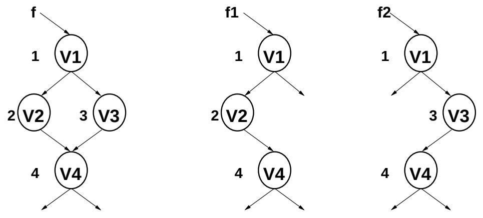


Fig. 2. slicing f into f_1 and f_2

The cost function depends on a choice of $0 \leq \alpha \leq 1$. An $\alpha = 0$ means that the cost function completely ignores the reduction factor, while $\alpha = 1$ means that the cost function completely ignores the redundancy factor. Our algorithm uses a novel approach in which α is adaptive and its value changes in each application of the slicing algorithm, so that the following goals are achieved: **(1)** the size of each slice is below the given threshold δ, and **(2)** redundancy is kept as small as possible.

Initially, the algorithm attempts to find a BDD variable which only minimizes the redundancy factor ($\alpha = 0$), while reducing the memory requirements below the threshold (i.e., $\max(|f_1|, |f_2|) \leq |f| - \delta$). If such a slicing does not exist the algorithm increases α (i.e., allows more redundancy) gradually until $\max(|f_1|, |f_2|) \leq |f| - \delta$ is achieved.

The threshold is used to guarantee that the partition is not trivial, i.e., it is not the case that $|f_1| \approx |f|$ and $|f_2| \approx 0$. If the largest slice is approximately $|f| - \delta$ and the redundancy is small, it is very likely that the other slice is approximately of size δ.

The pseudo code for the algorithm `SelectVar`(f, δ), is given in Figure 3. We set `STEP` $= \min(0.1, \frac{1}{k})$ and $\delta = \frac{|f|}{k}$, where k is the number of overall slices we want to achieve.

```
$\alpha$ = $\Delta\alpha$ = STEP
BestVar = the variable with minimal $cost(f, v, \alpha)$
while $((\max(|f \wedge v|, |f \wedge \overline{v}|) > |f| - \delta) \wedge (\alpha <= 1))$
    $\alpha = \alpha + \Delta\alpha$
    BestVar = the variable with minimal $cost(f, v, \alpha)$
return BestVar
```

Fig. 3. The pseudo code for the algorithm `SelectVar`(f, δ).

Note that, even though our algorithm may compute the cost functions for many different α, $|f \wedge v|$ and $|f \wedge \overline{v}|$ are computed only once for each variable v, therefore, computation time is not increased. Furthermore, the computation of $|f \wedge v|$ and $|f \wedge \overline{v}|$ for different variables, v, is done in parallel. Different computers compute the values for the different variables. The values are then sent to a computer that determines the variable with the minimal cost.

Instead of gradually increasing α, it is possible to find the best α by binary search. For a model with a large number of BDD variables, and large k, this improvement is essential in making our method efficient.

Our slicing procedure is different from those of [9, 5] in that we use adaptive α, and put a lot of emphasis on obtaining small redundancy. Since cost functions are computed in parallel, we can allow computing them more precisely, thus achieving better fine-tuning of our slicing. The comparison to fixed α as suggested in [9, 5] is given in Figure 4.

3.2 Slicing a function into k slices

Recall that `SelectVar` may result in two unbalanced slices that are approximately of sizes $|f| - \delta$ and δ. When we aim at a partition of k slices, we therefore use $\delta = \frac{|f|}{k}$ and repeatedly slice the largest slice, until k slices are obtained. In this way we obtain a balanced partition.

4 Optimizing the `SelectVar` procedure

`SelectVar`(f, δ) selects a state variable v and uses it to split a set f into two sets: $f \wedge v$ and $f \wedge \overline{v}$. Recall that the algorithm attempts to satisfy two conditions; that the size of both resulting sets is at most $|f| - \delta$, and that the redundancy is minimized.

The efficiency of this algorithm and its success in meeting the conditions are crucial factors in the efficiency of the whole scheme, especially when the number of processes increases. In this section we observe that the algorithm can be improved in several ways.

The main observation in improving the slicing procedure is that the split of f which meets best the conditions might not be achieved using a single variable. Indeed, it was previously suggested that the algorithm can achieve better results by the choice of a general function g which determines two sets: $f \wedge g$ and $f \wedge \overline{g}$ [9]. However, since there is an exponential number of candidates for g, trying them all will take too much time. In the rest of this section we develop heuristics which help to choose a "good" g while keeping a reasonable complexity for the choice.

We construct g iteratively as follows. Suppose at a certain step we already have g'. We now choose a state variable v and compute the cost $Cost(f, \alpha, g)$ of all the functions of the form $g = g'$ op v. We use the following options: $g' \wedge v$, $\overline{g'} \wedge v$, $g' \bigotimes v$, $g' \wedge \overline{v}$, $g' \vee v$. Thus, the complexity of computing the cost of all the possibilities at a certain iteration, assuming that we try all state variables, can be as high as five times the number of states variables times the cost of a BDD operation.

In what follows we describe the ways to use the above observation. These are a set of heuristics that we found to be effective (see Table 4). There are several configurable parameters which appear in the description of the heuristics, and which we currently set in our implementation using a trial-and-error methods.

Optimization 1. The general construction of a splitting function g, as described above is called by `SelectVar`(f, δ) when a splitting variable `BestVar` which satisfies the conditions is found, we call `ImprovingSplit` which applies the construction with `BestVar` as a base function, in an attempt to further improve the cost.

Optimization 2. We choose a small number l of the best variables found so far and send them as inputs to the general construction of the splitting function. This time, we use each variable only once: we start with the best one, add the second best, etc. If any of the $l - 1$ resulting functions meet the conditions – we are done. Else, if the functions are different than those found at the end of the previous iteration of `SelectVar`, they are added to the existing list of variables. This increases the input to the next iteration of `SelectVar` by $l - 1$ functions which have high potential of becoming good slicers.

Optimization 3 (Only very small α.) We choose the best splitting variable so far, and iteratively add more variables according to the general construction of the slicing function. However, this time we first select those variables for which the resulting function strictly decreases the size of the slices. Only then, out of those variables selected, we choose the one for which the slicing function achieves a minimal cost.

5 Efficient transfer of BDDs

As described in Section 2, processes periodically exchange BDDs during reachability analysis. Two utility functions are used. `bdd2msg` translates a BDD into a more compact `msg` data and `msg2bdd` translates the `msg` data back to a BDD after it has been transferred. The purpose of `bdd2msg` is to serialize the BDD structure in order for it to be suitable for raw buffer transfer.

BDD nodes represent a boolean function f recursively. The functions 0 and 1 are represented by special BDDs called ZERO, and ONE respectively. Other functions are represented by a node that contains variable identification x, and two pointers, `leftPtr` and `rightPtr`, that point to two other BDD nodes that representing $f_{\overline{x}}$ and f_x, respectively. The function f is expressed based on the Shannon expansion: $\overline{x}f_{\overline{x}} + xf_x$.

The `msg` data is a sequence of records. Each `msg` record has four fields: An index for that record (symbolic pointer), denoted as `Sid`. The variable id of the record denoted as `Xid`. An `Sid` for the record left son, and an `Sid` for its right son. The index field indicated the record location in the `msg` data. The records ZERO, and ONE have special index.

`bdd2msg` traverses the nodes of BDD f in Depth First Search (DFS) order. It creates the corresponding `msg` records from the leaves upwards. Every time it creates a new `msg` record, it increments an index, which serves as the `Sid` for that record. `msg2bdd` traverses the `msg` records sequentially from start to end. It creates the corresponding BDD nodes one by one as it traverses the data. Shannon expansion is used to create the BDD node from the record. Such transformation is possible due to the fact that the `Xids` remain constant throughout the computation.

Remark: The transferred BDD f is compressed by the restrict operator described in [6], using the slice of the receiving process as the restricting domain.

6 Experimental results

In this section we report initial performance results of using our approach. We implemented our partitioned BDD and embedded it in an enhanced version of McMillan's SMV [7], due to IBM Haifa Research Laboratory [2].

Our parallel testbed includes 32 RS6000 machines, each consisting of a 67MHz PowerPC processor and 512MB memory. The communication between the nodes consists of a 16Mbit/second token ring. The nodes are non-dedicated; i.e., they are mostly workstations of employees who would often use them (and the network) at the same time that we ran our experiments.

We experimented using five of the largest circuits we found in the benchmarks ISCAS89 +addendum'93. We also used two large examples (BIQ and ARB) which are components in the IBM's Gigahertz processor. Characteristics of the seven circuits are given in Figure 5.

6.1 Slicing Results

The success of our slicing algorithm is a crucial factor in the efficiency of the parallel execution. This success is indicated by two parameters of the obtained partition: the *redundancy*, which is the ratio between the overall size of the slices and the original reachable size, and the *memory reduction*, which is the ratio between the original reachable size and the largest slice in the partition.

Figure 4 presents the slicing results of reachable sets for four slicing methods. In order to show phenomena which appear only towards large number of slices, results in Figure 4 are given for 16, 32, 64, and 130 slices. The slicing algorithms are invoked when the size of the reachable set exceeds the threshold 100,000 BDD nodes.

The first method selects as a slicing function the variable which achieves the biggest memory reduction. In algorithm SelectVar this corresponds to choosing $\alpha = 1$. The second method is the same as that used in Cabodi et. al. [5]. This corresponds to choosing the splitting variable with the best fixed α. The third method is the one presented in Section 3, adapting α to select the partition with minimal redundancy. The fourth method includes the optimizations described in Section 4, so that splitting is carried using a general function.

The table shows that the average increase in the memory reduction which can be attributed to our optimizations (adapting α and choosing a general splitting function), is 25%, 22%, 18%, and 10% for slicing into 130, 65, 32, and 16 parts, respectively. We conclude that these optimizations become more important as the level of slicing increases. This proves that the key to better slicing when the number of slices increases is to opt for lower duplication, which is the base orientation for our optimizations.

The average memory reduction factor achieved over our benchmark suit for slicing into 130 slices, is more than 55. We expect the results to improve for high slicing levels when a larger threshold is chosen. The reason for that is the threshold per slice, in our experiments $100,000/130 = 750$ which may be too small. On the other hand, efficiency dictates earlier split when the bottleneck is the complexity of the slicing algorithm, or the resources required by the initial sequential stage.

6.2 Parallel Reachability – Space Reduction

We now present the results for reachability analysis of the benchmark suit using our 32 machine testbed. Figures 6 to 12 summarize the memory usage, giving the reachable size and peak usage for every step. Each of the graphs compares the memory usage in the single-machine execution to that of the parallel system. For the parallel system we give both average and highest memory utilization in any of the machines.

The graphs show that scalability is obtained due to the performance of the slicing algorithm, which achieves a good memory reduction. The circuits which overflow always reach with the parallel execution to a farther step than when using the single-machine. Figure 11 shows the analysis process for circuit BIQ which safely reaches step 32 with the parallel execution. BIQ reaches only step 22 with the single machine execution.

One of the crucial factors in the success of the parallel reachability scheme is the dynamic memory balancing, which is in charge of maintaining the "accomplishments" of the slicing algorithm. The ratio of worst to average space usage in the graphs indicates that our dynamic load balancing algorithm succeeds to avoid extreme imbalance.

Note that the measures on peak size are subject to the gc scheduling policy, thus the phenomena appearing e.g., in Figure 7, where the reachable set shrinks while the peak remains very high.

6.3 Parallel Reachability – Timing and Communication

Figure 13 gives the timing breakdown for reachability analysis on the benchmark suit. This table provides information regarding the ratio of computation (compute) to communication (exchange) and memory balancing (balance) in our scheme. The table shows that the overall picture is fairly balanced. In other words, the table shows that communication is not a bottleneck in our algorithm, despite the fact that we use a relatively slow network.

7 Comments on related work

In this section we discuss the improvements of our algorithm over the reachability analysis algorithms presented in [5, 8]. Their algorithms slice the computation, but do not parallelize it. We also summarize the special consideration needed by parallel implementation.

The slicing suggested in [5] is more general than that in [8]. However, as shown in Section 6 above, its effectiveness is limited to a small number of slices. The adaptive α and the general slicing functions used by our algorithm proved scalability and worked well even with 130 slices. Experimental results show that the the impact of our optimizations increases with the level of slicing.

Balancing the memory requirements among slices during computation increases the overall reduction. The algorithm in [8] does not include any balancing. The balancing suggested by [5] constantly increases the number of slices. Our balancing method keeps the number of slices fixed, while successfully maintaining the work balanced. This is important when the network size is fixed.

At the end of each step, [5] and [8] write to the disk the sets `reachable` and `new`, obtained for the slice under consideration. Rather, we send on the network (which is much faster) only the non-owned part of `new`, which is relatively small.

In [8] the authors comment that they believe their algorithm can be parallelized. This, however, is not immediate. In order to exploit the full power of the parallel machinery, we had to adapt the BFS for asynchronous computation, we coordinated and minimized communications, avoided unnecessary blocking, and employed a distributed termination detection.

Our system uses a powerful model checker, which shows that our scheme integrates nicely with state of the art tools. As a result, we were able to experiment with very large circuits, reaching farther steps. For instance, [5] report overflow in step 4 of s5378, while our system reached fixed point at step 44.

8 Acknowledgement

We would like to thanks the SP Hardware Verification group in IBM Poughkeepsie and specifically John Aylward for enabling us to use their 60 nodes IBM SP system in order to perform out experiments.

References

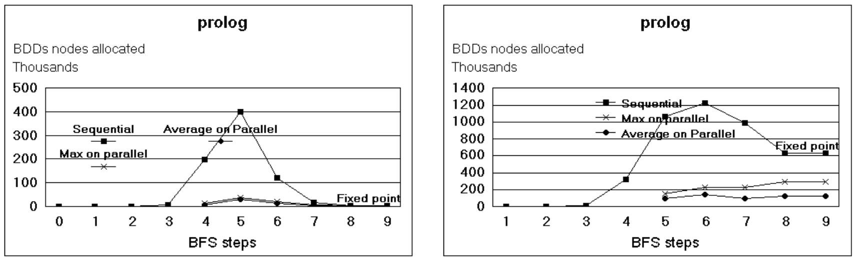
1. S. Basonov. Parallel Implementation of BDD on DSM Systems, 1998. M.Sc. thesis, Computer Science Department, Technion.
2. I. Beer, S. Ben-David, C. Eisner, and A. Landver. Rulebase: an industry-oriented formal verification tool. In *33rd Design Automation Conference*, pages 655–660, 1996.
3. I. Beer, S. Ben-David, and A. Landver. On-the-fly model checking of rctl formulas. In *10th Computer Aided Verification*, pages 184–194, 1998.
4. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.
5. Gianpiero Cabodi, Paolo Camurati, and Stefano Que. Improved reachability analysis of large fsm. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 354–360. IEEE Computer Society Press, June 1996.
6. Olivier Coudert, Jean C. Madre, and Christian Berthet. Verifying Temporal Properties of Sequential Machines Without Building their State Diagrams. In R. Kurshan and E. M. Clarke, editors, *Workshop on Computer Aided Verification, DIMACS*, pages 75–84. American Mathematical Society, Providence, RI, 1990.
7. K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publishers, 1993.
8. Adrian A. Narayan, Jain J. Jawahar Isles, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Reachability analysis using partitioned-robdds. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 388–393. IEEE Computer Society Press, June 1997.
9. Adrian A. Narayan, Jain Jawahar, M. Fujita, and A. Sangiovanni-Vincentelli. Partitioned-robdds. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 547–554. IEEE Computer Society Press, June 1996.
10. Ulrich Stern and David L. Dill. Parallelizing the murphy verifier. In *Proc. of the 9th International Conference on Computer Aided Verification, LNCS 1254*, pages 256–267. Springer, June 1997.
11. T. Stornetta and F. Brewer. Implementation of an efficient parallel bdd package. In *Design Automation Conference*. IEEE Computer Society Press, 1996.

circuit/method	16 slices		32 slices		65 slices		130 slices	
	mem	red	mem	red	mem	red	mem	red
prolog, 4th step, reachable size is 199,961 BDD nodes								
$\alpha = 1$	8.33	1.38	14.75	1.65	23.26	2.09	37.79	2.69
fixed α	8.33	1.38	14.98	1.65	23.26	2.09	37.76	2.60
adaptive α	10.23	1.34	16.82	1.44	28.54	1.84	43.23	2.34
adaptive α + optimizations	9.89	1.22	16.66	1.44	30.01	1.66	48.87	2.04
s1269, 3rd step, reachable size is 100,170 BDD nodes								
$\alpha = 1$	12.75	1.05	22.45	1.21	36.84	1.42	60.75	1.74
fixed α	12.75	1.04	22.33	1.19	36.16	1.38	61.47	1.66
adaptive α	12.75	1.04	21.40	1.20	36.04	1.38	62.53	1.62
adaptive α + optimizations	12.75	1.02	22.33	1.18	38.53	1.27	70.42	1.41
s3330, 4th step, reachable size is 233,952 BDD nodes								
$\alpha = 1$	6.18	2.17	9.23	2.86	13.46	3.93	19.83	5.35
fixed α	6.18	2.15	9.11	2.82	13.46	3.83	19.83	5.20
adaptive α	6.02	2.16	8.92	2.82	14.00	3.68	19.94	4.72
adaptive α + optimizations	6.17	1.92	9.81	2.40	19.81	3.16	24.16	4.16
s1423, 10th step, reachable size is 175,631 BDD nodes								
$\alpha = 1$	9.87	1.24	16.65	1.37	30.51	1.51	54.59	1.74
fixed α	9.87	1.24	16.65	1.34	31.97	1.47	55.60	1.69
adaptive α	10.64	1.13	19.21	1.21	35.88	1.36	64.76	1.50
adaptive α + optimizations	11.60	1.10	18.24	1.18	38.36	1.23	70.68	1.35
s5378, 7th step, reachable size is 177,105 BDD nodes								
$\alpha = 1$	5.37	2.13	8.96	2.72	13.83	3.43	22.84	4.39
fixed α	5.88	1.95	8.96	2.72	13.83	3.43	25.29	3.84
adaptive α	7.71	1.74	11.81	2.04	19.43	2.63	28.46	3.58
adaptive α + optimizations	8.63	1.56	12.14	2.03	19.94	2.51	30.70	3.26
BIQ, 7th step, reachable size is 203,019 BDD nodes								
$\alpha = 1$	8.06	1.48	13.33	1.67	24.43	1.93	41.12	2.26
fixed α	8.54	1.44	13.60	1.60	26.87	1.86	43.35	2.15
adaptive α	8.54	1.37	16.33	1.50	29.36	1.69	50.00	1.97
adaptive α + optimizations	8.79	1.27	20.11	1.21	37.65	1.24	65.50	1.38
ARB, 7th step, reachable size is 177,105 BDD nodes								
$\alpha = 1$	10.43	1.35	17.15	1.50	28.78	1.72	48.46	1.97
fixed α	9.99	1.31	16.15	1.45	28.77	1.63	51.76	1.83
adaptive α	10.03	1.28	17.46	1.34	32.61	1.46	58.31	1.66
adaptive α + optimizations	10.72	1.08	19.64	1.17	37.03	1.26	71.21	1.27

Fig. 4. Partitioning results measured by two parameters: the redundancy (red) which is the ratio between the overall size of the slices and the original reachable size, and the memory reduction (mem), which is the ratio between the original reachable size and the largest slice in the partition.

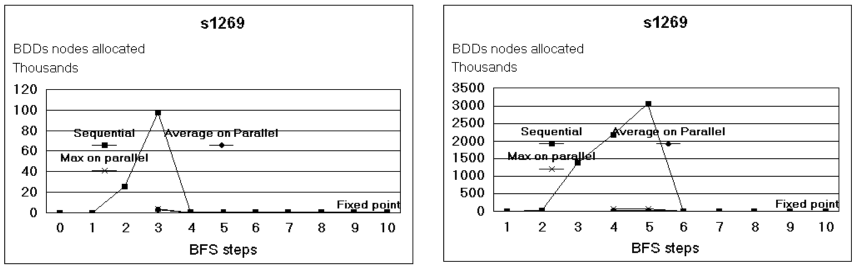
Circuit	#	max reachable		max new		peak		fixed point		gc
	vars	size	step	size	step	size	step	time	steps	time
prolog	117	402k	5	578k	5	1,283k	6	1,288	9	113
s1269	55	98k	3	128k	3	3,055k	5	1,371	10	175
s3330	172	839k	6	2,107k	6	4,250k	7	17,777	9	606
s5378	198	524k	25	157k	35	4,058k	19	98,209	44	6,206
s1423	91	3,831k	13	3,691k	13	11,413k	14	4,079	ov(14)	420
BIQ	187	1,744k	22	1,589k	22	8,954k	23	35,495	ov(23)	5,512
ARB	116	1,193k	6	1,183k	6	8,893k	7	3,112	ov(7)	473

Fig. 5. Characteristics of our benchmark suit, taken from ISCAS89+addendum'93, and the IBM's Gigahertz processor. All sizes are given in BDD nodes, and all times in seconds. Max reachable is the maximal (over the steps) set of nodes already reached. Max new is the maximal (over the steps) set of nodes reached but not yet developed. Note that new may be larger than reachable (at any step), since the joint BDD representation of the current-step' new and the previous-step' reachable may reduce in size. The peak is the maximal size at any point during a step. In order to mask the effect of garbage collection (gc) scheduling decisions, the peak is measured after every gc invocation. Fixed point is the number of steps/time it takes to get to fixed point. Ov(x) means memory overflow at step x. The time was measured using an RS6000 machine, consisting of a 225MHz PowerPC processor with 512MB memory.



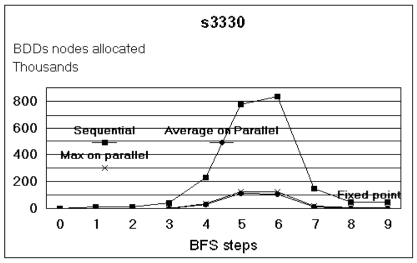
(a) **Size of reachable states set** (b) **Nodes allocated (peak)**

Fig. 6. Memory utilizations during reachability analysis of prolog.

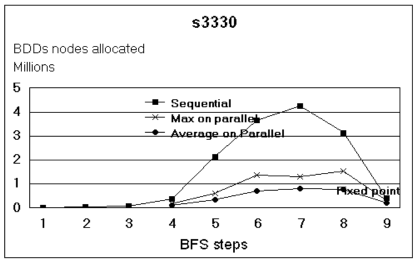


(a) **Size of reachable states set** (b) **Nodes allocated (peak)**

Fig. 7. Memory utilizations during reachability analysis of s1269.

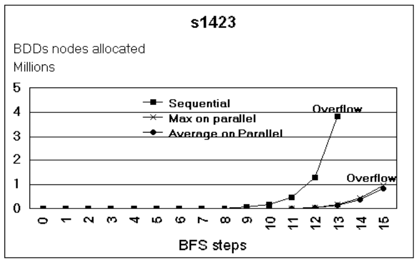


(a) **Size of reachable states set**

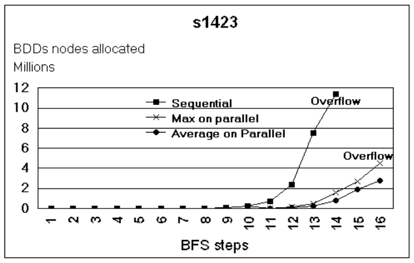


(b) **Nodes allocated (peak)**

Fig. 8. Memory utilizations during reachability analysis of s3330.

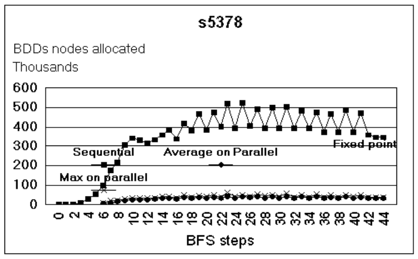


(a) **Size of reachable states set**

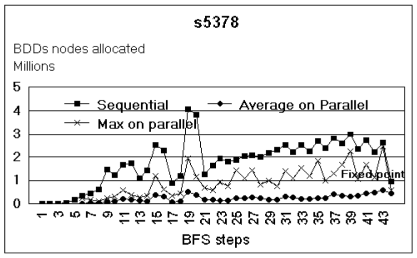


(b) **Nodes allocated (peak)**

Fig. 9. Memory utilizations during reachability analysis of s1423.

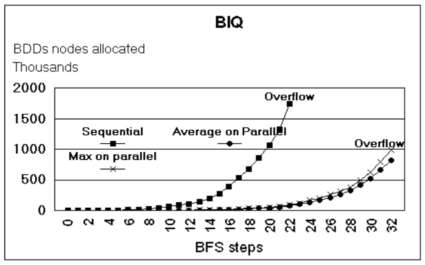


(a) **Size of reachable states set**

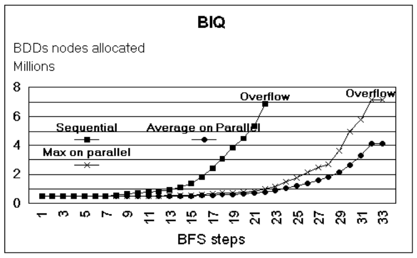


(b) **Nodes allocated (peak)**

Fig. 10. Memory utilizations during reachability analysis of s5378.

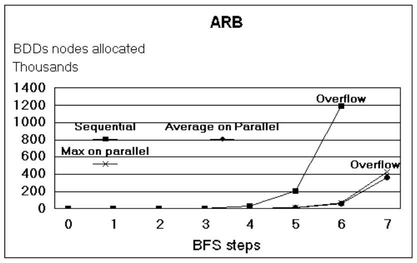


(a) **Size of reachable states set**

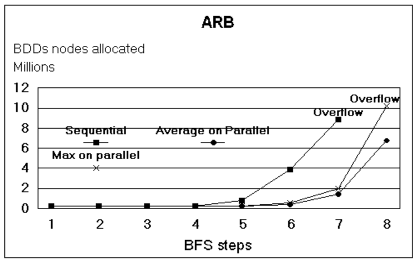


(b) **Nodes allocated (peak)**

Fig. 11. Memory utilizations during reachability analysis of BIQ.



(a) **Size of reachable states set**



(b) **Nodes allocated (peak)**

Fig. 12. Memory utilizations during reachability analysis of ARB.

Circuit	steps	seq. stage	slicing	total parallel	compute	exchange non-owned	balance	gc
prolog	9	47	3,289	1,282	478	544	421	371
s1269	10	312	1,091	167	101	46	0	119
s3330	9	28	1,218	9,765	8,113	1,907	646	619
s5378	44	345	1,203	17,224	9,247	2034	5,489	1,698
s1423	(12)14	155	820	5,541	678	186	4,192	681
BIQ	(22)32	50	389	78,515	35,717	4,458	30,881	9,262
ARB	(6)7	26	361	5,137	1,644	231	2,746	515

Fig. 13. Timing data (seconds) for parallel execution on 32×512MB machines. Each of the measures is the worst sample over all the machines. The steps count shows that in the case of overflow we got farther from where the 512MB single-machine experiment gave up (given in brackets). The sequential stage shows the time it took to get to the threshold where slicing is invoked. The total parallel is the total time over all steps, including computing, exchanging non-owned states, memory balancing, and garbage collection time. Note that the total time is the maxima over sums and not the sum over maxima. Note that communication time is counted only in the exchanging non-owned and balancing columns.