Verifying Very Large Industrial Circuits Using 100 Processes and Beyond

Limor Fix[2], Orna Grumberg[1],Amnon Heyman[3], Tamir Heyman[2], Assaf Schuster[1]

[1]Computer Science Department, Technion, Haifa, Israel
[2]Logic and Validation Technology, Intel Corporation, Haifa, Israel
[3] Phonedo, Herzliya, Israel

Abstract. Recent advances in scheduling and networking have cleared the way for efficient exploitation of large-scale distributed computing platforms, such as computational grids and huge clusters. Such infrastructures hold great promise for the highly resource-demanding task of verifying and checking large models, given that model checkers would be designed with a high degree of scalability and flexibility in mind.
In this paper we focus on the mechanisms required to execute a high-performance, distributed, symbolic model checker on top of a large-scale distributed environment. We develop a hybrid algorithm for slicing the state space and dynamically distribute the work among the worker processes. We show that the new approach is faster, more effective, and thus much more scalable than previous slicing algorithms. We then present a checkpoint-restart module that has very low overhead. This module can be used to combat failures which become probable with the size of the computing platform. However, checkpoint-restart is even more handy for the scheduling system: it can be used to avoid reserving large numbers of workers, thus making the distributed computation work-efficient. Finally, we discuss for the first time the effect of reorder on the distributed model checker and show how the distributed system performs more efficient reordering than the sequential one.
We implemented our contributions on a network of 200 processors, using a distributed scalable scheme that employs a high-performance industrial model checker from Intel. Our results show that the system was able to verify real-life models much larger than was previously possible.

1 Introduction

This paper presents several novel techniques to enhance distributed reachability computation. The techniques enable effective use of a network of 100 computers for the verification of large industrial hardware designs that could not be verified by previously available tools.

For a long time the state explosion problem has been the showstopper of BDD-based (symbolic) model checking [3]: The BDD structures simply cannot squeeze into the RAM available to a single computer. SAT-based model checking [2] can find errors in very large systems, but is limited when used for

verification [9]. In fact, BDD-based model checking is usually superior, when verification is required. Larger systems usually have longer diameters and therefore SAT-based bounded model checking can cover smaller parts of their state space.

In recent years, several distributed BDD-based reachability algorithms have been introduced [8, 7, 6] for networks of communicating computers with distributed memory. Reachability is an important problem because model checking of all temporal safety properties can be reduced to it [1]. Distributed reachability exploits the memory modules and the computation power of a heterogeneous cluster of computers, where more and more machines can be employed on demand. The collective storage offered by the cluster RAM is utilized in a memory- and work-efficient manner, essentially operating as a yet another layer in the memory hierarchy.

However, if these algorithms are to be scaled for very large models that require hundreds of computers, then several enhancements are required. First, fast and effective slicing is needed, in order to accommodate frequent splits in the memory content of overflowed computers. Second, a checkpoint/restart mechanism is needed to recover from a single computer failure and in order to better utilize clusters of computers when memory requirements vary significantly during computation. Finally, dynamic BDD variable reordering should be adapted to work well with the distributed algorithm.

Our work provides solutions for all of these requirements. We developed a *hybrid* algorithm for slicing very large sets quickly and effectively. The user provides the algorithm with measures for an effective slicer, and the algorithm searches for an adequate one. The algorithm is designed to spend as little time as possible in finding an adequate slicer, not necessarily the best one. It starts with a fast estimated computation. If no adequate slicer is found, it gradually applies more precise computations. We compare our hybrid algorithm with the fast estimating algorithm `Est` [5]. We show that our algorithm produces far fewer duplications. We also compare our algorithm to the exhaustive algorithm `Exh` [8], which is better than or equal to other exhaustive algorithms [4, 11, 10]. We show that it is faster than `Exh`, and, in fact, the difference in run time increases when the size of the BDD or the size of its support increase.

We also propose a non-coordinated checkpoint/restart mechanism as part of the distributed reachability computation. In the distributed reachability analysis [8], each worker *owns* a subset of the state space and iteratively computes the set of reachable states within its ownership. It may also find states owned by others workers, which it sends to them. Likewise, it receives owned states, found by others. The checkpoint mechanism consists of occasionally freezes by each worker. The worker stores its configuration, including the set of states it owns, the set of states computed so far, the iteration number, and the BDD variable ordering. Restart is performed by finding a set of configurations, all taken from the same iteration, whose ownership covers the whole state space. A set of new free workers is then initialized with these configurations and resumes the computation.

The checkpoint/restart mechanism is particularly useful when running on a non-dedicated network. Two tasks running on such a network may reach their memory peak at the same time, thus blocking each other. It then might be necessary to freeze one of them and enable the other to continue. When the memory requirement of the active task decreases, the frozen one can be resumed. In addition, when memory requirements vary significantly during computation, an effective utilization will require clusters of varying sizes. Changing the cluster size is done by freezing the active workers and restarting them on a different cluster with an appropriate size.

In order to maintain effective dynamic reordering, we propose a distributed paradigm to control the points at which dynamic variable reordering is performed. In sequential computation, reorder is invoked after garbage collection, if the BDD size exceeds a certain threshold. The distributed computation applies the same policy. In addition, for each worker, it uses two new controlling operations: enforcing reorder when an overflow occurs; and updating the threshold following an action that reduces the BDD size. Reorder when overflow occurs may save unnecessary splits.

Another improvement to the BDD package enforces timeout on BDD operations that do not terminate within a reasonable time. Usually this is due to the size of their operands. We then split the BDDs and resume the operations on two smaller BDDs.

We demonstrated the utility of our scheme by implementing it as a large-scale distributed engine that consists of more than 100 computers and uses a high-performance model checker. We ran our experiments on clusters composed of ordinary PCs. Our results show that the system can *verify* (apply full reachability to) much larger models than could previously be verified. In addition, our results show that in some cases, when the distributed algorithm needs more processes than available, it still reaches a further step than SAT-base bounded model checking does.

In summary, the contributions of the paper are:

- Fast and effective slicing with small memory overhead.
- A checkpoint/restart mechanism.
- An enhanced BDD package: adaptive dynamic variable reorder and timeout on BDD operations.
- Orthogonality to high-performance model checking: all features of sequential model checking remain effective in the distributed framework.

All of the above allows the *verification* of large industrial components.

The rest of the paper is organized as follows. Section 2 presents a new algorithm for fast and effective slicing of very large sets. Sections 3 describes the checkpoint/restart mechanism. Finally, Section 4 presents our distributed reachability analysis, including a paradigm for dynamic variable reordering, and presents our experimental results on verification of large industrial designs.

2 Hybrid Algorithm for Slicing Very Large Sets

In this section we present a new algorithm for slicing very large sets quickly and effectively. The approach makes use of user-supplied measures of effectiveness: the algorithm simply searches for a slicer that meets the measures. The algorithm attempts to reduce the time spent finding a sufficiently effective slicer; it does not necessarily search for the best one. Rather than checking all variables in the support of the set to be sliced, as was done previously, the proposed algorithm makes use of the abundance of good slicers in the support to pick one from a randomly selected sample.

The algorithm gets as its input a set of states as a characteristic function f and returns a variable v called slicer, which slices f into two subsets: $f \wedge v$ and $f \wedge \overline{v}$. Such slicing is *effective* if two requirements are fulfilled. First, the size of each of the subsets is smaller than the size of f itself: $\frac{\max|f\wedge v|,|f\wedge\overline{v}|}{|f|} < \delta_1$. Second, the *duplication* is not too big: $\frac{|f\wedge v|+|f\wedge\overline{v}|}{|f|} < \delta_2$. The minimum reduction factor and the maximum duplication factor δ_1, δ_2 are provided by the user, or by the higher-level procedure calling the algorithm.

The algorithm proceeds through a sequence of three phases. In each consecutive phase the algorithm spends more time trying to find an effective slicer. Once an effective slicer is found the algorithm declares success and terminates. After three unsuccessful phases the algorithm returns the best slicer it has found so far.

In order to test the effectiveness of a candidate slicer, the BDDs of $f \wedge v$ and $f \wedge \overline{v}$ must be built and their relative sizes measured. This consumes time and memory. In contrast, one can estimate the sizes of the slices in a single scan of the BDD of f without creating a new BDD [12]. Estimation is a lot faster than precise calculation and requires far fewer resources.

In the first phase the algorithm employs the method `Est` [5] to search for an effective slicer. This method initially computes an estimate of the size of $f \wedge v$ and $f \wedge \overline{v}$, for each variable v in the support of f. Then it selects as a slicer, among all other variables, the variable v for which the maximum of the estimates for $f \wedge v$ and $f \wedge \overline{v}$ is minimal. Next, a precise calculation is used to determine whether v is an effective slicer. If v is found to be effective, the algorithm terminates; otherwise it proceeds to the next phase.

In the second phase, the algorithm randomly selects a subset $varSet$ of variables out of the support of f. The $varSet$'s size depends on the required confidence degree in finding at least one effective variable (See Subsection 2.1). $effectiveSet$ holds all the variables in $varSet$ that were first estimated as effective, and only those that seem to be effective are checked precisely. If $effectiveSet$ is empty, the second phase ends unsuccessfully. Otherwise, the best slicer from $effectiveSet$ is selected by `Exh`. We remark that the `Exh` procedure itself is no different than the slicing mechanisms described in [8]. Thus, in this paper, we use it as a black box.

The third phase is similar to the second. The difference is that $effectiveSet$ now holds all the variables from $varSet$ that slice effectively using a precise cal-

culation. Finally, if the third phase fails and none of the variables is effective, the most effective variable, `MEff`, is selected among the variables that were computed in the third phase and this variable is returned.

Figure 1 describes the algorithm `Hybrid` for finding a slicer. Lines 1-2 describe the first phase, which uses the `Est` method to select a slicer v. If v is found to be effective, the algorithm terminates. Lines 3-6 describe the second phase where $varSet$ is randomly selected from the variables in the support of f. Then $effectiveSet$ gets only the variables that are effective slicers. This computation is done by first applying a fast estimated check and only then a precise check. The precise check is applied only on slicers that are estimated to be effective. Finally, the algorithm `Exh` is used to find the best slicer out of $effectiveSet$. Lines 7-9 describe the third phase where a precise check is applied to all variables in $varSet$. If the third phase failed, in line 10 is the most effective slicer found so far is returned.

```
function Hybrid(f)
 1  v=Est(f)
 2  if effective(v, precise) return v
 3  varSet=randomselect(support(f))
 4  effectiveSet={v | v ∈ varSet ∧ effective(v, fast) ∧ effective(v, precise)}
 5  if effectiveSet ≠ ∅
 6    return Exh(f, efectiveSet)
 7  effectiveSet={v | v ∈ varSet ∧ effective(v, precise)}
 8  if effectiveSet ≠ ∅
 9    return Exh(f, efectiveSet)
10  return MEff(f, efectiveSet)
```

Fig. 1. Pseudo–code for the slicing algorithm `Hybrid`

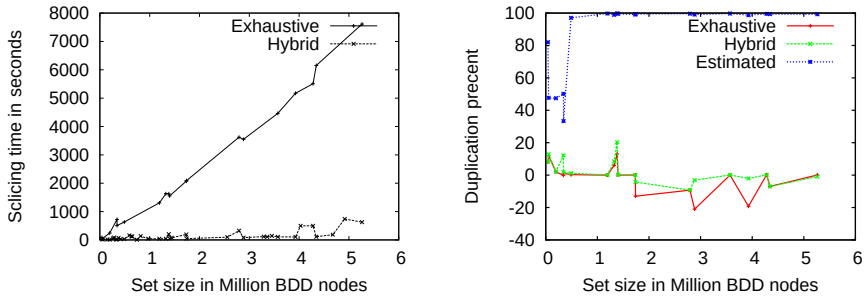
2.1 Size of the Randomly Selected Subset

In this section we discuss the relation between the confidence in finding at least one effective variable and the number of samples. Lemma 1 defines this relation.

Lemma 1. *[Sample size required] Let sup be the size of the support of a set. Let ef be the number of effective slicers in the support ($ef \leq sup$). Let s be the number of randomly selected variables($s \leq sup$). Let pr be the confidence in finding at least one effective slicer out of s samples. Then,* $pr \geq 1 - \left(1 - \frac{ef}{sup}\right)^s$.

the proof is straightforward and omitted for lack of space.

Our experimental results (Figures 5(a), 5(b), explained later) show that the minimum percentage of effective slicers is 4%. Therefore, confidence in finding at least one effective variable converges to 100% exponentially fast in the number of samples. More importantly, it does not depend on the number of variables. If, for example, we want 90% confidence that we will get at least one effective variable and 5% of the variables are effective, we need only 45 samples.



(a) **Run time** (b) **Duplication**

Fig. 2. Comparing the slicing algorithms for support size 687 - 712.

2.2 Experimental Results

We compare three slicing algorithms. The new algorithm `Hybrid`, presented in Figure 1; the exhaustive algorithm `Exh` when working on the entire set of support; and the fast estimation `Est` when working on the entire set of support.

We use three sets of examples, each with different support size. Each set includes varying BDD sizes, from half a a million to 7 millions nodes. The characteristics of the three sets are presented in Table 1.

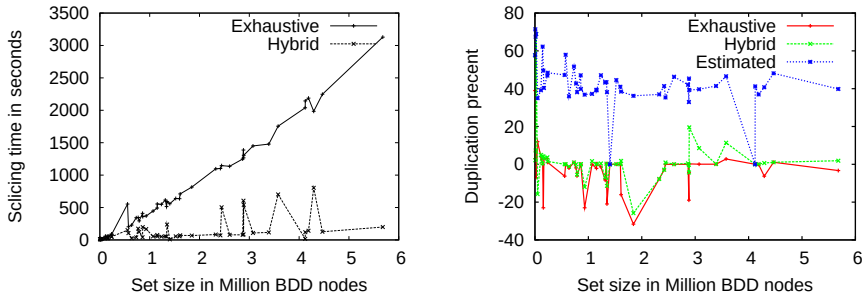
Set	BDD size range	support size range	number of sets of states
Small	0.5 - 3 Million	70	25
Large	0.5 - 6 Million	239 - 255	46
Extra large	0.5 - 7 Million	687 - 712	18

Table 1. Benchmark suite characteristics. For each set of examples we give the BDD size of the sets of states and the support size.

Slicing Efficacy and Memory Overhead We now analyze the run time and the duplication by the different slicing algorithms. Figures 2(a), 3(a) display run time of the slicing algorithms. In each graph the run times of `Hybrid`, and `Exh` algorithms are given compared to the size of the set being sliced.

Figures 2(a), 3(a) show that the run time of the `Exh` algorithm increases proportionally to the BDD size and increases proportionally to the support size, while the `Hybrid` algorithm runs in constant time.

Percentage of duplication is the difference between the size of the set being sliced and the sum of the subsets, in proportion to the size of the set being sliced: $\left(\frac{|f \wedge v|+|f \wedge \overline{v}|}{|f|} - 1\right) * 100$. Figures 2(b), 3(b) compare the percentage of duplication obtained by the `Est` algorithm to that obtained by the `Hybrid` algorithm.



(a) **Run time** (b) **Duplication**

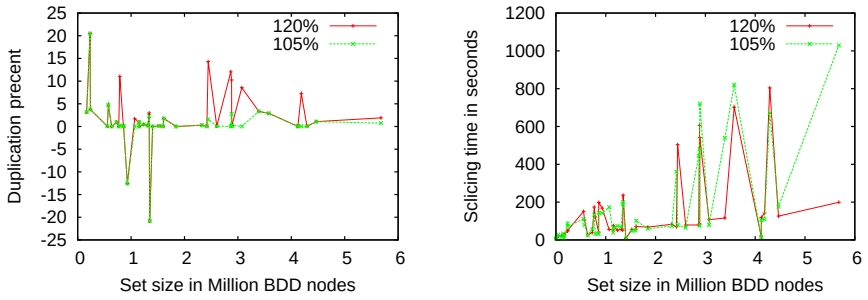
Fig. 3. Comparing the slicing algorithms for support size 239 - 255.

In each graph, the percentages of duplication in the `Est` and in the `Hybrid` algorithms are given compared to the size of the set being sliced. The graphs show that when the support size increases, the slicing by the `Est` algorithm generates much more duplication than `Hybrid`. When the size of the support is 239 - 255 variables (Figure 3(b)), the average percentage of duplication by the `Est` algorithm is 50%, while the `Hybrid` algorithm creates 7% percentage of duplication on average. When the size of the support is 687 - 712 variables (Figure 2(b)), the average percentage of duplication by the `Est` algorithm is 89%, while the `Hybrid` algorithm creates 3% percentage of duplication on average.

Figures 2(b), 3(b) compare the percentage of duplication obtained by the `Exh` algorithm with the `Hybrid` algorithm. The percentages of duplication of the `Exh` and the `Hybrid` algorithms are given compared to the size of the set being sliced. We set the maximum duplication factor δ_2 to be 1.2. We set the minimum reduction factor δ_1 to be 0.85. For all set sizes that are not too small (larger than 100K BDD nodes), the resulting slicer creates less duplication than the maximum duplication factor. When the set size is very small, no effective slicer was found by any of the three phases. Thus, the final phase finds a slicer with duplication factor of 1.5. The small memory requirement of such small sets makes slicing of them not effective.

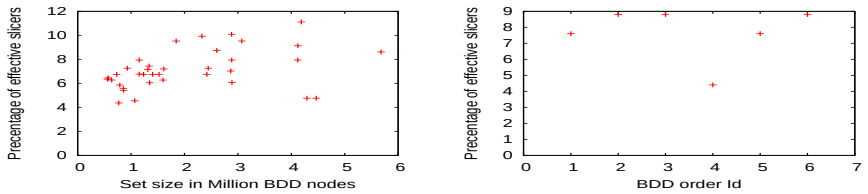
In some cases the percentage of duplication may be negative. This means that the sum of the sizes of the two subsets is less than the original set size. The `Exh` algorithm finds slicers with very small percentage of duplication down to -30%. I.e., the sum is 30% smaller than the original set size. Because the `Hybrid` algorithm stops as soon as it finds an effective slicer, it may miss these.

Changing the Measures of Effectiveness Figures 4(a), 4(b) present the effect of changing the maximum duplication factor from 120% to 105% on run time and on percentage of duplication. In each graph the duplications are given compared to the size of the set being sliced. Figure 4(a) presents the duplications for maximum duplication factors of 105% and 120%. For all set sizes, the final slicer creates duplication which is smaller than the maximum duplication factors;



(a) **Duplication** (b) **Run time**

Fig. 4. Support size 239 - 255. 105 means maximum duplication factor 105%. 120 means maximum duplication factor 120%.



(a) **Support size 239 - 255.** (b) **Support size 242, different orders.**

Fig. 5. Percentage of effective slicers.

hence, the duplication with 105% is less than or equal to the duplication with 120%.

Figure 4(b) presents the run time for duplication factors 105% and 120%. In most cases the run time of the algorithm is longer when the maximum duplication factor is 105%. In cases when the algorithm needs to run more phases, the run time with 105% can take up to five times longer than that with 120%. Since the algorithm uses a random selection, different runs may terminate with different results. The random selection causes sometime deprecate results where the run time takes comparably longer when using larger maximum duplication

Percentage of Effective Slicers The experiments presented in this section demonstrate that for different set sizes, regardless of the BDD order, at least 4% of the variables are effective slicers. Figure 5(a) presents the percentage of effective slicers in different sets. The percentage of effective slicers is given compared to the size of the set being sliced. Figure 5(a) shows that regardless the set size, a minimum of 4% of the slicers are effective. This means that the confidence in finding at least one effective slicer converge to 100% exponentially fast in the number of samples (see Section 2.1).

Figure 5(b) presents the percentage of effective slicers in a single set with different BDD orders. This example has 242 variables in the support and the set size is 2.4 million BDD nodes with the best order. The percentage of effective slicers is given for each order. Figure 5(b) shows that even when we change the BDD order, as happens in the distributed reachability algorithm, a minimum of 4% of effective slicers is maintained.

3 The Checkpoint Restart Algorithm

In this section we briefly describe the iterative BDD-based distributed algorithm for reachability [7]. We explain how to extend this algorithm with checkpoints and how to exploit these checkpoints in order to restart the reachability algorithm when needed, according to some scheduling policy. Finally we present experimental results which show that the associated overhead is negligible.

The basic paradigm followed by the algorithm is to compute the set of states which are reachable from a given set of initial states. At each iteration, starting from the set of initial states, the set R is computed. R consists of reachable states found so far. In addition, the set N of *undeveloped* states is computed. These are states that do not belong in R and are reachable from R in a single step, whose successors have not yet been found.

The distributed algorithm runs on a network of communicating workers with distributed memory. A set of *window functions* defines for each worker the subset of states it *owns*. This set is *complete*, meaning that it covers the whole state space. Worker *id* with window function W_{id} computes the sets R_{id} and N_{id}, both subsets of W_{id}.

Three coordinators control the distributed operation: the *pool manager* keeps track of the free processes; the *exchange coordinator* maintains the window functions of the active workers, and the *small coordinator* joins the windows of workers whose memory utilization decreases below a certain threshold.

Figure 6 describes an extension of the distributed algorithm with checkpoint-restart capability, called **reach_checkpt**. The pseudo–code is described for a single worker. For brevity, we omit the worker subscript *id* from R_{id}, N_{id}, and W_{id}. We remark that the sets R and N, and the window function W, may change during the execution.

The algorithm uses two utility functions to transfer BDDs between a sender and a receiver that may have different bdd order: `bdd2msg` translates a BDD into a compact `msg` data and `msg2bdd` translates the `msg` data back to a BDD after it has been transferred. We remark that the functions `bdd2msg` and `msg2bdd` themselves are not different from the functions described in [8]. Thus, in this paper, we use them as a black box.

The algorithm follows the same lines of the distributed reachability algorithm, except at the end of each iteration workers sometimes store checkpoints. The data stored in a checkpoint consists of R, N, W, the iteration number $\#it$, and its current BDD order *bdd_order* (line 9). The checkpoint of a worker may be stored on a persistent storage system, e.g., a distributed file system such as NFS,

or simply on the private disk of a peer worker (in which case it is assumed the peer worker does not crash in when the worker does).

Recall that the basic reachability paradigm is an iterative, synchronous process. Thus, the collection of all checkpoints from all workers at the end of an iteration forms a consistent view of the global reachability process at that point.

If a restart is needed because of a failure, or due to rescheduling of the reachability process on another distributed system, the collection of checkpoints may set a starting point for pursuing the computation. The restart algorithm searches for a set of checkpoints taken from the same iteration, which forms a complete set of window functions. If an incomplete set is found, indicating that some but not all the workers succeeded in storing checkpoints for the corresponding iteration previous to the abort, then the algorithm searches for a complete set that was stored at the end of a previous iteration. Such a set is guaranteed to exist because the workers follow the same policy, at the end of which iterations checkpoints are stored, and because a previous checkpoint is never removed before the current global checkpoint is known to be complete (e.g., at the end of the next iteration).

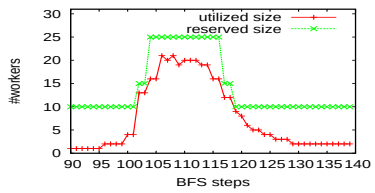
Every active worker in the restarted process is restored using its local checkpoint data, and is replaced by a worker from the free pool in the new distributed system. The new worker restores R and N according to the BDD order *bdd_order* and assumes W as its window function.

```
function reach_checkpt(R, W, N)
 1  Loop until termination()
 2     N = Image(N), split if needed
 3     send non-owned states (N \ W) to their owners
 4     N=N∪ (received states in W from others), split if needed
 5     N=N \ R
 6     R=R ∪ N
 7     Collect_small(R, W, N)
 8     if (W = ∅) return to pool
 9     Check_point(R, W, N, #it, bdd_order)
```

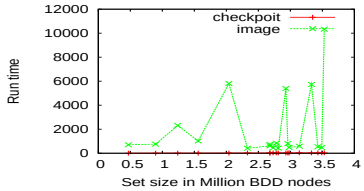
Fig. 6. Pseudo–code for a worker in the distributed reachability computation with checkpoints.

3.1 Experimental Results

The resources consumed by the distributed algorithm are evaluated by considering the following two figures. The *reserved size* is the number of machines carrying out the computation. These machines are either actively taking part in the computation, or they are part of the free pool. If they are part of the free pool, they might not be carrying out any useful computation because they are being reserved as potential additional resources for the reachability computation. The *utilized size* is the number of active non-free workers that are



(a) *reserved size* is the number of machines carrying out the computation. *utilized size* is the number of active non-free workers.



(b) Run time of image computation and the time to store the checkpoint data

actually taking part in the reachability computation. Of course, at any point during computation the utilized size is less than the reserved size.

Figure 7(a) presents the utilized size and reserved size during the distributed reachability computation. The graph shows how the checkpointing mechanism is used in order to reduce reserved size to a minimum. Checkpointing is used to vary the number of workers reserved, starting from a small cluster with only 10 machines. When more than 10 machines are required, the run temporarily halts, a cluster with more machine is reserved, the last checkpoint is moved to new cluster, and the computation is resumed on that cluster. With the larger cluster, the run can reach a further step, while it utilizes at least 10 machines. This way the free pool (of idle machines) is kept small compared to the number of reserved workers.

Yet another contribution of the checkpoint restart mechanism is in the case of termination as a result of failure in one of the resources. In case of a fail, the amount of resources that is wasted grows together with the accumulate number of reserved workers from the start till the iteration where the failure appeared. Furthermore, the more machines take part in the computation, and the more iterations involved in the computation, the higher the chance of a failure. Thus, the importance of the checkpoint restart mechanism increases with the number of iterations to fixpoint, and with the scale of the model checked (as indicated by the reserved size).

Figure 7(b) compares the run time of image computation and the time it takes to store the checkpoint data. For each set size, the graph shows the run time required for image computation and the run time required to store the checkpoint data. The graph shows that for all set sizes the checkpoint run time takes less than 20 seconds. Moreover, if there is no job failure, there is almost no overhead for storing checkpoints.

4 Distributed Reachability Analysis for Very Large Circuites Using 100 PCs

In the previous sections two enhancements to distributed reachability analysis were discussed. This section describes the extensions to the algorithm **reach_checkpt**. These extensions enable high performance distributed reachability analysis for

very large circuites using 100 PCs. With these extensions, the algorithm verifies circuits that could not be verified by any other tool. Furthermore, although an overflow in the required number of workers occurs in several cases, the distributed scheme still reaches an iteration much farther than that reached by the sequential BDD based model checker.

In order to have our distributed scheme scale out, two additional extensions to **reach_checkpt** are needed: *distributed reorder* and *BDD operations timeout*. We discuss the two extensions and then give experimental results.

4.1 Distributed Dynamic Variable Reordering

The dynamic variable reorder suggested by Rudell [13] works well for the sequential algorithm. Here we show how to use it with our distributed approach.

Rudell's algorithm is called by the BDD package according to the growth in the number of BDD nodes. A dynamic reorder threshold *dr_th* determines where the next threshold should be triggered. The threshold is examined after each garbage collection cycle, and variable reordering is triggered if the number of nodes allocated after the garbage collection is greater than *dr_th*. After each invocation of reorder, a new value for *dr_th* is set according to the number of nodes in the new order.

In the distributed scheme the BDD package uses Rudell's algorithm in the same way. However, since there are events such as splits and joins which affect the size of the BDD package, the distributed algorithm also controls the value of *dr_th* externally. The worker forces the BDD packages to adjust the value of *dr_th* after splitting a worker, after which the number of nodes decreases dramatically, and after exchanging nonowned states, after which the number of nodes may decrease or increase.

In addition, in case of overflow during image computation, triggering reorder may reduce the size of the BDD and thus avoid the costly splitting. Therefore, when an overflow occurs after many micro-steps but before the image computation is completed, the worker invokes reorder and then tries to complete the image computation. However, if the BDD package triggers reorder just before the micro-step overflowed, the worker avoids the additional reorder since it is unlikely to prevent the splitting.

4.2 Escape from BDD Operation Livelock Using Timeouts

BDD engines use a cache for previously executed BDD operations. When this cache is used, the run time commonly becomes linear in the sizes of the BDD operators, rather than exponential. Since the size of the cache cannot hold all the BDD operations, the engine replaces old results with new ones. If the result of a replaced BDD operation is required, it will be recalculated. Recalculation increases the run time, and in some cases, can cause the execution of a single BDD operation to proceed for hours.

In the distributed scheme a split can help a single worker if it got stuck on a single BDD operation, because the size of the cache is effectively doubled as a

result of the split and because the split reduces the BDD operation operands. To facilitate the split, a single micro-step is stopped if it turns out to be too long, just as it would be stopped if a memory overflow occurred, and split is invoked. Our experiments show that cases in which a large number of recalculations take hours can be efficiently avoided in this way.

4.3 Experimental results

Our parallel testbed consists of 100 PC machines, each consisting of a two-way 2.4GHz Pentium 4 processors with 1GB memory. For optimal utilization of this configuration we let two workers execute on the same machine. A fast Ethernet connection is used for communication between the nodes. The sequential runs use a PC machine consisting of four way 3.1GHz Pentium 4 processors with 4GB memory.

The distributed algorithm that we tested uses **reach_checkpt** enhanced with the algorithm `Hybrid`, as well as distributed dynamic reordering and the micro-steps timeout. The external model checker used by the distributed algorithm is a high-performance industrial tool from Intel.

We conducted our experiments using examples for which the fixpoint had never been reached before, such as the s1423 design from the ISCAS89 benchmarks. We remark that other examples from this benchmark suite, such as s3330, s1269 and s5378, require only a single process when using Intel's high-performance model checker. Thus, they are not suitable as benchmarks for the distributed system. In addition to s1423, we experimented with six large examples which are components in Intel's designs.

The characteristics of the six test cases are given in Table 2.

Circuit	#vars	Overflow step	Overflow $\mid R \mid$
H21	274	55	3,203,064
H20	276	44	3,922,742
I1	147	98	8,006,120
H11	300	44	5,211,955
I3	793	46	5,557,672
I3s	439	54	7,076,762
s1423	88	14	9,705,214

Table 2. Benchmark suite characteristics. In each example we give the step in which the memory requirements by the sequential model checker overflow and the size of the BDD representing the set of reachable states R at that step.

The distributed reachability analysis results are given in Table 3. Four examples reached fixpoint and the verification completed. Three examples required more workers than were available to us at this point (we did not always have all 100 machines at our disposal), Therefore worker overflow occurred at some step,

but always at a much farther step than that reached by the sequential model checker.

We next compare the results in Table 3 to the results of the high performance industrial SAT model checker tool of Intel. The SAT model checker could not complete the verification of any of the examples. Computing bounded model checking with timeout of 10,000 seconds, SAT reached the bounds of 85 and 94 on I3s and I3, respectively.

Finally we compare these results to previous distributed symbolic model checking [7] and [8]. In [8] a high performance model checker was used by their distributed algorithm, yet s1423 reached only step 17, while the new distributed algorithm reached step 19. Other examples from ISCAS89 are so small that they were completed by the sequential model checker. Other examples from [8] were not made available to the public. In [7] a non-sophisticated model checker was used. Therefore a relatively small example such as s3330 required 54 workers to complete. The high performance model checker used in this work can complete this example using a single worker.

Circuit	Fixpoint	Max workers	Time	$\max_{ite} \sum_i \mid R_i \mid$	at Seq Overflow	
					$\sum_i \mid R_i \mid$	#workers
H21	85	3	23h			
H20	85	9	11h			
I1	139	25	70h	15.5M	6.6M	3
H11	98	7	28.5h	4.4M	1.3M	4
I3	WOvf(60)	>50		47.2M	7.1M	5
I3s	WOvf(118)	>150		358.8M	7.1M	4
s1423	WOvf(19)	>200		208.3M	8.8M	8

Table 3. Distributed reachability on the benchmark suite. Four examples reached fixpoint and verification completed. Three examples required more workers than were available to us and therefore worker overflow occurred. The Max workers column indicates the maximum number of active workers during the computation. The run time when the verification is completed is given in hours. Run time is time elapsed since the first worker starts to run until the last worker finishes the run. Two measures are given for the iteration at which the sequential algorithm overflows: The sum of the sizes of the BDDs representing the subsets of reachable states, and the number of active workers at this iteration

It is especially interesting to compare Tables 2 and 3. It turns out that at the point where the sequential algorithm overflows, the aggregate space requirement for the distributed algorithm (given in the tables as the size of R in BDD nodes) is *smaller* than the corresponding size in the sequential algorithm! This means that the distributed algorithm is more efficient in maintaining its data structures (the BDD which holds R,N), sometimes to a factor of two or more. This comes as a surprise, since common wisdom tells us to expect some overhead and duplication of work, rather than increased efficiency.

The explanation, however, is straightforward. Recall that with the distributed scheme reorder is optimized individually at every worker, taking into account

the worker data only. In this way, bdd reordering by the distributed algorithm is much more efficient than by the sequential algorithm because every worker finds a better order when looking only at its data. The overall effect is an aggregate reduction in the number of BDD nodes, which implies improved overall efficiency.

Acknowledgement: We would like to thank Moshe Vardi for many creative and helpful discussions.

References

1. 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.
2. A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic Model Checking using SAT Procedures Instead of BDDs. In *36th Design Automation Conference*, pages 317–320, 1999.
3. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–171, June 1992. Special Issue: Selections from 1990 IEEE Symposium on Logic in Computer Science.
4. G. Cabodi, P. Camurati, and S. Quer. 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.
5. R. Fraer, G. Kamhi, B. Ziv, M.Y. Vardi, and L. Fix. Prioritized Traversal: Efficient Reachability Analysis for Verification and Falsification. In *Proc. of the 12th International Conference on Computer Aided Verification, LNCS*, 2000.
6. O. Grumberg, T. Heyman, N. Ifergan, and A. Schuster. Achieving speedups in distributed symbolic reachability analysis through asynchronous computation. In *CHARME (to appear)*, 2005.
7. O. Grumberg, T. Heyman, and A. Schuster. A Work-Efficient Distributed Algorithm for Reachability Analysis. In *Proc. of the 15th International Conference on Computer Aided Verification, LNCS*, 2003.
8. T. Heyman, D. Geist, O. Grumberg, and A. Schuster. Achieving Scalability in Parallel Reachability Analysis of Very Large Circuits. *Formal Methods in System Design*, 21(2):317–338, November 2002.
9. K.L. McMillan. Interpolation and SAT-Based Model Checking. In *Proc. of the 15th International Conference on Computer Aided Verification, LNCS*, 2003.
10. A. Narayan, A. Isles, J. Jain, R. Brayton, and A. 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.
11. A. Narayan, J. Jain, M. Fujita, and A. L. Sangiovanni-Vincentelli. Partitioned-ROBDDs. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 547–554. IEEE Computer Society Press, June 1996.
12. Kavita Ravi, Kenneth L. McMillan, Thomas R. Shiple, and Fabio Somenzi. Approximation and Decomposition of Binary Decision Diagrams. In *35th Design Automation Conference*, pages 445–450, 1998.
13. R. Rudell. Dynamic Variable Ordering for Ordered Binary Decision Diagrams. In *Intl. Conf. on Computer Aided Design*, Santa Clara, Ca., November 1993.