Integrating RAM and Disk Based Verification within the Murφ Verifier*

Giuseppe Della Penna[1], Benedetto Intrigila[1], Igor Melatti[1], Enrico Tronci[2], and Marisa Venturini Zilli[2]

[1] Dip. di Informatica, Università di L'Aquila, Coppito 67100, L'Aquila, Italy
{dellapenna,intrigila,melatti}@di.univaq.it
[2] Dip. di Informatica Università di Roma "La Sapienza", Via Salaria 113, 00198 Roma, Italy
{tronci,zilli}@dsi.uniroma1.it

Abstract. We present a verification algorithm that can automatically switch from *RAM based* verification to *disk based* verification without discarding the work done during the RAM based verification phase. This avoids having to choose beforehand the *proper* verification algorithm. Our experimental results show that typically our integrated algorithm is as fast as (sometime faster than) the fastest of the two *base* (i.e. RAM based and disk based) verification algorithms.

1 Introduction

Disk based verification algorithms [4,5,8,3,2] turn out to be very useful to counteract *state explosion* (i.e. the huge amount of memory required to complete state space exploration). However, using a disk based verification algorithm for a task that could have been completed just using a RAM based verification algorithm results in a waste of time. Unfortunately it is hard to predict beforehand the size of the set of *reachable states* so as to use the *proper* (RAM based or disk based) verification algorithm.

In this paper we present an explicit verification algorithm that can automatically switch from RAM based verification to disk based verification without discarding the work done during the RAM based verification phase. This avoids having to choose beforehand the kind of verification algorithm, thus saving on the verification time.

Our main contributions can be summarized as follows.

– We present (Section 3) an integration scheme (we call it *serialization scheme*) for the RAM based verification algorithm presented in [9] and the disk based verification algorithm presented in [2].
– We present (Section 4) experimental results on using our serialization scheme implemented within the Murφ verifier. Our experimental results show that

* This research has been partially supported by MURST projects: MEFISTO and SAHARA

```
FIFO_Queue Q; HashTable T;
bfs(init_states, next) {
foreach s in init_states Enqueue(Q, s); /*load Q with init states*/
foreach s in init_states Insert(T, s); /*mark init states as visited*/
while (Q is not empty) { s = Dequeue(Q); /* current state */
 foreach s' in next(s)  /* expand current state */
   if (s' is not in T) {Insert(T, s'); Enqueue(Q, s');}}}
```

Fig. 1. Explicit Breadth First Visit (RAM based)

typically our integrated algorithm is as fast as (sometime faster than) the fastest of the two *base* (i.e. RAM based and disk based) verification algorithms. This means that on a single machine we are able to run two verification attempts (RAM based and then disk based) within the time taken by the first terminating verification attempt.

2 State Space Exploration Algorithms

Our goal is to devise a serialization scheme for the RAM based state exploration algorithm presented in [9] (CBF, for *Cached Breadth First visit* in the following) and the disk based state exploration algorithm presented in [2] (DBF, for *Disk Breadth First visit* in the following).

Figure 1 shows the algorithm and data structures used by a *Breadth First* (BF) visit. Both the `Enqueue()` operation on BF queue `Q` as well as the `Insert()` operation on the visited states hash table `T` in Figure 1 may fail because of lack of memory. In such cases the BF visit stops with an *out of memory* message.

Algorithm CBF [9] implements BF queue `Q` on disk and, most importantly, replaces with a cache table the hash table `T` used by the *standard* BF visit in Figure 1. Using a cache table rather than a hash table means that, upon a collision, CBF may forget visited states and, as a result, it may revisit states. To prevent nontermination due to revisiting states, CBF terminates when the collision rate (i.e. the ratio between the number of collisions and the number of insertions) is above a user given threshold.

Algorithm DBF [2] is a disk based version of CBF. DBF uses a hash table `M` to store signatures (e.g. see [6]) of *recently* visited states, a file `D` to store signatures of *all* visited states (*old states*) and splits the BF queue `Q` of CBF into two queues: `Q_ck` and `Q_unck`. DBF uses the *checked queue* `Q_ck` to store the states in the currently explored BF level and uses the *unchecked queue* `Q_unck` to store the states that are candidates to be on the next BF level. At the end of each BF level, DBF uses file `D` to remove old states from `Q_unck`.

Note that with DBF all data structures that grow with the state space size (namely: `D`, `Q_ck`, `Q_unck`) are on disk, thus DBF bottleneck is computation time, rather than memory space.

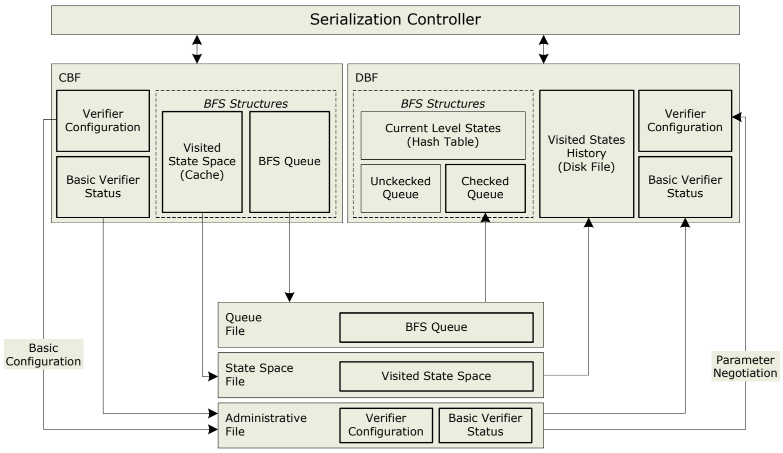

Fig. 2. Serialization scheme for CBF, DBF.

3 Serializing CBF and DBF

In our context, a *serialization* scheme is an algorithm that allows us to stop the current verification task and to *resume* it possibly using a different algorithm without losing the work previously done.

Let $\mathcal{S}$ be a FSS and Time$(A, \mathcal{S})$ the time needed by algorithm A to complete state space exploration of $\mathcal{S}$. A *serialization* scheme for state space exploration algorithms A and B is a state space exploration algorithm $[A, B]$ s.t. there exist time instants $0 \leq t_1 < t_2 \leq$ Time$([A, B], \mathcal{S})$ s.t. for all $t \leq t_1$, $[A, B]$ *behaves as* A and for all $t \geq t_2$, $[A, B]$ *behaves as* B.

Of course a serialization scheme for algorithms A and B is interesting only if the ratio Time$([A, B], \mathcal{S})$/min(Time$(A, \mathcal{S})$, Time$(B, \mathcal{S})$) (*serialization ratio*) is close to 1 for most FSS $\mathcal{S}$. This means that on a single machine we are able to run two verification attempts (namely A and B) within the time taken by the first terminating verification attempt among the two.

In this section we present a serialization scheme for the RAM based state space exploration algorithm CBF [9] and the disk based state space exploration algorithm DBF [2].

To switch from CBF to DBF we have to save on disk the *current status* of CBF in such a way that *CBF status disk image* can then be used to initialize DBF data structures. Figure 2 summarizes our serialization scheme.

CBF status disk image includes the following elements:
1. A file (*queue file* in Figure 2) containing BF queue `Q` of Figure 1.
2. A file (*state space file* in Figure 2) containing the visited states (namely, cache table `T` of Figure 1).
3. A file (*administrative file* in Figure 2) containing *administrative* information about the verification process. For example, such a file may contain: compression options with which CBF has been started (e.g. bit compression [1], hash compaction [6]); random seeds used in various hashing functions (e.g. in the

computation of state signatures [6]), the BF level reached in the BF visit, the number of states visited so far, etc.
In our serialization scheme switching from CBF to DBF is normally requested by the *serialization controller* (Figure 2) when CBF collision rate becomes greater than a user given threshold.

Serialization is requested by sending a *signal* to (the suitably modified) CBF. Indeed, to keep easy and efficient our serialization scheme, we only allow CBF to be stopped when it is easy to dump CBF current status to disk. Namely, before a new state is dequeued from the verification queue `Q`. The CBF queue `Q`, the cache `T` and the parameters are saved to disk in the respective files (Figure 2).

To initialize DBF using the disk image of CBF, first DBF parameters defining state format are overridden by CBF parameters saved in the administrative file on disk. This is needed to ensure compatibility between the data format saved on disk and DBF data format.

CBF queue `Q` stored on disk is then loaded and connected to the DBF checked queue `Q_ck`. This is the best choice since `Q` has already been *checked* w.r.t `T`. DBF unchecked queue `Q_unck` and DBF hash table `M` are left empty. DBF history file `D` is initialized with the set of visited states in `T` (Figure 2). After the above steps DBF can start normally.

4 Experimental Results

We implemented both algorithms CBF and DBF within the Murφ verifier. This was done as illustrated, respectively, in [9] and [2]. The resulting verifiers are called, respectively, *Cached–Murφ* and *Disk–Murφ*. Thus, not surprisingly, we implemented the serialization scheme outlined in Section 3 within the Murφ verifier. We call *Serial–Murφ* the resulting verifier. Unless otherwise stated, in this Section CBF denotes Cached–Murφ, DBF denotes Disk–Murφ and [CBF, DBF] denotes Serial–Murφ.

Serial–Murφ runs first Cached–Murφ until it completes the verification or the collision rate hits a user given threshold γ (set to 0.1 in our experiments). If the collision rate is greater than or equal to γ, Serial–Murφ switches to Disk–Murφ.

Note that, from [9], we know that Cached–Murφ behaves as standard Murφ (both for explored states and verification time) if the collision rate is low. The limitation to 10% of collision rate used in our experiments makes Cached–Murφ very similar to standard Murφ in terms of performance.

In this Section we report the experimental results we obtained using [CBF, DBF]. Our goal is of course to assess effectiveness of our serialization scheme. Let $\mathcal{S}$ be the FSS to be verified. Effectiveness in our case means that the serialization ratio (Section 3) (Time([CBF, DBF], $\mathcal{S}$)/ min(Time(CBF, $\mathcal{S}$), Time(DBF, $\mathcal{S}$))) ≈ 1.

We know [2] that if CBF has enough RAM then Time(CBF, $\mathcal{S}$) $<$ Time(DBF, $\mathcal{S}$). In such cases [CBF, DBF] never switches to DBF and thus behaves as CBF. Thus in such cases (Time([CBF, DBF], $\mathcal{S}$)/min(Time(CBF, $\mathcal{S}$), Time(DBF, $\mathcal{S}$))) ≈ 1 holds.

Table 1. Serial–Murφ versus Disk–Murφ.

Protocol	Mem	0.5	0.4	0.3
`eadash`	Rules	0.725	0.753	0.879
	States	0.989	0.955	1.011
	Time	1.009	1.041	1.024
`ldash`	Rules	0.754	0.749	0.816
	States	0.985	0.923	0.945
	Time	0.870	1.046	1.038
`mcslock1`	Rules	0.594	0.587	0.510
	States	0.820	0.770	0.632
	Time	0.837	1.109	0.836
`n_peterson`	Rules	0.635	0.689	0.739
	States	1.002	0.984	0.951
	Time	0.959	0.952	1.009
`sym.cache3`	Rules	0.709	0.568	0.654
	States	0.966	0.733	0.767
	Time	1.029	0.988	1.057
`kerb`	Rules	0.431	0.466	0.779
	States	0.876	0.760	0.964
	Time	0.875	0.859	1.049
`list6`	Rules	0.538	0.673	0.752
	States	0.756	0.857	0.888
	Time	1.033	1.000	1.110
`mcslock2`	Rules	0.673	0.760	0.843
	States	1.006	1.030	1.050
	Time	0.984	1.169	1.040
`sci`	Rules	0.607	0.709	0.554
	States	0.867	0.975	0.694
	Time	1.013	0.976	0.971

Hence the interesting cases for us are those in which CBF does not have enough RAM to complete the verification task. In such cases Time(CBF, $\mathcal{S}$) = ∞, thus min(Time(CBF, $\mathcal{S}$), Time(DBF, $\mathcal{S}$)) = Time(DBF, $\mathcal{S}$). Thus we need to check whether Time([CBF, DBF], $\mathcal{S}$)/Time(DBF, $\mathcal{S}$) ≈ 1, which means that [CBF, DBF] completes verification taking about the same time as DBF (even after trying CBF first).

To carry out our experiments we used the benchmark protocols included in the Murφ distribution [1] that need at least (about) 100Kb of memory to be verified by standard Murφ, and the `kerb` protocol from [7].

First, for each protocol p in our benchmark we determined the minimum amount of memory $M(p)$ needed by Murφ (version 3.1 from [1]) to complete the verification. Then we compared Serial–Murφ performances with those of Disk–Murφ for decreasing fractions of such a memory amount. Namely, we ran each protocol p with memory limits $0.5M(p)$, $0.4M(p)$ and $0.3M(p)$.

In this way, we experimented our approach under conditions in which Serial–Murφ at some point is forced to switch to Disk–Murφ since there is not enough RAM for Cached–Murφ to complete its verification task.

Our results are shown in Table 1, where columns correspond to the memory fraction used for the experiment (e.g. column 0.5 corresponds to half of the needed memory), and rows report the results obtained for a protocol in terms of fired rules, visited states and time to complete the verification. To highlight the usefulness of our approach, we report these results as ratios between the values obtained by Serial–Murφ and the values obtained using Disk–Murφ on the same protocols with the same memory restrictions. Thus row Time in Figure 1 gives us the serialization ratio.

The results in Table 1 show that using Serial–Murφ two verification attempts (namely: Cached–Murφ and then Disk–Murφ) take about the same time of the fastest terminating one (namely Disk–Murφ). Indeed, in Table 1 Time rows range from 1.1 (i.e. a time overhead of 10%, worst case) to 0.8 (i.e. Serial–Murφ is 20% faster than Disk–Murφ), averaging to 0.99.

From Table 1 we see that sometimes Serial–Murφ is faster than Disk–Murφ. This is because Serial–Murφ starts verification using a RAM based algorithm (CBF) which is faster than the disk based algorithm (DBF) to which Serial–Murφ switches only after part of the verification work has been done (in RAM).

Summing up, the results in Table 1 show that Serial–Murφ is typically as fast as (sometime faster than) the fastest terminating one among Cached–Murφ and Disk–Murφ. Thus, using Serial–Murφ we can run two verification attempts in the time normally taken by one.

5 Conclusions

We presented a verification algorithm that can automatically switch from *RAM based* verification to *disk based* verification without discarding the work done during the RAM based verification phase.

Our experimental results show that typically our integrated algorithm is as fast as (sometime faster than) the fastest of the two *base* (i.e. RAM based and disk based) verification algorithms. This means that on a *single machine* we are able to run *two verification attempts* (RAM based and then disk based) within the time taken by the first terminating verification attempt.

References

1. url: `http://sprout.stanford.edu/ dill/ murphi.html`.
2. G. Della Penna, B. Intrigila, E. Tronci, and M. Venturini Zilli. Exploiting transition locality in the disk based murφ verifier. In *Proc. of 4th International Conference on "Formal Methods in Computer Aided Verification" (FMCAD)*, LNCS, Portland, Oregon, USA, Nov 2002. Springer.
3. J. V. Sanghavi, R. K. Ranjan, R. K. Brayton, and A. Sangiovanni-Vincentelli. High performance bdd package by exploiting memory hierarchy. In *33rd IEEE Design Automation Conference*, 1996.
4. U. Stern and D. Dill. Parallelizing the murφ verifier. In *Proc. 9th Int. Conference on: Computer Aided Verification*, volume 1254, pages 256–267, Haifa, Israel, 1997. LNCS, Springer.
5. U. Stern and D. Dill. Using magnetic disk instead of main memory in the murφ verifier. In *Proc. 10th Int. Conference on: Computer Aided Verification*, volume 1427, pages 172–183, Vancouver, BC, Canada, 1998. LNCS, Springer.
6. U. Stern and D. L. Dill. Improved probabilistic verification by hash compaction. In *IFIP WG 10.5 Advanced Research Working Conference on: Correct Hardware Design and Verification Methods (CHARME)*, pages 206–224, 1995.
7. url: `http://verify.stanford.edu/ uli/ research.html`.
8. T. Stornetta and F. Brewer. Implementation of an efficient parallel bdd package. In *33rd IEEE Design Automation Conference*, pages 641–644, 1996.
9. E. Tronci, G. Della Penna, B. Intrigila, and M. Venturini Zilli. Exploiting transition locality in automatic verification. In *IFIP WG 10.5 Advanced Research Working Conference on: Correct Hardware Design and Verification Methods (CHARME).* LNCS, Springer, Sept 2001.