Exploiting Transition Locality in the Disk Based Murφ Verifier[*]

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

[1] Dip. di Informatica, Università di L'Aquila, Coppito 67100, L'Aquila, Italy
{gdellape,intrigil}@univaq.it

[2] Dip. di Informatica Università di Roma "La Sapienza",
Via Salaria 113, 00198 Roma, Italy
{tronci,zilli}@dsi.uniroma1.it

Abstract. The main obstruction to automatic verification of *Finite State Systems* is the huge amount of memory required to complete the verification task (*state explosion*). This motivates research on distributed as well as disk based verification algorithms.
In this paper we present a disk based Breadth First *Explicit State Space Exploration* algorithm as well as an implementation of it within the Murφ verifier. Our algorithm exploits *transition locality* (i.e. the statistical fact that most transitions lead to unvisited states or to recently visited states) to decrease disk read accesses thus reducing the time overhead due to disk usage.
A disk based verification algorithm for Murφ has been already proposed in the literature. To measure the time speed up due to locality exploitation we compared our algorithm with such previously proposed algorithm. Our experimental results show that our disk based verification algorithm is typically more than 10 times faster than such previously proposed disk based verification algorithm.
To measure the time overhead due to disk usage we compared our algorithm with RAM based verification using the (standard) Murφ verifier with enough memory to complete the verification task. Our experimental results show that even when using 1/10 of the RAM needed to complete verification, our disk based algorithm is only between 1.4 and 5.3 times (3 times on average) slower than (RAM) Murφ with enough RAM memory to complete the verification task at hand.
Using our disk based Murφ we were able to complete verification of a protocol with about 10^9 reachable states. This would require more than 5 gigabytes of RAM using RAM based Murφ.

1 Introduction

State Space Exploration (*Reachability Analysis*) is at the very heart of all algorithms for automatic verification of concurrent systems. As well known, the

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

[**] Corresponding Author: Enrico Tronci. Tel: +39 06 4991 8361 Fax: +39 06 8541 842

main obstruction to automatic verification of *Finite State Systems* (FSS) is the huge amount of memory required to complete state space exploration (*state explosion*).

For protocol like systems, *Explicit* State Space Exploration often outperforms *Symbolic* (i.e. OBDD based, [1,2]) State Space Exploration [8]. Since here we are mainly interested in protocol verification, we focus on explicit state space exploration. Tools based on explicit state space exploration are, e.g., SPIN [6, 14] and Murφ [4,11].

In our context, roughly speaking, two kinds of approaches have been studied to counteract (i.e. delay) state explosion: *memory saving* and *auxiliary storage.*

In a memory saving approach essentially one tries to reduce the amount of memory needed to represent the set of visited states. Examples of the memory saving approach are, e.g., in [23,9,10,17,18,7].

In an auxiliary storage approach one tries to exploit disk storage as well as distributed processors (network storage) to enlarge the available memory (and CPU). Examples of this approach are, e.g., in [15,16,12,20,13,5].

Exploiting *statistical properties* of protocol transition graphs it is possible to trade space with time [21,22], thus enlarging the class of systems for which automatic verification is feasible. In particular in [21] it has been shown that protocols exhibit *locality.* That is, w.r.t. levels of a *Breadth First Search* (BFS), state transitions tend to be between states belonging to close levels of the transition graph. In [21] an algorithm was also presented exploiting locality in order to save RAM as well as an implementation of such an algorithm within the Murφ verifier. It is then natural and worth doing looking for a way to exploit locality also when using a disk based state exploration algorithm.

In this paper we present a *Disk based Breadth First Search* (DBFS) algorithm that exploits transition locality. Our algorithm is obtained by modifying the DBFS algorithm presented in [16]. Our main results can be summarized as follows.

– We present a DBFS algorithm that is able to exploit transition locality. Essentially, our algorithm is obtained from the one in [16] by using only a suitable subset of the states stored on disk to clean up the *unchecked states* BFS queue of [16]. By reducing disk read accesses we also reduce our time overhead w.r.t. a RAM based BFS state space exploration.
– We implemented our algorithm within the Murφ verifier. As the algorithm in [16], our algorithm is compatible with all state reduction techniques implemented in the Murφ verifier.
– We run our DBFS algorithm on some of the protocols included in the standard Murφ distribution [11]. Our experimental results can be summarized as follows.
    - Even when using 1/10 of the RAM needed to complete verification, our disk based Murφ is only between 1.4 and 5.3 times slower (3 times on average) than (RAM based) standard Murφ [11] with enough RAM to complete the verification task at hand.

  - Our disk based algorithm is typically more than 10 times faster than the disk based algorithm presented in [16].

– Using our disk based Murφ we were able to complete verification of a protocol with almost 10^9 reachable states. Using standard Murφ this protocol would require more than 5 gigabytes of RAM.

2 Transition Locality for Finite State Systems

In this section we define (from [21]) our notion of locality for transitions. For our purposes, a protocol is represented as a *Finite State System.*

A *Finite State System* (FSS) $\mathcal{S}$ is a 4-tuple (S, I, A, R) where: S is a finite set (of states), $I \subseteq S$ is the set of initial states, A is a finite set (of transition labels) and R is a relation on $S \times A \times S$. R is usually called the *transition relation* of $\mathcal{S}$.

Given states $s, s' \in S$ and $a \in A$ we say that there is a transition from s to s' labeled with a iff $R(s, a, s')$ holds. We say that there is a transition from s to s' (notation $R(s, s')$) iff there exists $a \in A$ s.t. $R(s, a, s')$ holds. The set of successors of state s (notation $\texttt{next}(s)$) is the set of states s' s.t. $R(s, s')$.

The set of *reachable states* of $\mathcal{S}$ (notation Reach) is the set of states of $\mathcal{S}$ reachable in 0 or more steps from I. Formally, Reach is the smallest set s.t.: 1. $I \subseteq$ Reach; 2. for all $s \in$ Reach, $\texttt{next}(s) \subseteq$ Reach.

The transition relation R of a given system defines a graph (*transition graph*). Computing Reach (*reachability analysis*) means visiting (exploring) the transition graph starting from the initial states in I. This can be done, e.g., using a *Depth First Search* (DFS) or a *Breadth First Search* (BFS). In the following we will focus on BFS.

As well known a BFS defines *levels* on the transition graph. Initial states (i.e. states in I) are at level 0. The states in ($\texttt{next}(I) - I$) (states reachable in one step from I and not in I) are at level 1, etc.

Formally we define the set of states *at level* k (notation $L(k)$) as follows. $L(0) = I$, $L(k + 1) = \{s' \mid \exists s \text{ s.t. } s \in L(k) \text{ and } R(s, s') \text{ and } s' \notin \cup_{i=0}^{i=k} L(i)\}$.

Given a state $s \in$ Reach we define level(s) $= k$ iff $s \in L(k)$. That is level(s) is the level of state s in a BFS of $\mathcal{S}$.

The set Visited(k) of states *visited* (by a BFS) by level k is defined as follows. Visited(k) $= \cup_{i=0}^{i=k} L(i)$.

Informally, *transition locality* means that for most transitions source and target states will be in levels not too far apart.

Let $\mathcal{S}$ = (S, I, A, R) be an FSS. A transition in $\mathcal{S}$ from state s to state s' is said to be *k-local* iff $|\text{level}(s') - \text{level}(s)| \leq k$.

In [21] it is shown experimentaly the following fact. For most protocols, we have that for most states more that 75% of the transitions are 1-local.

```
/* Global Variables */
hash table M;  /* main memory table */
file D;        /* disk table */
FIFO queue Q_ck;  /* checked state queue */
FIFO queue Q_unck;  /* unchecked state queue */
int disk_cloud_size; /* number of blocks to be read from file D */
```

Fig. 1. Data Structures

3 A Disk Based State Space Exploration Algorithm Exploiting Transition Locality

Magnetic disk read/write times are much larger than RAM read/write times. Thus, not surprisingly, the main drawback of DBFS (*Disk based Breadth First Search*) w.r.t. RAM-BFS (*RAM based Breadth First Search*) is the time overhead due to disk usage. On the other hand, because of *state explosion*, memory is one of the main obstructions to automatic verification. Thus using magnetic disks to increase the amount of memory available during verification is very appealing.

In [16] a DBFS algorithm has been proposed for the Murφ verifier. Here we show that by exploiting *transition locality* (Section 2) the algorithm in [16] can be improved. In particular, disk accesses for reading can be reduced. This decreases the time overhead (w.r.t. a RAM-BFS) due to disk usage.

As in [16] we actually have two DBFS algorithms: one for the case in which *hash compaction* [17,18] (Murφ option `-c`) is enabled and one for the case in which *hash compaction* is not enabled. As the algorithm in [16] our algorithm can adjust for both cases. In the following we only present the version which is compatible with the hash compaction option. When hash compaction is not enabled the algorithm is actually simpler and can be easily obtained from the algorithm compatible with the hash compaction option.

In the following we call LDBFS our *L*ocality based DBFS algorithm. Figs. 1, 2, 3, 4, 5, 7 define our LDBFS using a C like programming language.

```
Search()
{
/* initialization */
M = empty; D = empty; Q_ck = empty; Q_unck = empty;
for each startstate s {Insert(s);} /* startstate generation */
do   /* search loop */
 {  while (Q_ck is not empty)
        {
          s = dequeue(Q_ck);
          for all s' in successors(s) {Insert(s');}
        }  /* while */
    Checktable();
 } while (Q_ck is not empty); /* do */ } /* Search()*/
```

Fig. 2. Function `Search()`

3.1 Data Structures

The data structures used by LDBFS are in Fig. 1 and are essentially the same as the ones used in [16]. We have: a table `M` to store signatures of *recently* visited states; a file `D` to store signatures of *all* visited states (*old states*); a *checked queue* `Q_ck` to store the states in the BFS level currently explored by the algorithm (BFS *front*); an *unchecked queue* `Q_unck` to store pairs (`s`, `h`) where `s` is a state candidate to be on the next BFS level and `h` is the signature of state `s`.

As in [16] state signatures in `M` do not necessarily represent *all* visited states. In `M` we just have *recently* visited states. Using the information in `M` we build the unchecked queue `Q_unck`, i.e. the set of states candidate to be on the next BFS level. Note that the states in `Q_unck` may be *old* (i.e. previously visited) since using `M` we can only avoid re-inserting in `Q_unck` recently visited states. As in [16] we use disk file `D` to remove old state signatures from table `M` as well as to *check* `Q_unck` to get rid of old states. The result of this checking process is the checked queue `Q_ck`.

The main difference between our algorithm and the one in [16] is that in the checking process we only use a subset of the state signatures in `D`. In fact we divide `D` into blocks and then use only some of such blocks to clean up `M` and `Q_unck`. The global variable `state_cloud_size` holds the number of blocks of `D` we use to remove old state signatures from table `M`. Our algorithm *dynamically* adjust the value of `state_cloud_size` during the search.

Using only a subset of the states in `D` decreases disk usage and thus speeds up verification. Note however that in [16] the checked queue `Q_ck` only contains *new* (i.e. not previously visited) states whereas in our case `Q_ck` may also contain some *old* (i.e. already visited) state. As a result our algorithm may mark as new (unvisited) a state that indeed is old (visited). This means that some state may be visited more than once and thus appended to file `D` more than once. However, thanks to *transition locality* (Section 2), this does not happen too often. It is exactly this statistical property of transition graphs that makes our approach effective.

Table `M` is in main memory (RAM) whereas file `D` is on disk. We use disk memory also for the BFS queues `Q_ck`, `Q_unck` which instead are kept in main memory in the algorithm proposed in [16]. Our low level algorithm to handle disk queues `Q_ck` and `Q_unck` is exactly the same one we used in *Cached Murφ* [21,3] for the same purpose, thus we do not show it here.

Note that all the data structures that grow with the state space size (namely: `D`, `Q_ck`, `Q_unck`) are on disk in LDBFS. In [16] `D` is on disk, however state queues are in RAM. Since states in the BFS queue are not compressed [11] we have that for large verification problems the BFS queue can be a limiting factor for [16]. For this reason in LDBFS we implemented state queues on disk.

3.2 Function `Search()`

Function `Search()` (Fig. 2) is the same as the one used in the DBFS algorithm in [16].

```
Insert(state s)
{ h = hash(s); /* compute signature of state s */
  if (h is not in M)
   { insert h in M;
     enqueue((s, h), Q_unck);
     if (M is full) Checktable(); } /* if */  } /* Insert()  */
```

Fig. 3. Function `Insert()`

Function `Search()` is a *Breadth First Search* using the checked queue `Q_ck` as the *current level* state queue.

Function `Search()` first loads the BFS queue (`Q_ck`) with the initial states. Then `Search()` begins dequeuing states from `Q_ck`. For each successor `s'` of each state dequeued from `Q_ck`, `Search()` calls `Insert(s')` to store potentially new states in `M` as well as in `Q_unck`.

When queue `Q_ck` becomes empty it means that all transitions from all states in the current BFS level have been explored. Thus we want to move to the next BFS level. Function `Search()` does this by calling function `Checktable()` which refills the checked queue `Q_ck` with fresh (non visited) states, if any, from the unchecked queue `Q_unck`. If, after calling `Checktable()`, `Q_ck` is still empty it means that all reachable states have been visited and the BFS ends.

3.3 Function `Insert()`

Functions `Insert()` (Fig. 3) is the same as the one used in the DBFS algorithm in [16].

Consider the pair (`s`, `h`), where `s` is a state whose signature is `h`. If signature `h` is not in table `M` then `Insert(s)` inserts pair (`s`, `h`) in the unchecked queue `Q_unck` and signature `h` in table `M`.

When `M` is full, function `Insert()` calls function `Checktable()` to clean up `M` as well as the queues. Function `Checktable()` is also called at the end of each BFS level (when `Q_ck` is empty).

3.4 Exploiting Locality in State Filtering

Function `Checktable()` in the DBFS algorithm in [16] uses all state signatures in disk file `D` to remove old states from `Q_unck`. Exploiting locality (Section 2) here we are able to use only a fraction of the state signatures on disk `D` to clean up table `M` and queue `Q_unck`. Disk usage is what slows down DBFS w.r.t. a RAM-BFS. Thus, by reading less states from disk, we save w.r.t. [16] some of the time overhead due to disk (read) accesses.

The rationale of our approach stems from the following observations.

First we should note that state signatures are appended to `D` in the same order in which new states are discovered by the BFS. Thus, as we move towards the *tail* of file `D` we find (signatures of) states whose BFS level is closer and closer to the *current* BFS level, i.e. the BFS level reached by the search. From [21] we

```
Checktable() /* old/new check for main memory table */
{
  /* Disk cloud defined in Section 3.4 */

/*number of states deleted from M that are in disk cloud*/
deleted_in_cloud = 0;

/*number of states deleted from M that are on disk but
  not in disk cloud*/
deleted_not_in_cloud = 0;

/* Randomly choose indexes of disk blocks to read (disk cloud) */
DiskCloud = GetDiskCloud();

/* something_not_in_cloud is true iff
   there exists a state on disk that is not in the disk cloud */
if (there exists a disk block not selected in DiskCloud)
     something_not_in_cloud = true;
else something_not_in_cloud = false;

Calibration_Required = QueryCalibration();

for each Block in D {
  if (Block is in DiskCloud or Calibration_Required) {
   for all state signatures h in Block {
     if (h is in M) {
       remove h from M;
       if (Block is in DiskCloud) { deleted_in_cloud++; }
       else /* Block is not in DiskCloud */
          {deleted_not_in_cloud++; }}}}}

/* remove old states from state queue and add new states to disk */
while (Q_unck is not empty) {
 (s, h) = dequeue(Q_unck);
 if (h is in M) {append h to D; remove h from M; enqueue(Q_ck, s);}}

/* clean up the hash table */
remove all entries from M;

/* adjust disk cloud size, if requested */
if (Calibration_Required)
{ if (something_not_in_cloud and
     (deleted_in_cloud + deleted_not_in_cloud > 0))
  {Calibrate(deleted_in_cloud,deleted_not_in_cloud);}

  if (disk access rate has been too long above a given critical limit)
  {reset disk cloud size to its initial value with given probability P;}
}  /* if Calibration_Required */    }  /* Checktable()  */
```

Fig. 4. Function `Checktable()` (state filtering)

```
GetDiskCloud()
{
   Randomly select disk_cloud_size blocks from disk
   according to the probability distribution shown in Fig. 6
   Return the indexes of the selected blocks.
}
```

Fig. 5. Function `GetDiskCloud()`

know that most transitions are *local*, i.e. they lead to states that are on BFS levels close to the current one. This means that *most* of the old states in `M` can be detected and removed by only looking at the *tail* of file `D`.

We can take advantage of the above remarks by using the following approach.

We divide the disk file `D` into *blocks*. Rather than using the whole file `D` in the `Checktable()` (as done in [16]) we only use a subset of the set of disk blocks. We call such a subset *disk cloud*. The disk cloud is created by selecting at random several disk blocks. Selection probability of disk blocks is not uniform. Instead, to exploit locality, disk block selection probability increases as we approach the tail of `D` (see Fig. 6).

In [21] it is shown that locality allows us to save about 40% of the memory required to complete verification. This suggests to just use say 60% of the disk blocks. Thus the size (number of blocks) of the disk cloud should be 60% of the number of disk blocks. This works fine. However we can do more. Our experimental results show that, most of the time, we need much less than 60% of the disk blocks to carry out the clean up implemented by function `Checktable()`. Thus we *dynamically* adjust the fraction of disk blocks used by function `Checktable()`.

3.5 Function `Checktable()`

Function `Checktable()` (Fig. 4), using disk file `D`, removes signatures of old (i.e. visited) states from table `M`. Then, using such cleaned `M`, `Checktable()` removes old states from the unchecked queue `Q_unck`. Finally, `Checktable()` moves the states that are in the (now cleaned) unchecked queue `Q_unck` to the checked queue `Q_ck`.

3.6 Disk Cloud Creation

Function `GetDiskCloud()` (Fig. 5) is called by function `Checktable()` to create our disk cloud. Function `GetDiskCloud()` selects `disk_cloud_size` disk blocks according to the probability curve shown in Fig. 6.

We number disk blocks starting from 0 (oldest block). Thus the lower the disk block index the older (closer to the head of file `D`) the disk block.

On the x axis of Fig. 6 we have the relative disk block index ρ, i.e. $\rho =$ <block index> /<number of blocks>. E.g. $\rho = 0$ is the (relative index of the) first (oldest) disk block inserted in disk `D`, whereas $\rho = 1$ is the last (newest) disk block inserted. On the y axis of Fig. 6 we have the probability of selecting a disk block with a given ρ.

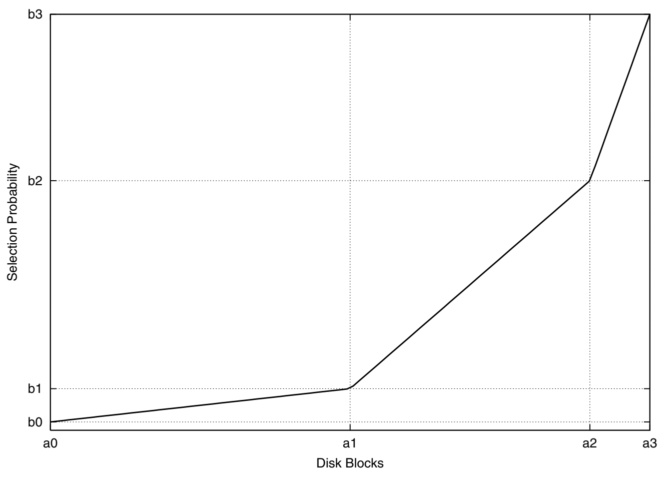

Fig. 6. Probability curve for disk cloud block selection (used by `GetDiskCloud()`)

The selection probability curve in Fig. 6 ensures that the most recently created blocks (ρ close to 1) are selected with a higher probability than old blocks thus exploiting transition locality [21]. Note that, defensively, the selection probability of old blocks (ρ close to 0) is $b_0 > 0$. This is because we want to have some *old* blocks to remove occasional *far back states* (i.e. states belonging to an old BFS level far from the current one) reached by *occasional* non local transitions.

Function `GetDiskCloud()` returns to `Checktable()` the indexes of the selected blocks.

Since our min and max values for the relative disk block indexes are, respectively, 0 and 1, in Fig. 6 we have $a_0 = 0$ and $a_3 = 1$. The value of b_3 is always $1/K$, where K is a normalization constant chosen so that the sum over all disk blocks of the selection probabilities is 1. The pairs (a_1, b_1), (a_2, b_2) define our selection strategy. The values we used in our experiments are: $a_1 = 0.4$, $b_1 = 0.4/K$, $a_2 = 0.7$, $b_2 = 0.6/K$.

Two strategies are possible to partition disk `D` in state signature blocks. We can have either a variable number of fixed size blocks or a fixed number of variable size blocks.

Reading a block from disk `D` can be done with a sequential transfer, whereas moving disk heads from one block to another requires a disk *seek* operation. Since seeks take longer than sequential transfers we decided to limit the number of seeks. This led us to use a fixed number of variable size blocks.

Let N be the number of disk blocks we want to use and let S be the number of state signatures in file `D`. Then each block (possibly with the exception of the last one that will be smaller) has $\lceil S/N \rceil$ state signatures. As a matter of fact, to avoid having too small blocks, we also impose a minimum value B for the number of state signatures in a block. Thus we may have less than N blocks if S is too small.

```
Calibrate(deleted_in_cloud, deleted_not_in_cloud)
{
deleted_states = deleted_in_cloud + deleted_not_in_cloud;
beta = deleted_not_in_cloud / deleted_states;

if (beta is close to 1)
 /* low disk cloud effectiveness: increase disk access rate */
 { /* increase disk_cloud_size by a given percentage */
   disk_cloud_size = (1 + speedup)*disk_cloud_size; }
else
  if (beta is close to 0)
  /* high disk cloud effectiveness: decrease disk access rate */
   { /* decrease disk_cloud_size by a given percentage */
     disk_cloud_size = (1 - slowdown)*disk_cloud_size; }}
```

Fig. 7. Function `Calibrate()`

In our experiments here we used $N = 100$ and $B = 10^4$. Thus, e.g. to have 100 disk blocks we need at least 10^6 reachable states.

3.7 Disk Cloud Size Calibration

Function `Calibrate()` (Fig. 7) is called by function `Checktable()` every time a calibration is needed for the disk cloud size. Two parameters are passed to function `Calibrate()`. Namely: the number of disk states deleted from `M` by `Checktable()` by only using disk blocks that are in the disk cloud (`deleted_in_cloud` in Fig. 7) and the number of disk states deleted from M by only using disk blocks that are not in the disk cloud (`deleted_not_in_cloud` in Fig. 7).

Function `Calibrate()` reads the *whole* file `D` and computes the ratio (`beta` in Fig. 7) between the number of deleted states not in the disk cloud and the number of total deleted states (`deleted_states` in Fig. 7). A value of `beta` close to 1 (low disk cloud effectiveness) means that the disk cloud has not been very effective in removing old states from table `M`. In this case, the variable `disk_cloud_size` (holding the disk cloud size) is increased by (`speedup`*`disk_cloud_size`). A value of `beta` close to 0 (high disk cloud effectiveness) means that the disk cloud has been very effective in removing old states from table `M`. In this case, we decrease the value of `disk_cloud_size` by (`slowdown`*`disk_cloud_size`) in order to lower the disk access rate.

In our experiments here we used `speedup` = 0.15 and `slowdown` = 0.15.

3.8 Calibration Frequency

Function `QueryCalibration()` called by function `Checktable()` (Fig. 4) tells us whether a calibration has to be performed or not. The rationale behind function `QueryCalibration()` is the following.

Calling function `Calibrate()` *too often* nullifies our efforts for reducing disk usage. In fact a calibration of the disk cloud size requires reading the *whole* file

D. However calling function Calibrate() *too sporadically* may have the same effect. In fact waiting too much for a calibration may lead to use an *oversized* disk cloud or an *undersized* one.

An oversized disk cloud increases disk usage beyond needs. Also an undersized disk cloud increases disk usage, since many old states will not be removed from M and we will be revisiting many already visited states.

In our current implementation function QueryCalibration() enables a calibration for every 10 calls of function Checktable() (Fig. 4). Our experimental results suggests that this is a reasonable *calibration frequency.*

4 Experimental Results

We implemented the LDBFS algorithm of Sect. 3 within the Murφ verifier. In the following we call *DMurφ* the version of the Murφ verifier we obtained.

In this section we report the experimental results we obtained by using DMurφ. Our experiments have two goals. First we want to know if by using locality there is indeed some gain w.r.t. the algorithm proposed in [16]. Second we want to measure DMurφ time overhead w.r.t. standard Murφ performing a RAM-BFS.

To meet our goals we proceed as follows. First, for each protocol in our benchmark we determine the minimum amount of memory needed to complete verification using the Murφ verifier (namely Murφ version 3.1 from [11]). Then we compare Murφ performances with those of DMurφ and with those of the disk based algorithm proposed in [16].

Our benchmark consists of some of the protocols in the Murφ distribution [11] and the kerb protocol from [19].

4.1 Results with Murφ

The Murφ verifier takes as input the amount of memory M to be used during the verification run as well as the fraction g (in $[0, 1]$) of M used for the queue (i.e. g is gPercentActive using a Murφ parlance).

We say that the pair (M, g) is *suitable* for protocol p iff the verification (with Murφ) of p can be completed with memory M and queue gM. For each protocol p we determine the least M s.t. for some g, (M, g) is suitable for p. In the sequel we denote by $M(p)$ such an M.

Of course $M(p)$ depends on the compression options used. Murφ offers *bit compression* (-b) and *hash compaction* (-c). Our approach (as the one in [16]) is compatible with all Murφ compression options. However, a disk based approach is really interesting only when, even using all compression options, one runs out of RAM. For this reason we only present results about experiments in which all compression options (i.e. -b -c) are enabled.

Fig. 8 gives some useful information about the protocols we considered in our experiments. The meaning of the columns in Fig. 8 is explained in Fig. 9.

					mu -b -c		
Protocol and Parameters	Bytes Diam	Reach	Rules	Max Q	M	g	T
`ns` 1,1,3,2,10	96 12	2,455,257	8,477,970	1,388,415	145,564,125	0.57	1,211.02
`n_peterson` 9	20 241	2,871,372	25,842,348	46,657	15,290,000	0.02	764.27
`newlist6` 7	32 91	3,619,556	21,612,905	140,382	22,590,004	0.04	1,641.67
`ldash` 1,4,1,false	144 72	8,939,558	112,808,653	509,751	118,101,934	0.06	12,352.93
`sci` 3,1,1,2,1	60 94	9,299,127	30,037,227	347,299	67,333,575	0.04	2,852.03
`mcslock1` 6	16 111	12,783,541	76,701,246	392,757	70,201,817	0.03	3,279.45
`sci` 3,1,1,5,1	64 95	75,081,011	254,261,319	2,927,550	562,768,255	0.04	35,904.86
`sci` 3,1,1,7,1	68 143	126,784,943	447,583,731	4,720,612	954,926,331	0.04	99,904.47
`kerb` NumIntruders=2	148 15	7,614,392	9,859,187	4,730,277	738,152,956	0.62	2,830.83
`newlist6` 8	40 110	81,271,421	563,937,480	2,875,471	521,375,945	0.03	31114.87

Fig. 8. Results on a INTEL Pentium III 866Mhz with 512M RAM. Murφ options used: `-b` (bit compression), `-c` (40 bit hash compaction), `-ndl` (no deadlock detection).

Attribute	**Meaning**
Protocol	Name of the protocol.
Parameters	Values of the parameters we used for the protocol. We show our parameter values in the same order in which such parameters appear in the `Const` section of the protocol file included in the Murφ distribution [11]. When such list is too long, as for the `kerb` protocol, we just list the assignments we modified in the `Const` section w.r.t. the distribution.
Bytes	Number of bytes needed to represent a state in the queue when bit compression is used. For protocol p we denote such number by StateBytes(p). Note that since we are using bit compression as well as hash compaction (`-b -c`), 5 bytes are used to represent (the signature of) a state in the hash table.
Reach	Number of reachable states for the protocol. For protocol p, we denote such number by $\|\text{Reach}(p)\|$.
Rules	Number of rules fired during state space exploration. For protocol p, we denote such number by RulesFired(p).
Max Q	Maximum queue size (i.e. number of states) attained during space state exploration. For protocol p we denote such number by MaxQ(p).
Diam	Diameter of the transition graph.
M	Minimum amount of memory (in kilobytes) needed to complete state space exploration. That is $M(p)$. Let b_h be the number of bytes taken by a state in the hash table (for us $b_h = 5$ since we are using hash compaction). From the Murφ source code [11] we can compute $M(p)$. We have: $M(p) = \|\text{Reach}(p)\| \, (b_h + (\text{MaxQ}(p)/\|\text{Reach}(p)\|)\text{StateBytes}(p))$.
g	Fraction of memory M used for the queue. From the Murφ source code [11] we can compute g. We have: $g = \text{MaxQ}(p)/\|\text{Reach}(p)\|$.
T	CPU time (in seconds) to complete state space exploration when using memory $\boldsymbol{M}$ and queue $\boldsymbol{gM}$. For protocol p, we denote such number by $T(p)$.

Fig. 9. Meaning of the columns in Fig. 8.

From column ***M*** of Fig. 8 we see that there are protocols requiring more than 512M bytes of RAM to complete. Thus we could not use standard Murφ on our 512M PC. However we were able to complete verification of such protocols using Cached Murφ (CMurφ) [3]. Giving to CMurφ enough RAM we get a very low *collision rate* and from [21] we know that in this case the CPU time taken by CMurφ is essentially the same as that taken by standard Murφ with enough RAM to complete the verification task. For this reason in the following we will regard the results in Fig. 8 as if they were all obtained by using standard Murφ with enough (i.e. $M(p)$) RAM to complete the verification task.

4.2 Results with DMurφ

Our next step is to run each protocol p in Fig. 8 with less and less (RAM) memory using our DMurφ. Namely, we run protocol p with memory limits $M(p)$, $0.5M(p)$ and $0.1M(p)$.

This approach allows us to easily compare the experimental results obtained from different protocols. The results we obtained are in Fig. 10. We give the meaning of rows and columns in Fig. 10.

Columns ***Protocol*** and ***Parameters*** have the meaning given in Fig. 9.

Column α (with $\alpha = 1, 0.5, 0.1$) gives information about the run of protocol p with memory $\alpha M(p)$.

Row ***States*** gives the ratio between the visited states (by DMurφ) when using memory $\alpha M(p)$ and $|\text{Reach}(p)|$ (in Fig. 8). This is the *state* overhead due to revisiting already visited states. This may happen since in function `Checktable()` (Fig. 4) we do not use the whole disk file `D` to remove old states from table `M`.

Row ***Rules*** gives the ratio between the rules fired (by DMurφ) when using memory $\alpha M(p)$ and RulesFired(p) (in Fig. 8). This is the *rule* overhead due to revisiting already visited states.

Row ***Time*** gives the ratio between the time $T_{DMur\varphi,\alpha}(p)$ (in seconds) to complete state space exploration (with DMurφ) when using memory $\alpha M(p)$ and $T(p)$ in Fig. 8. This is our time overhead w.r.t. RAM-BFS. Note that $T_{DMur\varphi,\alpha}(p)$ is the time elapsed between the start and the end of the state space exploration process. That is $T_{DMur\varphi,\alpha}(p)$ is not just the CPU time, instead $T_{DMur\varphi,\alpha}(p)$ also includes the time spent on disk accesses.

Note that for the *big* protocols in Fig. 8 (i.e. those requiring more than 512M of RAM) we could not run the experiments with $\alpha = 1$ on our machine with 512M of RAM. However, of course, the most interesting column for us is the one with $\alpha = 0.1$.

The experimental results in Fig. 10 show that even when $\alpha = 0.1$ our disk based approach is only between 1.4 and 5.3 (3 on average) times slower than a RAM-BFS with enough RAM to complete the verification task.

Protocol	Parameters	Mem	1	0.5	0.1
`n_peterson`	9	States	1.178	1.124	1.199
		Rules	1.178	1.124	1.199
		Time	2.148	2.056	2.783
`ns`	1,1,3,2,10	States	1.348	1.405	1.373
		Rules	1.487	2.011	1.645
		Time	1.734	2.144	1.953
`newlist6`	7	States	1.366	1.335	1.384
		Rules	1.365	1.334	1.382
		Time	1.703	1.765	2.791
`ldash`	1,4,1,false	States	1.566	1.668	1.702
		Rules	1.528	1.626	1.658
		Time	2.037	2.226	3.770
`sci`	3,1,1,2,1	States	1.260	1.189	1.183
		Rules	1.279	1.206	1.200
		Time	1.811	1.798	2.888
`mcslock1`	6	States	1.346	1.550	1.703
		Rules	1.346	1.550	1.703
		Time	1.915	2.477	5.259
`sci`	3,1,1,5,1	States	—	1.169	1.143
		Rules	—	1.195	1.167
		Time	—	1.828	2.553
`sci`	3,1,1,7,1	States	—	1.130	1.097
		Rules	—	1.152	1.115
		Time	—	1.421	1.743
`kerb`	`NumIntruders=2`	States		1.282	1.279
		Rules	—	1.060	1.080
		Time	—	1.234	1.438
`newlist6`	8	States	—	1.416	1.406
		Rules	—	1.412	1.405
		Time	—	2.612	4.436

Min		Time	1.703	1.234	1.438
Avg		Time	1.891	1.954	2.961
Max		Time	2.148	2.612	5.259

Fig. 10. Comparing DMurφ with RAM Murφ [11] (compression options: `-b -c`)

4.3 Results with Disk Based Murφ

To measure the time speed up we obtain by exploiting locality we are also interested in comparing our locality based disk algorithm DMurφ with the disk based Murφ presented in [16].

The algorithm in [16] is not available in the standard Murφ distribution [11]. However, if we omit the calibration (Fig. 7) step in function `Checktable()` (Fig. 4) and always use all disk blocks to clean up the unchecked queue `Q_unck` and

Protocol	Parameters	Mem	1	0.5	0.1
`n_peterson`	9	States	1.000	1.000	0.527
		Rules	1.000	1.000	0.507
		Time	2.623	2.430	> 90.704
`ns`	1,1,3,2,10	States	1.000	1.000	0.747
		Rules	1.000	1.000	0.309
		Time	1.259	242.131	>77.895
`newlist6`	7	States	1.000	1.000	0.253
		Rules	1.000	1.000	0.203
		Time	1.331	1.357	>42.817
`ldash`	1,4,1,false	States	0.355	—	—
		Rules	0.245	—	—
		Time	>50.660	—	—
`sci`	3,1,1,2,1	States	1.000	0.361	—
		Rules	1.000	0.647	—
		Time	1.616	> 11.863	—
`mcslock1`	6	States	1.000	1.000	0.137
		Rules	1.000	1.000	0.115
		Time	1.821	1.691	>11.605

Fig. 11. Comparing Disk Murφ in [16] with RAM Murφ [11] (compression options: `-b -c`)

table `M` (Fig. 1) we obtain exactly the algorithm in [16] (quite obviously since [16] was our starting point). Thus in the sequel for the algorithm in [16] we use the implementation obtained as described above.

For the algorithm in [16] (implemented as above) we wanted to repeat the same set of experiments we run for DMurφ. However the *big* protocols of Fig. 8 took too long. Thus we did not include them in our set of experiments.

Our results are in Fig. 11. Rows and columns in Fig. 11 have the same meaning as those in Fig. 10, but those of Fig. 11 refer to the algorithm in [16] (while those of Fig. 10 refer to DMurφ).

Computations taking too much longer than the time in Fig. 8 were aborted. In such cases we get a lower bound to the time overhead w.r.t. standard Murφ. This is indicated with a $>$ sign before the lower bound.

For aborted computations the rows ***States*** and ***Rules*** are, of course, less than 1 and give us an idea of the fraction of the state space explored before the computation was terminated.

Fig. 12 compares performances of our DMurφ with those of the disk based Murφ in [16]. The meaning of rows and columns of Fig. 12 is as follows.

Columns ***Protocol***, ***Parameters*** and column α (with $\alpha = 1, 0.5, 0.1$) have the meaning given in Fig. 9.

Row ***Time*** gives the ratio (or a lower bound to the ratio) between the verification time when using disk based Murφ in [16] and the verification time when using DMurφ.

Of course the interesting cases for us are those for which $\alpha = 0.1$ (i.e. there is not enough RAM to complete verification using a RAM-BFS). For such cases,

Protocol	Parameters	Mem	1	0.5	0.1
`n_peterson`	9	Time	1.221	1.182	> 32
`ns`	1,1,3,2,10	Time	0.726	112.934	> 39
`newlist6`	7	Time	0.781	0.768	> 15
`ldash`	1,4,1,false	Time	> 24	> 24	> 24
`sci`	3,1,1,2,1	Time	0.892	> 6	> 6
`mcslock1`	6	Time	0.950	0.683	> 2

Min		Time	0.726	0.683	> 2
Avg		Time	>4.762	> 24.261	> 19.667
Max		Time	> 24	112.934	> 39

Fig. 12. Comparing DMurφ with disk based Murφ in [16].

from the results in Fig. 12 we see that our algorithm is typically more than 10 times faster than the one presented in [16].

Note however that the results in Fig. 12 should be regarded more as qualitative results rather than quantitative results. In fact, as described above, we obtained the algorithm in [16] by eliminating the calibration step from our algorithm. It is quite conceivable that when calibration is not to be performed one can devise optimizations that are not possible when calibration has to be performed. Still, the message of Figs. 10, 11, 12 is quite clear: because of transition locality most of the time we do not need to read the whole disk `D`. This saves disk accesses and thus verification time.

Protocol	Parameters	Bytes	Reach	Rules	MaxQ
mcslock2	`N` $= 4$	16	945,950,806	3,783,803,224	30,091,568

Diam	T	Mem	HMem	QMem	TotMem
153	406,275	300	4,729,754	481,465	5,211,219

Fig. 13. Results for DMurφ on a 1GHz Pentium IV PC with 512M of RAM. Murφ options used: `-ndl` (no deadlock detection), `-b` (bit compression), `-c` (40 bit hash compaction).

4.4 A Large Protocol

We also wanted to test our disk based approach on a protocol out of reach for both standard Murφ [4,11] and Cached Murφ [21,3] on our 512M machine.

We found that the protocol `mcslock2` (with `N` $= 4$) in the Murφ distribution suites our needs. Our results are in Fig. 13. The meaning of the columns of Fig. 13 is as follows.

Columns ***Protocol***, ***Parameters***, ***Bytes***, ***Reach***, ***Rules***, ***MaxQ***, ***Diam***, ***T*** have the same meaning as in Fig. 8 but they refer to DMurφ (while those of Fig. 8 refer to standard Murφ).

Column ***Mem*** gives the total RAM memory (in Megabytes) given to DMurφ to carry out the given verification task.

Column ***HMem*** gives the hash table size (in kilobytes) that would be needed if we were to store all reachable states in a RAM hash table.

Column ***QMem*** gives the RAM size (in kilobytes) needed for the BFS queue if we were to keep all BFS queue in RAM.

Column ***TotMem*** gives the RAM size (in kilobytes) needed to complete the verification task using a RAM-BFS with standard Murφ. ***TotMem*** is equal to (***HMem*** + ***QMem***).

5 Conclusions

We presented a disk based Breadth First *Explicit State Space Exploration* algorithm as well as an implementation of it within the Murφ verifier. Our algorithm has been obtained from the one in [16] by exploiting *transition locality* [21] to decrease disk usage (namely, disk read accesses).

Our experimental results show the following. Our algorithm is typically more than 10 times faster than the disk based algorithm proposed [16]. Moreover, even when using 1/10 of the RAM needed to complete verification, our algorithm is only between 1.4 and 5.3 times (3 times on average) slower than RAM-BFS (namely, standard Murφ) with enough RAM memory to complete the verification task at hand.

Statistical properties of transition graphs (as transition locality is) have proven quite effective in improving state space exploration algorithms ([21,22]) on a single processor machine. Looking for new statistical properties and for ways to exploit such statistical properties when performing verification on distributed processors are natural further developments for our research work.

Acknowledgements. We are grateful to Igor Melatti and to FMCAD referees for helpful comments and suggestions on a preliminary version of this paper.

References

[1] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, C-35(8), Aug 1986.

[2] 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), 1992.

[3] url: `http://univaq.it/~tronci/cached.murphi.html`.

[4] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–5, 1992.

[5] R. Sisto F. Lerda. Disributed-memory model checking with spin. In *Proc. of 5th International SPIN Workshop*, volume 1680. LNCS, Springer, 2000.

[6] G. J. Holzmann. The spin model checker. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.

[7] G. J. Holzmann. An analysis of bitstate hashing. *Formal Methods in Systems Design*, 1998.

[8] A. J. Hu, G. York, and D. L. Dill. New techniques for efficient verification with implicitily conjoined bdds. In *31st IEEE Design Automation Conference*, pages 276–282, 1994.

[9] C. N. Ip and D. L. Dill. Better verification through symmetry. In *11th International Conference on: Computer Hardware Description Languages and their Applications*, pages 97–111, 1993.

[10] C. N. Ip and D. L. Dill. Efficient verification of symmetric concurrent systems. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 230–234, 1993.

[11] url: `http://sprout.stanford.edu/dill/murphi.html`.

[12] R. K. Ranjan, J. V. Sanghavi, R. K. Brayton, and A. Sangiovanni-Vincentelli. Binary decision diagrams on network of workstations. In *IEEE International Conference on Computer Design*, pages 358–364, 1996.

[13] 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.

[14] url: `http://netlib.bell-labs.com/netlib/spin/whatispin.html`.

[15] 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.

[16] 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.

[17] 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.

[18] U. Stern and D. L. Dill. A new scheme for memory-efficient probabilistic verification. In *IFIP TC6/WG6.1 Joint International Conference on: Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification*, 1996.

[19] url: `http://verify.stanford.edu/uli/research.html`.

[20] T. Stornetta and F. Brewer. Implementation of an efficient parallel bdd package. In *33rd IEEE Design Automation Conference*, pages 641–644, 1996.

[21] 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.

[22] E. Tronci, G. Della Penna, B. Intrigila, and M. Venturini Zilli. A probabilistic approach to space-time trading in automatic verification of concurrent system. In *Proc. of 8th IEEE Asia-Pacific Software Engineering Conference (APSEC)*, Macau SAR, China, Dec 2001. IEEE Computer Society Press.

[23] Pierre Wolper and Dennis Leroy. Reliable hashing without collision detection. In *Proc. 5th Int. Conference on: Computer Aided Verification*, pages 59–70, Elounda, Greece, 1993.