Exploiting Transition Locality in Automatic Verification[*]

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

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

[2] Dip. di Scienze dell'Informazione, Università di Roma "La Sapienza", Via Salaria 113, 00198 Roma, Italy
zilli@dsi.uniroma1.it

Abstract. In this paper we present an algorithm to contrast *state explosion* when using *Explicit State Space Exploration* to verify protocols. We show experimentally that protocols exhibit *transition locality.*
We present a verification algorithm that exploits transition locality as well as an implementation of it within the Murφ verifier.
Our algorithm is compatible with all Breadth First (BF) optimization techniques present in the Murφ verifier and it is by no means a substitute for any of them. In fact, since our algorithm trades space with time, it is typically most useful when one runs out of memory and has already used all other state reduction techniques present in the Murφ verifier.
Our experimental results show that using our approach we can typically save more than 40% of RAM with an average time penalty of about 50% when using (Murφ) bit compression and 100% when using bit compression and hash compaction.

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 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 verification, *Explicit* State Space Exploration often outperforms *Symbolic* (i.e. OBDD based, [1,2]) State Space Exploration [6]. 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 [4, 15] and Murφ [3,10].

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 [22,7,8,18,19,5].

[*] This research has been partially supported by MURST project TOSCA

[**] Corresponding Author: Enrico Tronci. Tel: +39 0862 433129. Fax: +39 0862 433180.

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 [16,17,13,21,14].

In this paper we study the possibility of exploiting statistical properties of protocol transition graphs to improve state exploration algorithms. This is quite similar to what is usually done when optimizing a CPU on the basis of program profiling [12].

Our algorithm allows us to reduce the RAM needed to complete state space exploration and exploits auxiliary storage as well. We pay for this memory saving with a longer time to complete state space exploration. Our results can be summarized as follows.

- We show experimentally (Sect. 2) that protocols exhibit *transition locality.* That is transitions tend to be local w.r.t. levels of a Breadth First (BF) visit. We support our claim by measuring transition locality for the set of protocols included in the Murφ verifier distribution. To the best of our knowledge this is the first time that such *profiling* of transition graphs is presented.
- We present a verification algorithm that exploits transition locality as well as an implementation of it within the Murφ verifier (Sect. 3). Essentially our algorithm replaces the hash table used in a BF state space exploration with a cache memory (i.e. no collision detection is done) and uses auxiliary (disk) storage for the BF queue.
  Using a fixed size cache memory we do not incur state explosion. However we may incur nontermination when our cache memory is *too small.* In fact in this case we may visit over and over the same set of states. Note however that upon termination, we are guaranteed that all reachable states have been visited.
  To the best of our knowledge this is the first time that a cache based state space exploration algorithm is presented.
  Note in particular that the approach in [22] is a state compression technique and "no collision detection" there refers to state signatures. That is, a (signature) collision in [22] may lead to declare as visited a nonvisited state. On the other hand, we simply forget a visited state upon a collision, thus declaring as nonvisited a visited state.
  Note that the SPIN verifier can use disk storage for the Depth First (DF) stack. However states are still stored in a hash table which is where state explosion typically occurs.
- Our algorithm is compatible with all well known state compression techniques (e.g. as those in [19,5]). In particular it is compatible with all state reduction techniques present in the Murφ verifier. We show experiments using our algorithm together with the bit compression [10] and hash compaction [18,19] features of the Murφ verifier (Sect. 4).
- Our experimental results (Sect. 4) show that we can verify systems more than 40% larger than those that can be handled using a hash table based approach. Our time penalty is about 50% when using (Murφ) bit compression and 100% when using bit compression and hash compaction.

2 Transition Locality for Finite State Systems

In this section we define our notion of locality for transitions and show experimentally that for protocols most transitions are local. We do this by showing that for all our benchmark protocols most transitions are indeed local.

We used as benchmark protocols all those available in the Murφ verifier distribution [10] plus the Kerberos protocol from [20]. This gives us a fairly representative benchmark set.

For our purposes, a protocol is represented as a *Finite State System.*

Definition 1. *1. 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 $\mathcal{S}$.*

2. 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. $R(s, s')$.

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

In the following we will always refer to a given system $\mathcal{S} = (S,\ I,\ A,\ R)$. Thus, e.g., we will write Reach for $\text{Reach}(\mathcal{S})$. Also we may speak about the set of initial states I as well as about the transition relation R without explicitly mentioning $\mathcal{S}$.

The core of all automatic verification tools is the *reachability analysis*, i.e. the computation of Reach given a definition of $\mathcal{S}$ in some language.

Since the transition relation R of a system defines a graph (*transition graph*) computing Reach means visiting (exploring) the transition graph starting from the initial states in I. This can be done, e.g., using a *Depth First* (DF) visit or a *Breadth First* (BF) visit.

For example the automatic verifier SPIN [15] uses a DF visit. Murφ [10] may use DF as well as BF, although certain compression options can only be used with a BF visit.

In the following we will focus on BF visit. As well known a BF visit 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.

Definition 2. *Formally we define the set of states* at level k *(notation $L(k)$) as follows. $L(0) = I$, $L(k+1) = \{s' \mid s \in L(k)$ and $R(s, s')$ and $s' \notin \cup_{i=0}^{i=k} L(i)\}$.*

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

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

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

Definition 3. *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* $|\text{level}(s') - \text{level}(s)| \leq k$.

*Transition $R(s, a, s')$ is said to be a k-*transition *iff* $\text{level}(s') - \text{level}(s) = k$. *Note that for k-transitions, $k \leq 1$ and can be negative.*

A 1-transition from state s is a *forward transition*, i.e. a transition leading to a *new state* (a state not in Visited(level(s))). A k-transition with $k < 0$ is a *backward transition*, i.e. a transition leading to a visited state. A 0-transition from state s just leads to a state s' in the same level of s.

We are interested in the distribution of k-transitions in the transition graph. This motivates the following definition.

Definition 4.
1. *We denote with $N(s, k)$ the number of k-transitions from s and with $N(s)$ the number of transitions from s.*
2. *We define: $\delta(s, k) = N(s, k)/N(s)$. That is $\delta(s, k)$ is the fraction of transitions from s that are k-transitions, i.e. the probability of getting a k-transition when picking at random a transition from s. Of course if s is at level λ we have: $\sum_{k=-\lambda}^{k=1} \delta(s, k) = 1$.*
3. *If we consider the experiment consisting of picking at random a state s in* Reach *and returning $\delta(s, k)$ then we get a random variable that we denote with $\mathbf{\Delta}(k)$. The expected value $E\{\mathbf{\Delta}(k)\}$ of $\mathbf{\Delta}(k)$ is the average value of $\delta(s, k)$ on all reachable states. That is:*
$E\{\mathbf{\Delta}(k)\} = \frac{1}{|\text{Reach}|} \sum_{s \in \text{Reach}} \delta(s, k)$.
4. *As usual, we denote with $\sigma^2(k)$ the variance of $\mathbf{\Delta}(k)$ and with $\sigma(k)$ its standard deviation [11].*

We show experimentally that transitions in protocols tend to be *local* w.r.t. levels. We do this by showing that for the set of protocols shown in Fig. 1 most transitions are 1-local. That is, for most transitions $R(s, a, s')$ s' will be either in the same level as s (0-transition) or in the next level (1-transition) or in the previous level (-1-transition).

We want to setup experiments to measure what is the percentage of 1-local transitions in the transition graph of a given protocol.

A 1-local transition can only be a k-transition with $k = -1, 0, 1$. The expected fraction of k-transitions is $E\{\mathbf{\Delta}(k)\}$. Thus the expected fraction of 1-local transitions is $SumAvg = E\{\mathbf{\Delta}(-1)\} + E\{\mathbf{\Delta}(0)\} + E\{\mathbf{\Delta}(1)\}$.

When *SumAvg* is close to 1 almost all transitions in the protocol are 1-local. When *SumAvg* is close to 0 there are almost no 1-local transitions in the protocol.

Although not strictly needed for our present purposes we are also interested in knowing how 1-local transitions are distributed in the graph. Namely, we want to know whether 1-local transitions are concentrated only in some part of the transition graph or rather they are more or less uniformly distributed in the transition graph.

Protocol	$E\{\delta(-1)\}$	$\sigma(-1)$	$E\{\delta(0)\}$	$\sigma(0)$	$E\{\delta(1)\}$	$\sigma(1)$	Sum Avg
`n_peterson.m`	0	0	0	0	0.958174	0.0877715	0.958174
`adash.m`	0.0381775	0.122	0.00558149	0.0436066	0.723393	0.292406	0.76715199
`adashbug.m`*	0.0376604	0.124403	0.00228899	0.0295028	0.793586	0.270018	0.83353539
`eadash.m`	0.050598	0.0960145	0.015647	0.0562551	0.765236	0.178076	0.831481
`ldash.m`	0.0107259	0.0719168	0.0763593	0.138464	0.624139	0.191624	0.7112242
`arbiter.m`*	0.00848939	0.057543	0.0107366	0.0537135	0.92784	0.168859	0.94706599
`cache3.m`	0.0401213	0.178884	0.00476603	0.0558438	0.565482	0.381654	0.61036933
`cache3multi.m`	0.0389835	0.102299	0.00299148	0.0263492	0.831139	0.176004	0.87311398
`newcache3.m`	0.0360339	0.0991869	0.00912116	0.0533416	0.736229	0.227796	0.78138406
`sym.cache3.m`	0.041675	0.10478	0.00757049	0.0445772	0.876884	0.17786	0.92612949
`down.m`*	0	0	0.0776385	0.142608	0.922361	0.142608	0.9999995
`kerb.m`	0	0	0.18417	0.385163	0.441651	0.494666	0.625821
`list6.m`	0.0183668	0.0710808	0.0246248	0.0815233	0.844018	0.189053	0.8870096
`list6too.m`	0	0	0	0	0.988378	0.0621402	0.988378
`newlist6.m`	1.53327e-05	0.00175109	0	0	0.999586	0.00908985	0.9996
`mcslock1.m`	0.0128856	0.0675736	0	0	0.881379	0.162002	0.8942646
`mcslock2.m`	0.0054652	0.0458147	0.00056426	0.0151108	0.921489	0.156949	0.92751846
`ns-old.m`	0.546833	0.497802	0	0	0.101695	0.302247	0.648528
`ns.m`	0.585714	0.492598	0	0	0.0969388	0.295874	0.6826528
`sci.m`	0.215646	0.238654	0.0108192	0.0617188	0.642466	0.265885	0.8689312

Fig. 1. Transition Distribution Table

To some extent this can be done by computing the standard deviation $\sigma(k)$ of $\mathbf{\Delta}(k)$. If $\sigma(k)$ is small compared to $E\{\mathbf{\Delta}(k)\}$ then for most states $\delta(s,k)$ is *close* to $E\{\mathbf{\Delta}(k)\}$.

The above considerations led us to perform the experiments whose results are shown in Fig. 1. For each protocol in Fig. 1 for $k = -1, 0, 1$, in Fig. 1 we show $E\{\mathbf{\Delta}(k)\}$, $\sigma(k)$ and $SumAvg = E\{\mathbf{\Delta}(-1)\} + E\{\mathbf{\Delta}(0)\} + E\{\mathbf{\Delta}(1)\}$. Our findings can be summarized as follows.

Experimental Fact 1. *For all protocols listed in Fig. 1, we have that for most states, more than 60% of the transitions are 1-local. Indeed, for most of the protocols in Fig. 1, we have that for most states more that 75% of the transitions are 1-local.*

In fact, from Fig. 1 we have that for all protocols $SumAvg > 0.6$ and in most cases $SumAvg > 0.75$. This shows that most transitions are 1-local.

Moreover, for many protocols, standard deviations $\sigma(-1)$, $\sigma(0)$, $\sigma(1)$ are *relatively small* compared to $SumAvg$. In such cases, *for most states* the fraction of 1-local transitions is close to $SumAvg$.

Since for most states most transitions are 1-local we have that locality holds uniformly. Hence if we pick at random a state $s \in \text{Reach}$ in most cases most transitions from s (say about 75%) will be 1-local.

Remark 1. One may wonder why protocols exhibit locality. We conjecture that this is structural for *human made* systems and, indeed, is a consequence of the techniques used (consciously or unconsciously) to master complexity in the design task.

Remark 2. A very interesting result would be to prove (or disprove) that almost all FSSs exhibit locality. Note however that, a priori, truthness or falsehood of such result would not imply anything on protocols in particular.

E.g. for OBDDs we have already a similar situation (in *reverse mode*). In fact in [9] it is proved that for almost any boolean function f of n variables the smaller OBDD for f has size exponential in n. Nevertheless, it is experimentally known that most of the boolean functions implemented by digital circuits have OBDDs of size polynomial (often about linear) in the number of variables.

We measured locality using the Murφ verifier [10]. This gives us a wide benchmark set for free since many protocols have already been defined using the Murφ input language.

The results in Fig. 1 have been obtained by modifying the Murφ verifier so as to compute $E\{\mathbf{\Delta}(k)\}$ and $\sigma(k)$ while carrying out state space exploration.

All the experiments in Fig. 1 were performed using bit compression (Murφ option `-b`) [10] and disabling deadlock detection (option `-ndl`).

Deadlock detection has been disabled since some of the protocols (e.g. Kerberos `kerb.m`) are deadlocked. Murφ stops the state exploration as soon as a deadlock or a bug is found. Since our main interest here is in gathering information about the structure of the transition graph we disabled deadlock detection. This allows us to explore the whole reachable part of the transition graph also for protocols containing deadlocks.

The three protocols with superscript * in Fig. 1 contain bugs. In such cases, state space exploration stops as soon as a bug is found. We included such *buggy* protocols just for completeness.

The data in Fig. 1 depend neither on the used machine nor on the memory available as long as the visit is completed. Of course we would have obtained the same results without bit compression.

3 Cache Based Breadth First State Space Exploration

In this section we give an algorithm that takes advantage of the transition locality (Sect. 2) of protocols.

We modify the standard (hash table based) BF visit as follows. First, we use a *cache memory* rather than a hash table. This means that we do not perform any collision check. Thus, when a state s' is hashed to an entry already holding a state s, we replace s with s' so *forgetting* about s.

Using a fixed size cache rather than a hash table to store visited states is appealing since the cache size does not grow with the state space size. However this approach faces two obvious obstructions.

Queue Explosion. The queue size may get very large. In fact, since we forget visited states, at each level in our cache based BF visit the states *apparently* new tend to be much more than those of a true (hash based) BF visit. This may quickly fill up our queue.

Nontermination. Our state space exploration may not terminate. In fact, because of collisions, we forget visited states. Thus, we may visit again and again the same set of states.

When using a fixed size cache memory to hold visited states no state explosion occurs in the cache. However state explosion may occur in the queue now. For this reason, unlike previous work on usage of auxiliary storage in state space exploration (e.g. in [17]), we use disk storage to hold the queue rather than the hash table. This is good news since keeping a queue on disk is much easier than keeping a hash table on disk.

Nontermination is thus the real obstruction. Here is where *locality* helps us. Since most transitions are local one can reasonably expect that, to avoid looping, it is enough to remember the *recently* visited states rather than *all* visited states. This is exactly what a *cache memory* is for.

Note that a cache memory implements the above discipline only in a *statistical sense*. That is, replaced (i.e. *forgot*) states in the cache are *often* old states, but not always. Fortunately, this is enough for us.

Of course, if the cache memory is *too small* w.r.t. the size of the state space locality cannot help us.

Note that whenever our cache based BF visit terminates it gives the correct answer. That is *all* reachable states have been visited. When we stop our cache based visit to prevent looping we may or may not have visited all reachable states.

In the following we sketch our algorithm and show how we integrated it into the Murφ verifier.

To take advantage of transition locality we modify the standard BF visit as shown in Fig. 2.

We use a cache memory rather than a hash table. This may lead to nontermination since we may visit over and over the same set of states. For this reason we *guarded* the while loop with a check on the collision rate (i.e. <number of collisions in cache>/<number of insertions in cache>).

Since we may forget visited states, our queue tends to contain many more states than the one used in a standard BF visit. For this reason we decided to implement our queue using auxiliary memory. In the present implementation we use disk storage as auxiliary memory. Note however that (the RAM of) another workstation could have been used as well.

The implementation schema for the cache memory is quite standard. In the following we describe our implementation schema for the queue on disk (Fig. 3).

We split the queue `Q` into two segments: *head queue* and *tail queue*. Both head queue and tail queue are in RAM and each of them can hold `ram_queue_size` states. States are *enqueued* in the tail queue and *dequeued* from the head queue.

When the tail queue is full, its content is flushed on disk by appending it to `swapout_file`. Thus, since disks are quite large, our disk queue approximates a potentially infinite queue. It overflows only when our disk is full.

When the head queue is empty, it gets reloaded with states from `swapin_file`. If `swapin_file` is empty we swap `swapin_file` and `swapout_file` and try to load states from `swapin_file` again. If `swapin_file` is still empty then we swap the head queue with the tail queue.

```
Queue Q;      Cache T;       collision_rate = 0.0;
bfs(init_states, next) {
for s in init_states enqueue(Q, s);  /* load Q with initial states */
for s in init_states insert(T, s);  /* mark init states as visited */
while ((Q is not empty) and (collision_rate <= 0.9)) {s = dequeue(Q);
      for all s' in next(s) if (s' is not in T) {insert(T, s'); enqueue(Q, s'); }}}
```

Fig. 2. Cache based Breadth First Visit

```
void enqueue(state s) {if (tail queue is full) swap_out();
/* tail queue is not full */ insert s into tail queue; tail_queue_elements++;}

/* hyp: dequeue() always called on non empty queue */
state dequeue() {if (head queue is empty) swap_in();
/* head queue is not empty now */ head_queue_elements--; return top of head queue;}

void swap_out() { /* flush tail queue on disk and reset counter */
append tail queue to swapout_file; tail_queue_elements = 0;  }

void swap_in() { /* load head queue from disk  */
if (swapin_file is not empty)
    load head queue with at most ram_queue_size states from swapin_file;
else /* swapin_file is empty, we use swapout_file  */
   { swap the swapin_file and swapout_file;
     if (swapin_file is not empty)
        load head queue with at most ram_queue_size states from swapin_file;
     else /* also swapout_file is empty, we use tail_queue  */
       { swap the head queue and tail queue;
         if (head queue is empty)
            /* underflow error */ { error(``queue is empty''); }}}
head_queue_elements = number of states in head queue; }
```

Fig. 3. Disk Queue Functions

Using pointers, swapping (for files and for queue segments) is immediate (just a few assignments).

Rather than building a tool from scratch we decided to integrate our algorithm into an already existing tool. This gives us at least two advantages. First, we have immediately available many benchmark systems for testing. Second, we can exploit other memory reduction techniques that have already been implemented. We decided to integrate our algorithm within the Murφ verifier [10].

To be consistent with the *standard* (i.e. hash table based) Murφ verifier the value of `ram_queue_size` (Fig. 3) is such that the head queue and the tail queue together take the same amount of memory as the queue in standard Murφ. Namely, if M is the available memory for verification then `gPercentActive`$*M$ is the amount of RAM memory used for the queue in standard Murφ as well as in our cache based Murφ.

Integrating our algorithm into the Murφ verifier requires some care and consideration. In fact, to save RAM, Murφ (as well as SPIN for that matter) stores pointers to states rather than states in the queue.

As far as we are concerned, Murφ BF visit can have two behaviours. One when hash compaction is not used and another one when it is used. All other Murφ options have no impact on the hash table or the queue and thus are *transparent* to us.

When no hash compaction is used Murφ stores states in the hash table and pointers to hash table slots (states) in the queue. When hash compaction is used, Murφ stores state signatures in the hash table and pointers to states in the queue.

Let us consider the case without hash compaction first. Using a cache rather than a hash table leads to the following problem. When we overwrite a cache slot (collision) as a side effect we may also change the content of the queue. In fact that slot may be pointed to by the queue. For this reason we modified Murφ queue so that it holds states and not just pointers to states. Of course all Murφ functions depending on this fact have to be changed accordingly. This also fixes the situation for the case in which hash compaction is used.

Of course storing states rather than pointers (to states) in the queue takes more space. However, for the reasons explained above, we are going to use disk storage to implement our queue. Thus having a large queue does not pose any serious problem in our context.

With our approach all optimization strategies implemented within the Murφ verifier (bit compression, hash compaction, symmetry reduction, etc) are also available to us.

Note that the bound on the omission probability for the hash compaction computed "on-line" by Murφ accordingly to [19] is also (a fortiori) valid in our case. In fact at each (BF) level we visit a superset of the states visited by a standard BF visit.

4 Experimental Results

We report the experimental results we obtained using Murφ (modified as described in Sect. 3).

We want to measure how much (RAM) memory we can save by using our cache based approach. To make the results from different protocols comparable we proceed as follows.

First, for each protocol we determine the minimum amount of memory needed to complete verification using the Murφ verifier (namely Murφ version 3.1 from [10]).

Let M be the amount of memory and g (in $[0, 1]$) be the fraction 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 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 following we denote with $M(p)$ such M.

Of course $M(p)$ depends on the compression options one uses. Murφ offers *bit compression* (`-b`) and *hash compaction* (`-c`). We determined $M(p)$ when only bit compression is used (`-b`) and when bit compression and hash compaction are used (`-b -c`). The results are in Fig. 4.

The meaning of the columns in Fig. 4 is as follows. Column ***Bytes*** gives the number of bytes needed to represent the state when bit compression is used. With hash compaction 40 bits are used to represent the hash-compressed state. Column ***Reach*** gives the number of reachable states for the protocol. For pro-

						`mu -b`			`mu -b -c`		
Protocol	Bytes	Reach	Rules	Max Q	Diam	M	g	T	M	g	T
`n_peterson.m`	16	163298	1143086	3775	145	3204	0.0233	269.13	813	0.0233	273.32
`adash.m`	144	10466	137708	734	37	1516	0.075	60.57	55	0.075	62.98
`adashbug.m`	144	3742	39619	646	14	544	0.2	18.19	21	0.18125	17.60
`eadash.m`	376	133491	1786047	9050	47	49571	0.06875	4102.33	688	0.06875	4114.71
`ldash.m`	144	254986	2647358	14988	64	36930	0.075	4002.97	1307	0.062	3950.50
`arbiter.m`	8	1103	2365	301	12	15	0.4	0.1	7	0.3	0.1
`cache3.m`	12	577	2440	102	16	10	0.4	0.15	4	0.4	0.15
`cache3multi.m`	28	13738	65357	1229	29	435	0.1	34.16	73	0.1	35.11
`newcache3.m`	52	4357	20201	462	27	240	0.1125	8.01	24	0.15	7.93
`sym.cache3.m`	28	31433	264758	2877	32	994	0.1039	36.22	167	0.10625	36.03
`down.m`	4	10957	52315	1313	19	92	0.15	1.35	60	0.15	1.22
`kerb.m`	80	109282	172111	20523	19	9046	0.191	291.49	615	0.190625	301.86
`list6.m`	24	23410	99874	1095	53	645	0.05	13.73	119	0.05	14.90
`list6too.m`	20	1077	11622	64	36	26	0.2	15.55	6	0.1	16.83
`newlist6.m`	24	13044	53595	631	53	360	0.05	17.23	67	0.05	18.34
`mcslock1.m`	12	23644	94576	928	69	373	0.04023	16.17	120	0.04375	16.76
`mcslock2.m`	12	540219	1620657	15655	111	8503	0.0293	234.68	2693	0.02969	237.48
`ns-old.m`	24	1121	2578	424	11	33	0.4	0.69	8	0.4	0.69
`ns.m`	24	980	2314	382	11	29	0.4	0.59	7	0.4	0.62
`sci.m`	56	18193	60455	1175	62	1071	0.066	27.29	94	0.06875	28.17

Fig. 4. Results on a SUN Sparc machine with 512M RAM and SunOS 5.8. Murφ options: `-b` (bit compression), `-c` (40 bit hash compaction). All experiments have been carried out with option `-ndl` (no deadlock detection). Column M gives memory used (in kilobytes) and column T gives CPU time (in seconds).

tocol p, in the following we denote such number with $|\text{Reach}(p)|$. Column ***Rules*** gives the number of rules fired during state space exploration. This number only depends on the transition graph. For protocol p, in the following we denote such number with RulesFired(p). Column ***Max Q*** gives the maximum queue size in a BF visit. This number only depends on the transition graph. Column ***Diam*** gives the diameter of the transition graph. This number only depends on the transition graph. Column ***M*** gives the minimum amount of memory (in kilobytes) needed to complete state space exploration. Two cases: `-b` and `-b -c`. Column ***g*** gives the fraction of memory M used for the queue. Two cases: `-b` and `-b -c`. Column ***T*** gives the time (in seconds) to complete state space exploration when using memory M and queue gM. Two cases: `-b` and `-b -c`. For protocol p, in the following we denote such numbers with $T_b(p)$ and $T_{bc}(p)$ respectively.

Our next step is to run each protocol p with less and less memory using our cache based Murφ. That is we run p with memory limits $M(p)$, $0.9M(p)$, $\ldots 0.1M(p)$.

This approach allows us to easily compare the experimental results obtained from different protocols. Note that, as in [6], our RAM is not filled up. Thus OS buffers may reduce the time to access the queue. As a result our measured time may be slightly smaller than those obtained with a filled up memory.

The results using option `-b` (bit compression) are in Fig. 5. The results obtained using options `-b -c` (bit compression and hash compaction) are in Fig. 6.

We only considered protocols requiring at least 10 kilobytes of RAM (column M of Fig. 4) to complete state space exploration. In the experiments in Figs. 5, 6 the value of g (`gPercentActive`) is chosen as in Fig. 4.

	Mem	1	0.9	0.8	0.7	0.6	0.5	0.4	0.3	0.2	0.1
n_peterson	states	1.004	1.011	1.028	1.242	4.366*	5.028*	4.036*	3.037$^\infty$	2.039$^\infty$	1.035$^\infty$
	rules	1.004	1.011	1.028	1.242	4.364*	4.994*	3.997*	2.992$^\infty$	1.992$^\infty$	0.993$^\infty$
	time	0.972	0.981	1.004	1.201	4.222*	4.888*	3.914*	2.959$^\infty$	1.975$^\infty$	1.017$^\infty$
	coll	0.062	0.128	0.225	0.437	0.863*	0.901*	0.901*	0.901$^\infty$	0.902$^\infty$	0.904$^\infty$
adash	states	1.006	1.016	1.043	1.124	1.738	5.160*	4.300$^\infty$	3.631$^\infty$	2.389$^\infty$	1.147$^\infty$
	rules	1.007	1.016	1.043	1.125	1.736	4.749*	3.714$^\infty$	2.937$^\infty$	1.819$^\infty$	0.782$^\infty$
	time	1.001	1.010	1.039	1.124	1.747	4.824*	3.792$^\infty$	3.013$^\infty$	1.889$^\infty$	0.818$^\infty$
	coll	0.063	0.132	0.236	0.377	0.655	0.904*	0.907$^\infty$	0.919$^\infty$	0.916$^\infty$	0.918$^\infty$
adashbug	states	1.002	1.006	1.010	1.032	1.055	1.124	1.261	1.477	1.871*	1.336$^\infty$
	rules	1.001	1.001	1.003	1.011	1.021	1.061	1.145	1.275	1.504*	0.965$^\infty$
	time	0.983	0.980	0.982	0.988	0.999	1.045	1.135	1.280	1.540*	1.019$^\infty$
	coll	0.052	0.113	0.199	0.312	0.422	0.545	0.672	0.787	0.904*	0.931$^\infty$
eadash	states	1.007	1.021	1.066	1.382	6.540*	5.251$^\infty$	4.862$^\infty$	3.805$^\infty$	2.292$^\infty$	1.034$^\infty$
	rules	1.007	1.021	1.065	1.384	6.008*	4.440$^\infty$	3.910$^\infty$	2.918$^\infty$	1.649$^\infty$	0.667$^\infty$
	time	1.029	1.043	1.094	1.439	6.354*	4.653$^\infty$	4.133$^\infty$	3.155$^\infty$	1.831$^\infty$	0.775$^\infty$
	coll	0.063	0.135	0.251	0.493	0.908*	0.905$^\infty$	0.918$^\infty$	0.921$^\infty$	0.913$^\infty$	0.904$^\infty$
ldash	states	1.019	1.058	1.295	7.495*	6.114$^\infty$	6.106$^\infty$	4.067$^\infty$	3.753$^\infty$	2.361$^\infty$	1.020$^\infty$
	rules	1.018	1.056	1.287	6.698*	5.084$^\infty$	4.847$^\infty$	3.086$^\infty$	2.726$^\infty$	1.633$^\infty$	0.664$^\infty$
	time	1.043	1.078	1.324	6.758*	5.112$^\infty$	4.984$^\infty$	3.177$^\infty$	2.828$^\infty$	1.703$^\infty$	0.701$^\infty$
	coll	0.069	0.161	0.382	0.907*	0.902$^\infty$	0.918$^\infty$	0.902$^\infty$	0.920$^\infty$	0.915$^\infty$	0.902$^\infty$
arbiter	states	1.000	1.004	1.007	1.021	1.032	1.060	1.067	1.137	1.230	0.907$^\infty$
	rules	1.000	1.000	1.001	1.000	1.010	1.017	1.024	1.078	1.139	0.800$^\infty$
	time	1.000	1.100	1.100	1.200	1.200	1.300	1.300	1.400	1.500	1.200$^\infty$
	coll	0.054	0.133	0.189	0.327	0.405	0.537	0.611	0.755	0.833	0.931$^\infty$
cache3	states	1.002	1.021	1.047	1.192	1.459	3.000	3.466*	1.733$^\infty$	1.733$^\infty$	1.733$^\infty$
	rules	1.001	1.020	1.045	1.180	1.416	2.953	3.064*	1.383$^\infty$	1.291$^\infty$	0.776$^\infty$
	time	1.200	1.200	1.200	1.467	1.867	4.133	3.467*	1.667$^\infty$	1.667$^\infty$	0.800$^\infty$
	coll	0.050	0.127	0.230	0.406	0.586	0.831	0.901*	0.905$^\infty$	0.918$^\infty$	0.906$^\infty$
cache3multi	states	1.009	1.030	1.098	1.291	2.422	5.387*	4.367$^\infty$	3.203$^\infty$	2.184$^\infty$	0.946$^\infty$
	rules	1.011	1.037	1.117	1.328	2.535	5.173*	3.902$^\infty$	2.674$^\infty$	1.725$^\infty$	0.712$^\infty$
	time	1.010	1.037	1.118	1.337	2.564	5.255*	3.974$^\infty$	2.730$^\infty$	1.759$^\infty$	0.728$^\infty$
	coll	0.062	0.141	0.272	0.458	0.752	0.908*	0.909$^\infty$	0.908$^\infty$	0.911$^\infty$	0.900$^\infty$
newcache3	states	1.012	1.029	1.053	1.213	1.895	5.279*	4.131$^\infty$	2.984$^\infty$	2.066$^\infty$	1.148$^\infty$
	rules	1.011	1.027	1.050	1.222	1.927	4.851*	3.446$^\infty$	2.378$^\infty$	1.551$^\infty$	0.754$^\infty$
	time	1.004	1.022	1.049	1.230	1.953	4.905*	3.487$^\infty$	2.401$^\infty$	1.577$^\infty$	0.792$^\infty$
	coll	0.069	0.141	0.242	0.423	0.683	0.907*	0.907$^\infty$	0.902$^\infty$	0.911$^\infty$	0.919$^\infty$
sym.cache3	states	1.008	1.022	1.064	1.213	1.919	5.345*	4.327$^\infty$	3.436$^\infty$	2.672$^\infty$	1.304$^\infty$
	rules	1.010	1.026	1.072	1.233	1.970	5.138*	3.742$^\infty$	2.772$^\infty$	1.987$^\infty$	0.863$^\infty$
	time	1.024	1.043	1.093	1.266	2.035	5.388*	4.011$^\infty$	3.041$^\infty$	2.248$^\infty$	1.045$^\infty$
	coll	0.064	0.137	0.251	0.424	0.688	0.906*	0.908$^\infty$	0.913$^\infty$	0.926$^\infty$	0.924$^\infty$
down	states	1.001	1.006	1.013	1.023	1.038	1.058	1.104	1.202	1.416	1.004$^\infty$
	rules	1.001	1.005	1.011	1.019	1.032	1.050	1.092	1.179	1.365	0.747$^\infty$
	time	0.785	0.822	0.859	0.881	0.911	0.948	1.022	1.126	1.333	0.763$^\infty$
	coll	0.057	0.127	0.219	0.321	0.424	0.527	0.645	0.756	0.862	0.902$^\infty$
kerb	states	1.001	1.001	1.003	1.007	1.015	1.025	1.048	1.097	1.228	1.107$^\infty$
	rules	1.003	1.005	1.012	1.022	1.038	1.057	1.092	1.151	1.285	1.012$^\infty$
	time	0.982	0.986	0.987	0.994	1.004	1.015	1.041	1.090	1.221	0.828$^\infty$
	coll	0.060	0.122	0.207	0.305	0.409	0.512	0.618	0.727	0.837	0.910$^\infty$
list6	states	1.006	1.014	1.044	1.151	1.735	5.041*	4.528$^\infty$	3.460$^\infty$	2.392$^\infty$	1.025$^\infty$
	rules	1.007	1.015	1.045	1.149	1.721	4.531*	3.715$^\infty$	2.682$^\infty$	1.765$^\infty$	0.732$^\infty$
	time	0.999	0.999	1.038	1.157	1.752	4.701*	3.866$^\infty$	2.818$^\infty$	1.886$^\infty$	0.792$^\infty$
	coll	0.063	0.131	0.236	0.392	0.654	0.901*	0.912$^\infty$	0.914$^\infty$	0.917$^\infty$	0.904$^\infty$
list6too	states	1.006	1.030	1.082	1.232	1.516	4.643*	3.714*	2.786$^\infty$	1.857$^\infty$	22.284$^\infty$
	rules	1.006	1.035	1.087	1.240	1.571	4.689*	3.505*	2.459$^\infty$	1.674$^\infty$	6.290$^\infty$
	time	1.024	1.053	1.106	1.266	1.595	4.768*	3.553*	2.489$^\infty$	1.665$^\infty$	0.700$^\infty$
	coll	0.062	0.151	0.291	0.440	0.621	0.903*	0.912*	0.908$^\infty$	0.910$^\infty$	0.915$^\infty$
newlist6	states	1.009	1.022	1.049	1.198	1.830	5.136*	3.987$^\infty$	3.527$^\infty$	2.300$^\infty$	1.227$^\infty$
	rules	1.009	1.023	1.052	1.198	1.816	4.641*	3.328$^\infty$	2.783$^\infty$	1.734$^\infty$	0.888$^\infty$
	time	1.005	1.020	1.052	1.198	1.810	4.656*	3.320$^\infty$	2.777$^\infty$	1.725$^\infty$	0.868$^\infty$
	coll	0.066	0.134	0.239	0.415	0.671	0.903*	0.900$^\infty$	0.915$^\infty$	0.914$^\infty$	0.922$^\infty$
mcslock1	states	1.009	1.020	1.058	1.190	1.900	4.948*	4.187$^\infty$	3.299$^\infty$	2.368$^\infty$	1.100$^\infty$
	rules	1.009	1.020	1.058	1.190	1.900	4.613*	3.709$^\infty$	2.818$^\infty$	1.958$^\infty$	0.895$^\infty$
	time	1.015	1.030	1.074	1.198	1.890	4.547*	3.605$^\infty$	2.678$^\infty$	1.799$^\infty$	0.798$^\infty$
	coll	0.069	0.140	0.248	0.412	0.685	0.900*	0.904$^\infty$	0.911$^\infty$	0.917$^\infty$	0.911$^\infty$
mcslock2	states	1.009	1.017	1.032	1.063	1.178	2.751*	4.008$^\infty$	3.004$^\infty$	2.269$^\infty$	1.061$^\infty$
	rules	1.009	1.017	1.032	1.063	1.178	2.749*	3.742$^\infty$	2.710$^\infty$	1.986$^\infty$	0.903$^\infty$
	time	0.978	0.991	1.009	1.046	1.163	2.734*	3.721$^\infty$	2.692$^\infty$	1.951$^\infty$	0.879$^\infty$
	coll	0.092	0.152	0.236	0.343	0.491	0.818*	0.900$^\infty$	0.900$^\infty$	0.912$^\infty$	0.906$^\infty$
ns-old	states	1.000	1.011	1.000	1.019	1.029	1.103	1.259	2.140	1.784*	0.892$^\infty$
	rules	1.000	1.059	1.000	1.095	1.100	1.315	1.460	2.558	1.770*	0.813$^\infty$
	time	1.014	1.058	1.043	1.072	1.087	1.246	1.406	2.377	1.536*	0.754$^\infty$
	coll	0.047	0.125	0.192	0.302	0.428	0.549	0.682	0.870	0.917*	0.925$^\infty$
ns	states	1.001	1.000	1.010	1.046	1.024	1.057	1.147	2.476	2.041*	1.020$^\infty$
	rules	1.032	1.000	1.066	1.140	1.047	1.128	1.284	2.998	1.974*	0.891$^\infty$
	time	1.085	1.068	1.102	1.169	1.119	1.169	1.305	2.864	1.746*	0.746$^\infty$
	coll	0.042	0.098	0.202	0.326	0.409	0.530	0.659	0.886	0.919*	0.932$^\infty$
sci	states	1.003	1.008	1.020	1.041	1.095	1.369	4.067*	3.243$^\infty$	2.089$^\infty$	1.099$^\infty$
	rules	1.003	1.009	1.022	1.045	1.102	1.382	3.975*	2.919$^\infty$	1.749$^\infty$	0.850$^\infty$
	time	1.017	1.016	1.025	1.048	1.104	1.392	3.970*	2.912$^\infty$	1.752$^\infty$	0.858$^\infty$
	coll	0.062	0.127	0.219	0.328	0.453	0.635	0.902*	0.908$^\infty$	0.906$^\infty$	0.912$^\infty$

Fig. 5. Cache Based BF Visit (bit compression `-b`)

	Mem	1	0.9	0.8	0.7	0.6	0.5	0.4	0.3	0.2	0.1
`n_peterson`	states	1.000	1.005	1.024	1.225	4.262*	5.021*	4.036*	2.988$^\infty$	1.990$^\infty$	0.998$^\infty$
	rules	1.000	1.005	1.024	1.225	4.262*	4.988*	3.996*	2.942$^\infty$	1.944$^\infty$	0.957$^\infty$
	time	1.008	1.033	1.079	1.320	5.014*	5.993*	4.797*	3.560$^\infty$	2.374$^\infty$	1.207$^\infty$
	coll	0.006	0.104	0.219	0.428	0.859*	0.901*	0.901*	0.900$^\infty$	0.900$^\infty$	0.900$^\infty$
`adash`	states	1.000	1.009	1.025	1.133	1.565	5.064*	4.491$^\infty$	3.631$^\infty$	2.293$^\infty$	0.860$^\infty$
	rules	1.000	1.009	1.025	1.133	1.568	4.684*	3.895$^\infty$	2.931$^\infty$	1.763$^\infty$	0.572$^\infty$
	time	0.973	0.994	1.016	1.134	1.593	4.855*	4.046$^\infty$	3.081$^\infty$	1.863$^\infty$	0.609$^\infty$
	coll	0.002	0.103	0.207	0.380	0.611	0.902*	0.911$^\infty$	0.918$^\infty$	0.912$^\infty$	0.900$^\infty$
`adashbug`	states	1.000	1.006	1.012	1.032	1.061	1.135	1.268	1.532	2.138*	1.336$^\infty$
	rules	1.000	1.000	1.001	1.008	1.021	1.065	1.142	1.299	1.708*	0.961$^\infty$
	time	1.006	1.014	1.020	1.034	1.054	1.110	1.213	1.397	1.872*	1.080$^\infty$
	coll	0.001	0.133	0.235	0.336	0.446	0.567	0.686	0.802	0.912*	0.935$^\infty$
`eadash`	states	1.000	1.009	1.056	1.365	6.083*	5.124$^\infty$	4.847$^\infty$	3.813$^\infty$	2.285$^\infty$	1.041$^\infty$
	rules	1.000	1.009	1.055	1.366	5.568*	4.344$^\infty$	3.893$^\infty$	2.916$^\infty$	1.645$^\infty$	0.670$^\infty$
	time	1.005	1.014	1.065	1.401	5.870*	4.523$^\infty$	4.078$^\infty$	3.133$^\infty$	1.802$^\infty$	0.761$^\infty$
	coll	0.007	0.108	0.242	0.488	0.902*	0.902$^\infty$	0.918$^\infty$	0.922$^\infty$	0.913$^\infty$	0.906$^\infty$
`ldash`	states	1.000	1.028	1.284	7.189*	7.047$^\infty$	5.953$^\infty$	4.016	3.722	2.349	1.016
	rules	1.000	1.027	1.278	6.394*	5.861$^\infty$	4.719$^\infty$	3.048$^\infty$	2.705$^\infty$	1.622$^\infty$	0.062$^\infty$
	time	1.013	1.075	1.346	6.851*	6.177$^\infty$	4.987$^\infty$	3.278$^\infty$	2.946$^\infty$	1.762$^\infty$	0.727$^\infty$
	coll	0.007	0.125	0.377	0.903*	0.915$^\infty$	0.916$^\infty$	0.901$^\infty$	0.920$^\infty$	0.915$^\infty$	0.902$^\infty$
`cache3multi`	states	1.000	1.013	1.089	1.297	2.939	5.168*	4.440$^\infty$	3.421$^\infty$	2.329$^\infty$	0.946$^\infty$
	rules	1.000	1.017	1.104	1.330	3.078	4.929*	3.985$^\infty$	2.862$^\infty$	1.835$^\infty$	0.713$^\infty$
	time	0.978	1.005	1.107	1.363	3.344	5.418*	4.399$^\infty$	3.157$^\infty$	2.026$^\infty$	0.780$^\infty$
	coll	0.003	0.114	0.264	0.457	0.798	0.905*	0.911$^\infty$	0.915$^\infty$	0.918$^\infty$	0.903$^\infty$
`newcache3`	states	1.000	1.022	1.053	1.313	2.155	5.049*	4.131$^\infty$	3.213$^\infty$	1.836$^\infty$	1.148$^\infty$
	rules	1.000	1.020	1.052	1.330	2.192	4.646*	3.412$^\infty$	2.564$^\infty$	1.347$^\infty$	0.742$^\infty$
	time	1.032	1.061	1.111	1.450	2.483	5.462*	4.026$^\infty$	3.021$^\infty$	1.604$^\infty$	0.919$^\infty$
	coll	0.003	0.136	0.239	0.488	0.725	0.903*	0.908$^\infty$	0.911$^\infty$	0.909$^\infty$	0.936$^\infty$
`sym.cache3`	states	1.000	1.012	1.058	1.217	1.871	5.027*	4.358$^\infty$	3.500$^\infty$	2.641$^\infty$	1.304$^\infty$
	rules	1.000	1.013	1.065	1.235	1.921	4.756*	3.764$^\infty$	2.814$^\infty$	1.950$^\infty$	0.867$^\infty$
	time	1.020	1.054	1.142	1.384	2.298	6.182*	5.011$^\infty$	3.850$^\infty$	2.730$^\infty$	1.278$^\infty$
	coll	0.005	0.109	0.245	0.427	0.679	0.901*	0.910$^\infty$	0.914$^\infty$	0.925$^\infty$	0.927$^\infty$
`down`	states	1.000	1.002	1.009	1.019	1.037	1.067	1.102	1.179	1.393	1.369$^\infty$
	rules	1.000	1.001	1.007	1.016	1.031	1.058	1.089	1.160	1.347	1.040$^\infty$
	time	1.098	1.426	1.885	2.221	2.623	3.041	3.475	4.049	5.082	4.270$^\infty$
	coll	0.007	0.100	0.206	0.312	0.419	0.530	0.636	0.745	0.856	0.929$^\infty$
`kerb`	states	1.000	1.001	1.002	1.006	1.013	1.026	1.047	1.093	1.223	1.116$^\infty$
	rules	1.000	1.003	1.008	1.018	1.034	1.060	1.090	1.146	1.280	1.020$^\infty$
	time	0.983	0.990	1.002	1.011	1.026	1.054	1.078	1.135	1.274	0.891$^\infty$
	coll	0.007	0.101	0.202	0.305	0.407	0.513	0.618	0.726	0.836	0.911$^\infty$
`list6`	states	1.000	1.009	1.036	1.163	2.015	5.254*	4.742$^\infty$	3.759$^\infty$	2.478$^\infty$	1.111$^\infty$
	rules	1.000	1.009	1.037	1.160	1.997	4.670*	3.868$^\infty$	2.880$^\infty$	1.825$^\infty$	0.779$^\infty$
	time	0.996	1.039	1.121	1.332	2.631	6.946*	5.840$^\infty$	4.406$^\infty$	2.823$^\infty$	1.217$^\infty$
	coll	0.006	0.107	0.229	0.400	0.703	0.906*	0.917$^\infty$	0.922$^\infty$	0.923$^\infty$	0.919$^\infty$
`newlist6`	states	1.000	1.007	1.050	1.226	1.897	5.443*	4.140$^\infty$	3.373$^\infty$	2.300$^\infty$	1.073$^\infty$
	rules	1.000	1.008	1.053	1.224	1.882	4.863*	3.434$^\infty$	2.674$^\infty$	1.738$^\infty$	0.776$^\infty$
	time	1.046	1.066	1.140	1.360	2.192	6.009*	4.263$^\infty$	3.335$^\infty$	2.155$^\infty$	0.944$^\infty$
	coll	0.002	0.100	0.237	0.434	0.682	0.909*	0.905$^\infty$	0.912$^\infty$	0.917$^\infty$	0.917$^\infty$
`mcslock1`	states	1.000	1.010	1.046	1.163	1.848	5.160*	4.145$^\infty$	3.341$^\infty$	2.326$^\infty$	1.100$^\infty$
	rules	1.000	1.010	1.046	1.163	1.848	4.817*	3.680$^\infty$	2.849$^\infty$	1.926$^\infty$	0.893$^\infty$
	time	0.982	1.025	1.106	1.284	2.217	6.304*	4.776$^\infty$	3.652$^\infty$	2.410$^\infty$	1.085$^\infty$
	coll	0.006	0.104	0.232	0.395	0.674	0.903*	0.903$^\infty$	0.911$^\infty$	0.915$^\infty$	0.910$^\infty$
`mcslock2`	states	1.000	1.006	1.022	1.055	1.177	4.444*	4.206$^\infty$	3.038$^\infty$	2.279$^\infty$	1.063$^\infty$
	rules	1.000	1.006	1.022	1.055	1.177	4.427*	3.915$^\infty$	2.734$^\infty$	1.993$^\infty$	0.905$^\infty$
	time	1.023	1.092	1.181	1.301	1.538	6.956*	6.202$^\infty$	4.325$^\infty$	3.043$^\infty$	1.335$^\infty$
	coll	0.009	0.105	0.217	0.337	0.490	0.888*	0.905$^\infty$	0.901$^\infty$	0.912$^\infty$	0.906$^\infty$
`sci`	states	1.000	1.005	1.013	1.037	1.087	1.313	4.013*	3.133$^\infty$	2.309$^\infty$	1.154$^\infty$
	rules	1.000	1.006	1.014	1.041	1.093	1.325	3.950*	2.793$^\infty$	1.916$^\infty$	0.894$^\infty$
	time	1.027	1.043	1.061	1.105	1.180	1.458	4.544*	3.225$^\infty$	2.226$^\infty$	1.047$^\infty$
	coll	0.005	0.108	0.210	0.331	0.450	0.618	0.902*	0.906$^\infty$	0.917$^\infty$	0.920$^\infty$

Fig. 6. Cache Based BF Visit (bit compression and hash compaction `-b -c`)

m	SizeC	MemC	SizeQ	MemQ	Visited	Time	Coll	MaxQ	MaxMemQ
200	38,836,153	180	215,756	20	44,780,625	19,349.80	0.17024	16,215,144	1113.40
150	29,127,121	135	161,817	15	46,369,727	19,975.79	0.373777	16,239,606	1115
130	25,243,507	117	140,241	13	48,933,778	21,769.69	0.484372	16,264,901	1116.8

Fig. 7. Cache Based BF Visit for protocol `ns` with parameters: `NumInitiators` = 2, `NumResponders` = 1, `NumIntruders` = 2, `NetworkSize` = 2, `MaxKnowledge` = 10. Results obtained using options `-b` (bit compression), `-c` (40 bit hash compaction) and `-ndl` (no deadlock detection) on a Pentium III 866Mhz machine with 256Mb of RAM.

We give the meaning of rows and columns in Fig. 5. Those in Fig. 6 of course have exactly the same meaning, only they refer to the case in which both bit compression and hash compaction (`-b -c`) are used.

Column α (with $\alpha = 1, 0.9, \ldots, 0.1$) gives information about the run of protocol p with memory $\alpha M(p)$. Row ***States*** gives the ratio between the visited states when using memory $\alpha M(p)$ and $|\text{Reach}(p)|$. This is the *state* overhead due to revisiting already visited states. Row ***Rules*** gives the ratio between the rules fired when using memory $\alpha M(p)$ and RulesFired(p). This is the *rule* overhead due to revisiting already visited states. Row ***Time*** gives the ratio between the time (in seconds) to complete state space exploration when using memory $\alpha M(p)$ and option w and $T_w(p)$. Option w can be bit compression ($w = b$) or bit compression and hash compaction ($w = bc$). This is the time overhead. Row ***Coll*** gives the collision rate. That is the ratio between the number of collisions and the number of states inserted in the cache.

Rows *States*, *Rules*, *Time* should have close values. As shown in Figs. 5, 6 this is indeed the case.

Our state space exploration may not terminate. We stop our visit when the collision rate is greater than 0.9. or the max depth exceeds a given threshold.

Note that when our visit is stopped prematurely (for the above reasons) we do not know a priori if all reachable states have been visited. However, for the experiments in Figs. 5, 6 we have such information since they fit in our memory. In Figs. 5, 6 we report such information just to give an idea of the behaviour of our approach when a visit is stopped prematurely.

We mark with a $*$ superscript the data obtained when the following conditions are satisfied: 1. The visit has been stopped because the collision rate exceeded 0.9 or because the max depth exceeded a certain threshold; 2. All reachable states have been visited.

We mark with a ∞ superscript the data obtained when the following conditions are satisfied: 1. The visit has been stopped because the collision rate exceeded 0.9 or because the max depth exceeded a certain threshold; 2. There are reachable states that have not been visited.

The experimental results in Figs. 5, 6 show that our cache based approach typically saves about 40% of RAM w.r.t. to an approach based on hash table. This holds for case `b` (bit compression) as well as for case `bc` (bit compression and hash compaction).

For case `b` time penalty for column 0.6 (40% RAM saving) ranges from 0 to 150% with an average value of about 50%.

For case `bc` time penalty for column 0.6 (40% RAM saving) ranges from 0 to 234% with an average value of about 100%.

Note that with our cache based approach the amount of memory needed does not increase with the max depth. Instead in a hash table based approach using a max depth limit (e.g. as in SPIN) the memory required increases with the max depth.

We also wanted to test our cache based approach with a large protocol that heavily loads our machine. The results are in Fig. 7. The meaning of the columns of Fig. 7 is as follows.

Column **m** gives the total memory (in Megabytes) given to cached Murφ to complete the verification. Column **SizeC** gives the max number of states that the cache can contain (each state takes 40 bits with hash compaction). Column **MemC** gives the memory (in Megabytes) reserved for the cache (0.9***m**). Column **SizeQ** gives the max number of states that the RAM queue can contain (each state takes 68 bytes in the queue). Column **MemC** gives the memory (in Megabytes) reserved for the RAM queue (0.1 * **m**). Column **Visited** gives the number of states visited by the verifier. Column **Time** gives the time (in seconds) taken to complete the verification task. Column **Coll** gives the collision rate. Column **MaxQ** is the max number of states contained in the BF queue (RAM + disk) during the verification. Column **MaxMemQ** is the max amount of memory (in Megabytes) taken by the BF queue (RAM + DISK).

When **m** = 200 the collision rate is low. Thus the number of visited states is about the number of reachable states. This means that to complete the verification of the protocol in Fig. 7 would require more than 1313MB of RAM using standard Murφ (i.e. 200MB for the hash table, 1113MB for the queue). This is more than our machine can handle (see Fig. 7). However, using our cache based Murφ, we were able to complete verification using only 130MB of RAM.

As shown in Figs. 5, 6, 7 there is a time-space tradeoff when using our cache based approach. However for verification tasks such tradeoff is often acceptable and in any case better than being left with an `out of memory` message after hours of computation.

5 Conclusions

We showed experimentally (Sect. 2) that protocols exhibit *transition locality*. We supported our claim by measuring transition locality for the set of protocols included the Murφ verifier distribution.

We devised a verification algorithm to exploit transition locality and implemented it within the Murφ verifier (Sect. 3). Essentially our approach replaces the hash table used in a BF state exploration with a fixed size cache (i.e. no collision detection) and uses disk storage for the BF queue.

Our experimental results (Sect. 4) show that, w.r.t. a hash table based approach, our cache based approach typically allows verification of systems more than 40% larger with a time overhead of about 100%.

Our approach is compatible with all (BF) optimization techniques present in the Murφ verifier and, because of its time penalty, it is intended to be used only when such techniques are not enough to avoid running out of memory.

The auxiliary memory used in our algorithm can also be implemented using a NOW (*Network Of Workstations*) rather than with (or together with) disk storage. Moreover, our preliminary analysis shows that our cache memory (holding visited states) can also be implemented using secondary storage (namely disk storage). We are currently working on both such issues.

We think that there are other statistical properties of system transition graphs that can be used to improve performances of state space exploration algorithms. This is a natural next step for our research.

Acknowledgements. We are grateful to our anonymous referees for their 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] 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.

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

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

[6] 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.

[7] 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.

[8] 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.

[9] Heh-Tyan Liaw and Chen-Shang Lin. On the obdd-representation of general boolean functions. *IEEE Trans. on Computers*, C-41(6), June 1992.

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

[11] A. Papoulis. *Probability, Random Variables and Stochastic Processes.* McGraw-Hill Series in System Sciences, 1965.

[12] D. A. Patterson and J. L. Hennessy. *Computer Architecture A Quantitative Approach.* Morgan Kaufmann, 1996.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

[18] 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*, pages 206–224, 1995.

[19] 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.

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

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

[22] 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.