Int J Softw Tools Technol Transfer (2004) 6: 320–341 / Digital Object Identifier (DOI) 10.1007/s10009-004-0149-6

Regular contribution

Exploiting transition locality in automatic verification of finite-state concurrent systems

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

[1] Dipartimento di Informatica, Università di L'Aquila, Coppito 67100, L'Aquila, Italy
e-mail: {dellapenna,intrigila,melatti}@di.univaq.it
[2] Dip. di Scienze dell'Informazione, Università di Roma "La Sapienza", Via Salaria 113, 00198 Roma, Italy
e-mail: {tronci,zilli}@di.uniroma1.it

Published online: 6 July 2004 –

Abstract. In this paper we show that statistical properties of the transition graph of a system to be verified can be exploited to improve memory or time performances of verification algorithms.

We show experimentally that protocols exhibit *transition locality*. That is, with respect to levels of a breadth-first state space exploration, state transitions tend to be between states belonging to close levels of the transition graph. We support our claim by measuring transition locality for the set of protocols included in the Murφ verifier distribution.

We present a *cache-based verification algorithm* that exploits transition locality to decrease memory usage and a *disk-based verification algorithm* that exploits transition locality to decrease disk read accesses, thus reducing the time overhead due to disk usage. Both algorithms have been implemented within the Murφ verifier.

Our experimental results show that our cache-based algorithm can typically save more than 40% of memory with an average time penalty of about 50% when using (Murφ) bit compression and 100% when using bit compression and hash compaction, whereas our disk-based verification algorithm is typically more than ten times faster than a previously proposed disk-based verification algorithm and, even when using 10% of the memory needed to complete verification, it is only between 40 and 530% (300% on average) slower than (RAM) Murφ with enough memory to complete the verification task at hand. Using just 300 MB of memory our disk-based Murφ was able to complete verification of a protocol with about 10^9 reachable states. This would require more than 5 GB of memory using standard Murφ.

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

This paper is an extended version of the papers [27, 39].

Keywords: Automatic verification – Model checking – Explicit state space exploration

1 Introduction

State space exploration (*reachability analysis*) is at the very heart of all algorithms for automatic verification of concurrent systems. In fact, checking that a *finite-state system* (FSS) satisfies a given property (e.g., a safety property) typically entails exploring (visiting) all reachable system states.

Unfortunately, reachability analysis is plagued by the well-known *state explosion* problem: that is, our computer runs out of memory because of the huge number of reachable states that have to be stored in memory.

Essentially there are two main techniques to perform state space exploration: *explicit* and *symbolic*. *Explicit* state space exploration uses a hash table to store the set of visited states, whereas *symbolic* state space exploration represents the set of visited states with its characteristic function which, in turn, is represented and manipulated using *ordered binary-decision diagrams* (OBDDs) [3, 4].

For protocol verification, *explicit* state space exploration often outperforms *symbolic* state space exploration [18]. Indeed, our results [26] show that explicit state space exploration can be quite useful also for automatic verification of *hybrid systems*. Since here we are mainly interested in protocol verification, we focus on explicit state space exploration: tools based on this technique are, e.g., SPIN [15, 32] and Murφ [7, 23].

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 can be found in [16, 17, 19, 20, 28, 35, 36, 42].

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 can be found in [29, 30, 33, 34, 38].

In this paper we study the possibility of exploiting a statistical property of protocol transition graphs, namely, the *transition locality*, to improve memory and time performances of state verification algorithms. This is quite similar to what is usually done when optimizing a CPU on the basis of program profiling [25]. Statistical properties of protocols have also been studied in the context of *probabilistic verification* [40].

We show experimentally (Sect. 3) that protocols exhibit *transition locality*. That is, transitions tend to be local with respect to levels of a breadth-first (BF) search. We support our claim by measuring transition locality for the set of protocols included in the Murφ verifier distribution.

Finally, we present a *cache-based BF state space exploration* algorithm (Sect. 4) that exploits transition locality to decrease memory usage and a *disk-based BF state space exploration* algorithm (Sect. 5) that exploits transition locality to save disk read accesses, thus reducing the time overhead with respect to a normal disk-based verification.

Both algorithms have been implemented within the Murφ verifier [23] and are compatible with all the well-known state compression techniques (such as those in [16, 36]) and, in particular, with all state reduction techniques present in Murφ.

By using Murφ, we can take advantage of a full-featured verifier to test our algorithms together with all the other memory reduction techniques already implemented in Murφ, and we have a large set of benchmark protocols to test. Of course, our algorithms can be included within any explicit verifier implementing a BF search (e.g., the latest SPIN version).

1.1 Cache-based BF state space exploration

In our *cache-based BF state space exploration* algorithm we essentially replace the hash table used in a memory-based BF state space exploration with a fixed-size *cache memory* (i.e., no collision detection is done), using auxiliary (disk) storage for the BF queue.

Therefore, we do not incur state explosion, but we may incur nontermination when our cache memory is *too small*: in fact, in this case we may visit the same set of states over and over. However, upon termination we are guaranteed that all reachable states have been visited. To the best of our knowledge this is the first time that such a fixed size memory state space exploration algorithm has been presented.

Note that our use of the cache memory differs from that in [42] (see also [8, 13, 14]). In fact [42] presents a state compression technique, and "no collision detection" in [42] refers to state signatures [34, 35], that is, a (signature) collision in [42] may lead to declare as visited a nonvisited state, whereas our algorithm simply forgets a visited state upon collision, thus possibly declaring as nonvisited a visited state.

The implementation of our cache-based algorithm within the Murφ verifier (Sect. 4.2), named *CMurphi3.1*, can be downloaded from [5].

Experiments using our cache-based algorithm (Sect. 6.2) show that on average 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.

1.2 Disk-based BF state space exploration

Our *disk-based BF state space exploration* algorithm is obtained from the one in [34] by using only a suitable subset of the visited states stored on disk to clean up the BF queue. By reducing disk read accesses we also reduce our time overhead with respect to a memory-based BF state space exploration.

Note that the SPIN verifier can use disk storage for the depth-first (DF) stack. However, visited states are still stored in a memory-based hash table, which is where state explosion typically occurs.

The implementation of our disk-based algorithm within the Murφ verifier (Sect. 5), named *CMurphi4.1*, can be downloaded from [5].

Experimental results on using our disk-based algorithm (Sect. 6.4) show that, even when using 10% of the memory needed to complete verification, our disk-based Murφ is only between 40 and 530% times slower (300% on average) than (memory-based) standard Murφ with enough memory to complete the verification task at hand. Moreover, our disk-based algorithm is typically more than ten times faster than the disk-based algorithm presented in [34].

Using just 300 MB of memory our disk-based Murφ was able to complete verification of a protocol with almost 10^9 reachable states. Using standard Murφ this protocol would require more than 5 GB of memory.

2 Background

In this section we give some basic information that will be useful in the subsequent discussion.

2.1 Finite-state systems

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' labelled a *if and only if $R(s, a, s')$ holds. We say that* there is a transition from s to s' *(notation $R(s, s')$) if and only if there exists $a \in A$ so that $R(s, a, s')$ holds. The set of successors of state s (notation $\mathtt{next}(s)$) is the set of states s' so that $R(s, s')$.*
3. *The set of* reachable states *of $\mathcal{S}$ (notation $\mathbf{Reach}(\mathcal{S})$) is the set of states of $\mathcal{S}$ reachable in 0 (zero) or more steps from I.*
   Formally, $\mathbf{Reach}(\mathcal{S})$ is the smallest set so that
   1. $I \subseteq \mathbf{Reach}(\mathcal{S})$,
   2. For all $s \in \mathbf{Reach}(\mathcal{S})$, $\mathtt{next}(s) \subseteq \mathbf{Reach}(\mathcal{S})$.

In the following discussion we will always refer to a given system $\mathcal{S} = (S, I, A, R)$. Thus, for example, we will write **Reach** for $\mathbf{Reach}(\mathcal{S})$. 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*, 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 with the initial states in I. This can be done, e.g., using a *depth-first* (DF) search or a *breadth-first* (BF) search. For example, Murφ [23] and (the latest version of) SPIN [32] may use a DF as well as a BF search.

In the following discussion we will focus on BF search. As is well known, a BF search defines *levels* on the transition graph. Initial states (i.e., states in I) are at level 0. The states in $(\mathtt{next}(I) - I)$ (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.*
$L(0) = I$,
$L(k+1) = \{s' |$ *there exists $s \in L(k)$ so that* $[R(s, s')$ *and* $s' \notin \cup_{i=0}^{i=k} L(i)]\}$.

Given a state $s \in \mathbf{Reach}$, we define $\mathrm{level}(s) = k$ *if and only if $s \in L(k)$. That is,* $\mathrm{level}(s)$ *is the level of state s in a BF search of $\mathcal{S}$.*

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

2.2 The Murφ verifier

In this section we give a short overview of the Murφ verifier. For further details we refer the reader to [7, 23].

From a conceptual point of view Murφ takes as input a *finite-state system* $\mathcal{S}$ and checks whether a given invariant property φ for $\mathcal{S}$ is satisfied.

An *invariant property* can be seen as a map from (the states of) $\mathcal{S}$ to the set $\mathcal{B} = \{0, 1\}$ of boolean values. Thus the verification goal is to check that, for each state s, reachable from an initial state of $\mathcal{S}$, the given invariant φ holds (i.e., that $\varphi(s) = 1$). Of course, in general this check entails visiting all reachable system states.

Figure 1 shows the standard BF state space exploration algorithm. Essentially, this is the algorithm used by Murφ to visit the state space of a given system $\mathcal{S}$. The algorithm makes use of two main memory data structures. A *queue*, where states are stored and retrieved (in FIFO order) during the search, and a *hash table* used to store all visited states. In Fig. 1 invariants for state `s` may be checked whenever function `Enqueue(Q, s)` is called.

Since $\mathcal{S}$ is a finite-state system, the algorithm in Fig. 1 always terminates since we never visit the same state more than once.

Note that Murφ (like SPIN) represents states *explicitly*: each visited state is stored in memory (namely, in the hash table). There are model checkers that represent states *symbolically*. In symbolic model checking the set V of visited states is represented by its characteristic function f (i.e., $f(s) = \mathbf{if}\, s \in V\, \mathbf{then}\, 1\, \mathbf{else}\, 0$) and suitable data structures (such as OBDDs, *ordered binary-decision diagrams* [3]) are used to represent f. Examples of symbolic model checkers are SMV [31], UPPAAL [21, 41] (both based on OBDDs), and Hytech [1, 2, 12], which is based on polyedra in a multidimensional real space [6, 9–11].

Explicit model checkers (such as Murφ and SPIN) typically perform better on *softwarelike* (asynchronous) systems [18], whereas *symbolic* model checkers (such as SMV) typically perform better on *hardwarelike* (synchronous) systems.

```
FIFO_Queue Q;
HashTable T;

bfs(init_states, next)
{
 /* load Q with initial states */
 foreach s in init_states Enqueue(Q, s);

 /* mark init states as visited */
 foreach s in init_states Insert(T, s);

 /* visit */
 while (Q is not empty)
 {
  s = Dequeue(Q);
  foreach s' in next(s)
   if (s' is not in T)
   {
    Insert(T, s');
    Enqueue(Q, s');
   }
 }
}
```

Fig. 1. Explicit breadth-first search

Since explicitly storing each state can take a large amount of memory, Murφ offers several techniques to reduce the state size. In the development of our algorithms, two techniques are very important: bit compression [23] and hash compaction [34, 35].

When bit compression is enabled, Murφ saves memory by using every bit of the *state descriptor*, the memory structure maintaining the state variables. On the other hand, when bit compression is not used, state variables are aligned on byte boundaries of the state descriptor, thus wasting some space. Typically, bit compression saves on average 300% of memory; however, it slows down the verifier by approximately 25%.

When using hash compaction, compressed values (also called *state signatures*) are stored in the hash table instead of full state descriptors; however, there is a certain probability that some states will be omitted during verification, since different states may have the same signature. By default, Murφ compresses state descriptors up to 40 bits.

Hash compaction can be combined with bit compression: in this case, bit compression is applied on the state queue, whereas hash compaction is used on the hash table.

3 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 [23] plus the Kerberos protocol from [37]. Most of these protocols are scalable since they depend on a set of parameters. By modifying such parameters, the size of the protocol state space can be increased. To save time, in most of our experiments we set protocol parameters so as to obtain small-sized models. This is reasonable, since locality is a structural property of a protocol transition graph. That is, locality may or may not hold independently of the size of the transition graph.

The protocols tested cover a wide range of concurrent software typologies such as synchronization, authentication, cache coherence, distributed locks, etc. Thus we have a fairly representative benchmark set.

3.1 Transition locality

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 *if and only if* $|\text{level}(s') - \text{level}(s)| \leq k$.

Transition $R(s, a, s')$ is said to be a k-transition *if and only if* $\text{level}(s') - \text{level}(s) = k$. *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' on 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 by $N(s, k)$ the number of k-transitions from s and by $N(s)$ 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, which represents the probability of getting a k-transition when picking at random a transition from s. Of course, if s is on 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 by $\boldsymbol{\Delta}(k)$. The expected value $E\{\boldsymbol{\Delta}(k)\}$ of $\boldsymbol{\Delta}(k)$ is the average value of $\delta(s, k)$ on all reachable states. That is: $E\{\boldsymbol{\Delta}(k)\} = \frac{1}{|\textbf{\textit{Reach}}|} \sum_{s \in \textbf{\textit{Reach}}} \delta(s, k)$.*

4. As usual, we denote by $\sigma^2(k)$ the variance of $\boldsymbol{\Delta}(k)$ and by $\sigma(k)$ its standard deviation [24].

Our choice of $\boldsymbol{\Delta}(k)$ to measure locality stems from the fact that we also want to known how 1-local transitions are distributed in the transition graph. That is, we want to know if 1-local transitions are all concentrated on some part of the graph or if they are more or less uniformly distributed in the transition graph. This can be measured by computing the standard deviation of $\boldsymbol{\Delta}(k)$.

3.2 Measuring locality

We measured locality using the Murφ verifier [23]. This gives us a wide benchmark set for free since many protocols have already been defined using the Murφ input language. We modified Murφ so as to compute $E\{\boldsymbol{\Delta}(k)\}$ and $\sigma(k)$ while carrying out state space exploration.

All the experiments were performed using bit compression (Murφ option `-b`) and disabling deadlock detection (option `-ndl`). 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, deadlock detection has been disabled. This allows us to explore the whole reachable part of the transition graph also for protocols containing deadlocks (e.g., Kerberos `kerb`).

3.3 Experimental results about locality

We want to show experimentally that transitions in protocols tend to be *local* with respect to levels. We do this by showing that for the set of protocols shown in Table 1 most transitions are 1-local, that is, they are k-transitions with $k = -1, 0, 1$. In other words, for most transitions, $R(s, a, s')$, s' will be either on the same level as s (0-transition) or on the next level (1-transition) or on the previous level (-1-transition). More information (e.g., number of reachable states, etc.) about the protocols in Table 1 are available in Sect. 6, Table 3.

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 this value is close to 1, almost all transitions in the protocol are 1-local, whereas when it is close to 0, there are almost no 1-local transitions in the protocol.

To get some understanding of how 1-local transitions are distributed in the transition graph, we compute the standard deviation $\sigma(k)$ of $\mathbf{\Delta}(k)$. In fact, 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 results of our experiments are shown in Table 1 where, for each protocol and for $k = -1, 0, 1$, we show $E\{\mathbf{\Delta}(k)\}$, $\sigma(k)$ and $SumAvg$. Our measures 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. Note that the three protocols with superscript * in Table 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. Our findings can be summarized as follows.

Experimental Fact 1. *From column SumAvg of Table 1 we observe that, in all protocols of our benchmark set, for most states more than* 60% *of the transitions are 1-local. Indeed, for most of these protocols, we have that for most states more than* 85% *of the transitions are 1-local. 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 $s \in \mathbf{Reach}$, in most cases most transitions from s (say, about 85%) will be 1-local.

On the other hand, it is not difficult to define a simple example of a nonlocal system. Consider the system NLS defined in Fig. 2, whose transition graph is shown in Fig. 3. NLS is highly nonlocal: in fact, following the notation in Definition 4, we have that $E\{\mathbf{\Delta}(i)\} = O\left(\frac{\log n}{n}\right), i = -1, 0, 1$; thus $E\{\mathbf{\Delta}(-1)\} + E\{\mathbf{\Delta}(0)\} + E\{\mathbf{\Delta}(1)\} = O\left(\frac{\log n}{n}\right)$. Therefore, by choosing n large enough we can make the fraction of 1-local transitions in NLS as small as we like.

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 1. 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

Table 1. Transition distribution table

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

```
const n : 1000;

type state : 1..n;

var x : state;

startstate
 x := 1;
end;

ruleset i : state do
 rule "go"
  (x >= i - 1) ==>
  begin
   x := i;
  end
end;

invariant true;
```

Fig. 2. A simple nonlocal system

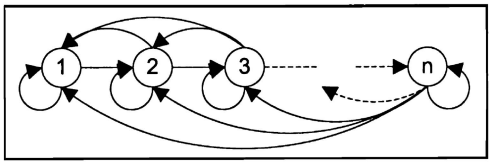

Fig. 3. Transition graph for the system in Fig. 2

a result would not imply anything about protocols in particular.

For example, for OBDDs we have already a similar situation (in *reverse mode*). In fact, in [22] it is proved that, for almost any boolean function f of n variables, the smallest 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.

4 Cache-based BF state space exploration

In this section we give an algorithm that takes advantage of the transition locality (Sect. 3) of protocols. Note that the correctness of our algorithm does not depend on the results in Sect. 3. However, such results help us to understand why the proposed algorithm is effective on protocol-like systems.

We modify the standard (hash-table-based) BF search to use a *cache memory* rather than a hash table. This means that we do not perform any collision check: when a state s' is hashed to an entry already holding a state s, we replace s with s', thus *forgetting* about s. In the following discussion, we shall call this algorithm CBFS (*cache-based breadth-first search*).

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 obstacles:

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

For this reason, unlike previous work on the use of auxiliary storage in state space exploration (e.g., in [34]), 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 and, since a queue has a high *locality of reference* (LOR) (whereas a hash table has a very poor LOR), the time overhead due to disk access is minimal. We point out that, working independently, Steven German at IBM [43] has also extended the queue implementation of the original Murφ verifier so as to use disk storage for the queue.

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

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 (*forgotten*) 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* with respect to the size of the state space, locality cannot help us.

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

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

4.1 From BFS to CBFS

To take advantage of transition locality we modify the standard BF search of Fig. 1 as shown in Fig. 4.

We use a cache memory rather than a hash table and we *guard* the main search loop with a check on the collision rate (i.e., the ratio <number of collisions in cache> / <number of insertions in cache>) since, as was already stated, using a cache memory may lead to nontermination. Indeed, when the collision rate is close to 1, nearly every state stored in the cache overwrites another state, so we are forgetting too much to complete the search.

```
FIFO_Queue Q;
Cache T;

collision_rate = 0.0;
bfs(init_states, next)
{
 /* load Q with initial states */
 foreach s in init_states Enqueue(Q, s);
 /* mark init states as visited */
 foreach s in init_states Insert(T, s);
 while ((Q is not empty) and
        (collision_rate <= 0.9))
 {
  s = Dequeue(Q);
  foreach s' in next(s)
   if (s' is not in T)
   {
    Insert(T, s');
    Enqueue(Q, s');
   }
 }
}
```

Fig. 4. Cache-based breadth-first search (CBFS)

Moreover, we implement our queue using auxiliary memory. In the present implementation we use a disk as auxiliary memory; thus, since disks are quite large, our disk queue approximates a potentially infinite queue. Note, however, that (the memory of) another workstation could have been used as well.

The implementation schema for the cache memory is quite standard. We reuse the open-addressed hash table and hash function implemented in Murφ to store the states. However, we limit the length of the collision chain so that if no free slot is found within `MAX_HT_CHAIN_LENGTH` steps, a random slot within the chain is selected and overwritten with the new state.

Figure 5 shows our implementation schema for the queue on disk. We split the queue `Q` into two segments: *head queue* and *tail queue*; both queues are in memory and each 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`.

When the head queue is empty, it gets reloaded with states from `swapin_file`; if this file is also empty, we swap `swapin_file` and `swapout_file` and try to load states from `swapin_file` again; if it is still empty, we swap the head queue with the tail queue.

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

4.2 Integrating CBFS within Murφ

Rather than building a tool from scratch, we decided to integrate our algorithm into the Murφ verifier. This gives us at least two advantages. First, we have immediately

```
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. 5. Disk queue functions

available many benchmark systems for testing. Second, we can exploit other memory reduction techniques that have already been implemented. We call CMurφ (*Cached Murφ*) [5] the resulting cache-based Murφ verifier.

To be consistent with the *standard* (hash-table-based) Murφ verifier, the value of `ram_queue_size` (Fig. 5) is such that the head queue and the tail queue together take the same amount of memory as the queue in standard Murφ. That is, 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 (`gPercentActive` is in $[0, 1]$ and is the Murφ constant defining the fraction of memory M used for the queue).

Integrating our algorithm into the Murφ verifier requires some care and consideration. As far as we are concerned, Murφ BF search can have two behaviors: one when hash compaction is not used and another when it is. 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, to save memory, Murφ stores states in the hash table and pointers to hash table slots (states) in the queue, whereas when hash compaction is used, it stores state signatures in the hash table and states in the queue.

Therefore, if we use a cache rather than a hash table, when hash compaction is not active and we overwrite a cache slot (collision), as a side effect we may also change the content of the queue, since that slot may be pointed to by the queue.

For this reason we modified the queue so that it holds states also when hash compaction is not used; all Murφ functions depending on this fact have been changed accordingly.

Of course, storing states rather than pointers (to states) 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 lead to any serious problem in our context.

Note that with our approach all optimization strategies implemented within the Murφ verifier (bit compression, hash compaction, symmetry reduction, etc.) are also available to us. Moreover, the bound on the omission probability for the hash compaction computed "online" by Murφ according to [36] 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.

5 Disk-based BF state space exploration

Disk read/write times are much larger than memory read/write times. Thus, not surprisingly, the main drawback of DBFS (*disk-based breadth-first search*) with respect to 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 obstacles to automatic verification; thus using disks to increase the amount of memory available during verification is very appealing.

In [34] a DBFS algorithm has been proposed for the Murφ verifier. Here we show how to improve that algorithm by exploiting *transition locality* (Sect. 3). In particular, disk accesses for reading can be reduced, leading to a reduction in the time overhead due to disk usage. In the following we call LDBFS our *l*ocality-based DBFS algorithm.

As in [34] we actually have two DBFS algorithms: one for the case in which *hash compaction* is enabled and one for the case in which it is not enabled. In the following discussion we only present the most interesting version that is compatible with the hash compaction option; the other version is simpler and easy to derive from the first.

We call DMurφ the resulting disk-based Murφ verifier.

5.1 Data structures

The data structures used by LDBFS are in Fig. 6 and are essentially the same as the ones used in [34]. 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*), and 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`.

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` that contains the set of state candidates to be in 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 reinserting in `Q_unck` recently visited states. Indeed, 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 [34] is that in the checking process we only use

```
/* main memory table */
HashTable M;
/* disk table */
FILE D;
/* checked state queue */
FIFO_Queue Q_ck;
/* unchecked state queue */
FIFO_Queue Q_unck;
/* number of blocks to be read from D */
int disk_cloud_size;
```

Fig. 6. DMurφ global data structures

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. This decreases disk usage and thus speeds up verification; however, with this approach Q_ck may also contain some *old* state and, 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* (Sect. 3), this does not happen too often. It is exactly this statistical property of transition graphs that makes our approach effective.

The global variable disk_cloud_size holds the number of blocks of D we use to remove old state signatures from table M; our algorithm *dynamically* adjusts the value of disk_cloud_size during the search.

Table M is in memory, whereas file D is on disk; we use a disk also for the BFS queues Q_ck, Q_unck. Our low-level algorithm to handle disk queues Q_ck and Q_unck is exactly the same one we used in CMurφ (Sect. 4).

Note that all the data structures that grow with the state space size (namely, D, Q_ck, Q_unck) are on disk in LDBFS, whereas in [34] the state queues are in memory; since states in the BFS queue are not compressed [23], for large verification problems the BFS queue can be a limiting factor for [34].

5.2 The main LDBFS loop

The Search() function (Fig. 7) is a *breadth-first search* using the checked queue Q_ck as the *current level* state queue. It first loads the BFS queue (Q_ck) with the initial states and then begins dequeuing states from Q_ck; for each successor s' of each state dequeued, Search() calls Insert(s') to store potentially new states in M as well as in Q_unck.

Given a state s, Insert(s) (Fig. 8) computes the signature h of s and, if h is not in table M, it inserts the 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.

When Q_ck becomes empty, it means that all transitions from all states in the current BFS level have been explored; thus we have 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 there are 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.

5.3 Exploiting locality in state filtering

The Checktable() function (Fig. 10), using disk file D, removes signatures of old (visited) states from table M.

```
Search()
{
 /* initialization */
 M = empty; D = empty;
 Q_ck = empty; Q_unck = empty;
 /* startstate generation */
 foreach s in init_states Insert(s);

 do   /* search loop */
 {
  while (Q_ck is not empty)
  {
   s = Dequeue(Q_ck);
   foreach s' in next(s)
    Insert(s');
  }  /* while */
  Checktable();
 } while (Q_ck is not empty); /* do */
} /* Search()*/
```

Fig. 7. Search() function

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

Fig. 8. Insert() function

Then, using this cleaned M, it 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.

In [34], this function uses *all* the states' signatures in D, whereas exploiting locality (Sect. 3) here we are able to use only a fraction of these signatures: by reading fewer states from disk, we save, with respect to [34], some of the time overhead due to disk accesses.

The rationale of our approach stems from the following observations. First, we should note that state signatures are appended to the file D in the same order in which new states are discovered by the BFS; thus, as we move toward the *tail* of the file we find (signatures of) states whose BFS level is closer and closer to the *current* level reached by the BFS.

From Sect. 3 we 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.

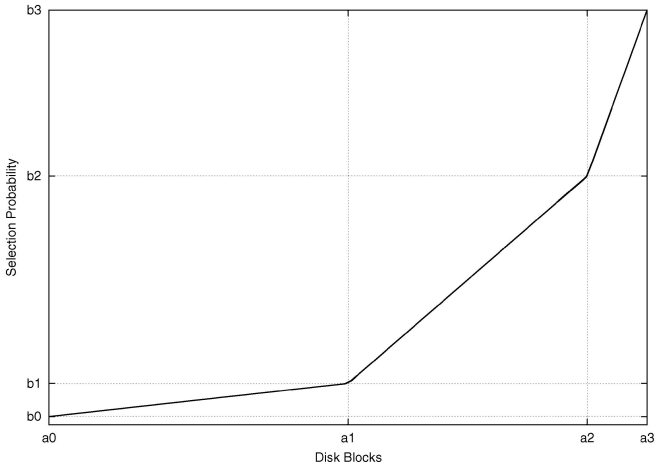

Fig. 9. Probability curve for disk cloud block selection (used by `GetDiskCloud()`)

We can take advantage of the above remarks by using the following approach. We divide the disk file `D` into *blocks*; then, rather than using the whole file `D` in the `Checktable()`, we only use a subset of the set of disk blocks, which we call *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, the selection probability increases as we approach the tail of `D` (Fig. 9).

Our experiments on CMurφ (Sect. 6.2) show that locality allows us to save about 40% of the memory required to complete verification. This suggests that it is necessary to use, say, just 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 fewer than 60% of the disk blocks to carry out the cleanup implemented by function `Checktable()`. Thus we *dynamically* adjust the fraction of disk blocks used by function `Checktable()`.

5.4 Disk cloud creation

The `GetDiskCloud()` function (Fig. 11) is called by the `Checktable()` function to create our disk cloud. The function selects `disk_cloud_size` disk blocks according to the probability curve shown in Fig. 9 and returns to `Checktable()` the indexes of the selected blocks.

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. 9 we have the relative disk block index ρ, i.e., $\rho =$<block index> / <number of blocks>. For instance, $\rho = 0$ is the (relative index of the) first (oldest) disk block inserted into disk `D`, whereas $\rho = 1$ is the last (newest) disk block inserted. On the y-axis of Fig. 9 we have the probability of selecting a disk block with a given ρ.

The probability curve is split into three segments. States closer to the current one must have a high selection probability, whereas states that are *not too far* from the current one have a medium selection probability. Note that, defensively, the selection probability of old blocks (ρ close to 0) must always be greater than zero. 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* nonlocal transitions. This kind of selection probability curve ensures that the most recently created blocks (ρ close to 1) are selected with a higher probability than old blocks, thus exploiting transition locality.

Since our min and max values for the relative disk block indexes are, respectively, 0 and 1, in Fig. 9 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 of the selection probabilities over all disk blocks is 1. The pairs (a_1, b_1), (a_2, b_2) define our selection strategy.

We tested many sets of values for these parameters, and those that obtained the best results in our experiments are: $a_1 = 0.4$, $b_1 = 0.4/K$, $a_2 = 0.7$, $b_2 = 0.6/K$. Of course, these values can be further refined, or they can be dynamically adjusted together with the disk cloud size.

Two strategies are possible to partition disk `D` into 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, so that the structure stored in disk `D` has a high *locality of reference*. 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.

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

5.5 Disk cloud size calibration

The `Calibrate()` function (Fig. 12) is called by the `Checktable()` function every time a calibration is needed for the disk cloud size.

Two parameters are passed to the `Calibrate()` function: the number of disk states deleted from `M` by `Checktable()` by only using disk blocks that are in the disk cloud (`deleted_in_cloud`) 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`).

```
Checktable() /* old/new check for main memory table */
{
 /* Disk cloud defined in Sect. 5.3 */

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

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

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

 if (there exists a disk block not selected in DiskCloud) {
  something_not_in_cloud = true;
 } else {
  something_not_in_cloud = false;
 }

 Calibration_Required = QueryCalibration();

 foreach Block in D {
  if (Block is in DiskCloud or Calibration_Required) {
   foreach state signature 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++;
     } } } } } /* foreach Block */

 /* remove old states from unchecked states queue (Q_unck) and add new
    states to disk and checked states queue (Q_ck) */
 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);
  } } /* while (Q_unck is not empty) */

 /* 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. 10. `Checktable()` function (state filtering)

```
GetDiskCloud()
{
 Randomly select disk_cloud_size blocks
 from disk, according to the probability
 distribution shown in Fig. 9;

 Return the indexes of the selected blocks;
}
```

Fig. 11. `GetDiskCloud()` function

```
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. 12. `Calibrate()` function

The function computes the ratio `beta` between the number of states deleted from `M` that are not in the disk cloud and the total number of states deleted from `M` (`deleted_states`). 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` 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 we used `speedup` $= 0.15$ and `slowdown` $= 0.15$.

5.6 Calibration frequency

The `QueryCalibration()` function tells `Checktable()` whether a calibration will be performed; that is, whether, at the end of the `Checktable()` function, `Calibrate()` will be called (Fig. 10). If a calibration will be performed, the `Checktable()` function reads the *whole* file `D` rather than just the disk cloud.

The rationale behind the `QueryCalibration()` function is the following. Calling the `Calibrate()` function *too often* nullifies our efforts for reducing disk usage, since a calibration of the disk cloud size requires reading the *whole* file `D`. However, calling `Calibrate()` *too sporadically* may have the same effect: in fact, waiting too long for a calibration may lead to the use of 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 this will lead to revisiting many already visited states.

In our current implementation, the `QueryCalibration()` function enables a calibration every γ calls of the `Checktable()` function. We tried many different values for γ; our experimental results suggest that 10 is a reasonable *calibration frequency*. A variable calibration frequency could make our algorithm more adaptive. However, the underlying heuristics should be chosen with extreme care. Indeed, our preliminary experiments showed that even a small change in the calibration frequency can produce a big degradation in the algorithm performances. This would be an interesting issue to investigate.

6 Experimental results

We report the experimental results we obtained using CMurφ (Sect. 4) and DMurφ (Sect. 5). We want to measure how much memory we can save by using our cache-based and disk-based approaches.

Our benchmark consists of some of the protocols in the Murφ distribution [23] and the `kerb` protocol from [37]. For some of these protocols we used several different versions with different parameter sets.

To make the results from all protocols comparable and machine independent, we will first determine the best performance of standard Murφ on each protocol and then give our experimental results as ratios with respect to the standard Murφ results.

6.1 Murφ measures

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 [23]).

Let M be the amount of memory and g (in $[0, 1]$) the fraction of M used for the queue (g is `gPercentActive` using a Murφ parlance). We say that the pair (M, g) is *suitable* for protocol p if and only if the verification of p can be completed with memory M and queue gM. For each protocol p we determine the least M so that for some

g, (M, g) is suitable for p. In the following discussion we denote by $M(p)$ such an M.

Of course, $M(p)$ depends on the compression options one uses. We determined $M(p)$ when only bit compression is used (option `-b`) and when bit compression and hash compaction are used (option `-b -c`).

The results are shown in Tables 3 and 4. The meaning of the columns is shown in Table 2. Table 3 shows the original protocols from our benchmark, whereas Table 4 shows our results for some bigger protocols obtained from those in Table 3 by modifying some of their parameters. For such protocols, we only present results about experiments in which all compression options are enabled.

We may note that there are protocols requiring more than 512 MB of memory to complete. Thus we could not use standard Murφ on our 512-MB PC. However, we were able to complete verification of such protocols using CMurφ (Sect. 4). In fact, when we give it enough memory, we get a very low *collision rate*, and from Sect. 6.2 we know that in this case the CPU time taken by CMurφ is essentially the same as that taken by standard Murφ with enough memory to complete the verification task. Of course, these protocols will not be used in the CBFS experiments but only to test the LDBFS algorithm in Sect. 6.4.

6.2 Results for CBFS

To test CBFS performances, we run each protocol p using our cache-based Murφ with the value of g (`gPercentActive`) chosen as in Table 3 and decreasing fractions of the minimal memory determined in Sect. 6.1. 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 [18], our memory 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 that obtained with a filled up memory.

The results using bit compression (option `-b`) are in Table 5, whereas the results obtained using bit compression and hash compaction (options `-b -c`) are in Table 6. Note that the tables do not show the results for all the protocols in our benchmark set but only the most representative cases (including the best and worst cases).

In these tables, the memory fraction used for the experiment is shown as a real number α, with $\alpha = 1, 0.9, \ldots, 0.1$, on the top of each column; thus, e.g., column 0.5 gives information about the run of protocol p with memory $0.5M(p)$ (half of the minimal memory needed by standard Murφ).

Table 2. Meaning of columns in Tables 3 and 4

Attribute	Meaning
Model	Name of the model.
Parameters	If the model parameters have been changed, we report our parameter values in the same order in which they appear in the `Const` section of the model file. When this list is too long, we just list the modified assignments.
Bytes per state	Number of bytes needed to represent a state in the queue when bit compression is used. For model p we denote this number by $\mathbf{StateBytes}(p)$. When bit compression and hash compaction are used, this number is always 5 bytes (signature size).
Reachable states	Number of reachable states for the model. For model p, we denote this number by $\lvert\mathbf{Reach}(p)\rvert$.
Rules fired	Number of rules fired during state space exploration. Each rule represents a transition in the model transition graph. For model p, we denote this number by $\mathbf{RulesFired}(p)$.
Max queue	Maximum queue size (i.e., number of states) attained during space state exploration. For model p we denote this number by $\mathbf{MaxQ}(p)$.
Diam	Diameter of the model transition graph.
Mem	Minimum amount of memory (in kilobytes) needed to complete state space exploration, denoted by $M(p)$. Let b_h be the number of bytes taken by a state in the hash table ($b_h = 5$ if hash compaction is used). From the Murφ source code we can compute $M(p)$. We have: $M(p) = \lvert\mathbf{Reach}(p)\rvert\ (b_h + (\mathbf{MaxQ}(p)/\lvert\mathbf{Reach}(p)\rvert)\mathbf{StateBytes}(p))$.
g	Fraction of memory used for the queue. From the Murφ source code we can compute g. We have: $g = \mathbf{MaxQ}(p)/\lvert\mathbf{Reach}(p)\rvert$.
Time	CPU time (in seconds) to complete state space exploration using the given parameters and Murφ options. We denote this number by $T(p)$.

Table 3. Murφ results on a Linux Pentium III 866-MHz PC with 256 MB RAM using bit compression and bit compression + 40-bit hash compaction. All experiments have been carried out with deadlock detection disabled

						Bit compression			Bit Compression & Hash compaction		
Model	Bytes per state	Reachable states	Rules fired	Max queue	Diam	Mem	g	Time	Mem	g	Time
cache3	12	577	2440	102	16	10	0.4	0.15	4	0.4	0.15
list6too	20	1077	11 622	64	36	26	0.2	15.55	6	0.1	16.83
arbiter	8	1103	2365	301	12	15	0.4	0.1	7	0.3	0.1
ns	24	980	2314	382	11	29	0.4	0.59	7	0.4	0.62
ns-old	24	1121	2578	424	11	33	0.4	0.69	8	0.4	0.69
adashbug	144	3742	39 619	646	14	544	0.2	18.19	21	0.18125	17.60
newcache3	52	4357	20 201	462	27	240	0.1125	8.01	24	0.15	7.93
adash	144	10 466	137 708	734	37	1516	0.075	60.57	55	0.075	62.98
newlist6	24	13 044	53 595	631	53	360	0.05	17.23	67	0.05	18.34
cache3multi	28	13 738	65 357	1229	29	435	0.1	34.16	73	0.1	35.11
sci	56	18 193	60 455	1175	62	1071	0.066	27.29	94	0.06875	28.17
list6	24	23 410	99 874	1095	53	645	0.05	13.73	119	0.05	14.90
mcslock1	12	23 644	94 576	928	69	373	0.04023	16.17	120	0.04375	16.76
sym.cache3	28	31 433	264 758	2877	32	994	0.1039	36.22	167	0.10625	36.03
kerb	80	109 282	172 111	20 523	19	9046	0.191	291.49	615	0.190625	301.86
eadash	376	133 491	1 786 047	9050	47	49 571	0.06875	4102.33	688	0.06875	4114.71
n_peterson	16	163 298	1 143 086	3775	145	3204	0.0233	269.13	813	0.0233	273.32
ldash	144	254 986	2 647 358	14 988	64	36 930	0.075	4002.97	1307	0.062	3950.50
mcslock2	12	540 219	1 620 657	15 655	111	8503	0.0293	234.68	2693	0.02969	237.48

Table 4. Murφ results on a 2-GHz INTEL Pentium IV Linux PC with 512 MB RAM using bit compression + 40-bit hash compaction. All experiments have been carried out with deadlock detection disabled

							Bit compression & Hash compaction		
Model	Parameters	Bytes per state	Reachable states	Rules fired	Max queue	Diam	Mem	g	Time
n_peterson	9	20	2 871 372	25 842 348	46 657	241	14 932	0.02	764.27
newlist6	7	32	3 619 556	21 612 905	140 382	91	22 060	0.04	1641.67
sci	3,1,1,2,1	60	9 299 127	30 037 227	347 299	94	65 755	0.04	2852.03
mcslock1	6	16	12 783 541	76 701 246	392 757	111	68 556	0.03	3279.45
ldash	1,4,1,false	144	8 939 558	112 808 653	509 751	72	115 334	0.06	12 352.93
ns	1,1,3,2,10	96	2 455 257	8 477 970	1 388 415	12	142 152	0.57	1211.02
newlist6	8	40	81 271 421	563 937 480	2 875 471	110	509 156	0.03	31 114.87
sci	3,1,1,5,1	64	75 081 011	254 261 319	2 927 550	95	549 578	0.04	35 904.86
kerb	NumIntruders=2	148	7 614 392	9 859 187	4 730 277	15	720 852	0.62	2830.83
sci	3,1,1,7,1	68	126 784 943	447 583 731	4 720 612	143	932 545	0.04	99 904.47

Rows **States**, **Rules**, and **Time** give the ratio between the visited states, the rules fired, and the time used to complete the state space exploration with CMurφ using the given memory fraction and the same values obtained with standard Murφ with the same compression options, as determined in Sect. 6.1.

Row **Coll** gives the collision rate, that is, the ratio between the number of collisions and the number of states inserted in the cache. Since our state space exploration may not terminate, we stop our visit when the collision rate is greater than 0.9. We report also the information on these experiments to give an idea of the behavior of our approach when a visit is stopped prematurely.

In this case, we do not know a priori if all reachable states have been visited. However, in our experiments we have such information since all the protocols can actually fit in our memory. We mark with a $*$ superscript the data obtained when the visit has been stopped because the collision rate exceeded 0.9 and all reachable states have been visited. We mark with a ∞ superscript the data obtained when the visit

Table 5. Cache-based BF search with bit compression

Model	Memory fraction	1	0.9	0.8	0.7	0.6	0.5	0.4	0.3	0.2	0.1
kerb	states	1.001	1.001	1.003	1.007	1.015	1.025	1.048	1.097	1.228	1.107^{∞}
	rules	1.003	1.005	1.012	1.022	1.038	1.057	1.092	1.151	1.285	1.012^{∞}
	time	0.982	0.986	0.987	0.994	1.004	1.015	1.041	1.090	1.221	0.828^{∞}
	coll	0.060	0.122	0.207	0.305	0.409	0.512	0.618	0.727	0.837	0.910^{∞}
ns	states	1.001	1.000	1.010	1.046	1.024	1.057	1.147	2.476	2.041^{*}	1.020^{∞}
	rules	1.032	1.000	1.066	1.140	1.047	1.128	1.284	2.998	1.974^{*}	0.891^{∞}
	time	1.085	1.068	1.102	1.169	1.119	1.169	1.305	2.864	1.746^{*}	0.746^{∞}
	coll	0.042	0.098	0.202	0.326	0.409	0.530	0.659	0.886	0.919^{*}	0.932^{∞}
sci	states	1.003	1.008	1.020	1.041	1.095	1.369	4.067^{*}	3.243^{∞}	2.089^{∞}	1.099^{∞}
	rules	1.003	1.009	1.022	1.045	1.102	1.382	3.975^{*}	2.919^{∞}	1.749^{∞}	0.850^{∞}
	time	1.017	1.016	1.025	1.048	1.104	1.392	3.970^{*}	2.912^{∞}	1.752^{∞}	0.858^{∞}
	coll	0.062	0.127	0.219	0.328	0.453	0.635	0.902^{*}	0.908^{∞}	0.906^{∞}	0.912^{∞}
mcslock2	states	1.009	1.017	1.032	1.063	1.178	2.751^{*}	4.008^{∞}	3.004^{∞}	2.269^{∞}	1.061^{∞}
	rules	1.009	1.017	1.032	1.063	1.178	2.749^{*}	3.742^{∞}	2.710^{∞}	1.986^{∞}	0.903^{∞}
	time	0.978	0.991	1.009	1.046	1.163	2.734^{*}	3.721^{∞}	2.692^{∞}	1.951^{∞}	0.879^{∞}
	coll	0.092	0.152	0.236	0.343	0.491	0.818^{*}	0.900^{∞}	0.900^{∞}	0.912^{∞}	0.906^{∞}
mcslock1	states	1.009	1.020	1.058	1.190	1.900	4.948^{*}	4.187^{∞}	3.299^{∞}	2.368^{∞}	1.100^{∞}
	rules	1.009	1.020	1.058	1.190	1.900	4.613^{*}	3.709^{∞}	2.818^{∞}	1.958^{∞}	0.895^{∞}
	time	1.015	1.030	1.074	1.198	1.890	4.547^{*}	3.605^{∞}	2.678^{∞}	1.799^{∞}	0.798^{∞}
	coll	0.069	0.140	0.248	0.412	0.685	0.900^{*}	0.904^{∞}	0.911^{∞}	0.917^{∞}	0.911^{∞}
newlist6	states	1.009	1.022	1.049	1.198	1.830	5.136^{*}	3.987^{∞}	3.527^{∞}	2.300^{∞}	1.227^{∞}
	rules	1.009	1.023	1.052	1.198	1.816	4.641^{*}	3.328^{∞}	2.783^{∞}	1.734^{∞}	0.888^{∞}
	time	1.005	1.020	1.052	1.198	1.810	4.656^{*}	3.320^{∞}	2.777^{∞}	1.725^{∞}	0.868^{∞}
	coll	0.066	0.134	0.239	0.415	0.671	0.903^{*}	0.900^{∞}	0.915^{∞}	0.914^{∞}	0.922^{∞}
adash	states	1.006	1.016	1.043	1.124	1.738	5.160^{*}	4.300^{∞}	3.631^{∞}	2.389^{∞}	1.147^{∞}
	rules	1.007	1.016	1.043	1.125	1.736	4.749^{*}	3.714^{∞}	2.937^{∞}	1.819^{∞}	0.782^{∞}
	time	1.001	1.010	1.039	1.124	1.747	4.824^{*}	3.792^{∞}	3.013^{∞}	1.889^{∞}	0.818^{∞}
	coll	0.063	0.132	0.236	0.377	0.655	0.904^{*}	0.907^{∞}	0.919^{∞}	0.916^{∞}	0.918^{∞}
newcache3	states	1.012	1.029	1.053	1.213	1.895	5.279^{*}	4.131^{∞}	2.984^{∞}	2.066^{∞}	1.148^{∞}
	rules	1.011	1.027	1.050	1.222	1.927	4.851^{*}	3.446^{∞}	2.378^{∞}	1.551^{∞}	0.754^{∞}
	time	1.004	1.022	1.049	1.230	1.953	4.905^{*}	3.487^{∞}	2.401^{∞}	1.577^{∞}	0.792^{∞}
	coll	0.069	0.141	0.242	0.423	0.683	0.907^{*}	0.907^{∞}	0.902^{∞}	0.911^{∞}	0.919^{∞}
cache3multi	states	1.009	1.030	1.098	1.291	2.422	5.387^{*}	4.367^{∞}	3.203^{∞}	2.184^{∞}	0.946^{∞}
	rules	1.011	1.037	1.117	1.328	2.535	5.173^{*}	3.902^{∞}	2.674^{∞}	1.725^{∞}	0.712^{∞}
	time	1.010	1.037	1.118	1.337	2.564	5.255^{*}	3.974^{∞}	2.730^{∞}	1.759^{∞}	0.728^{∞}
	coll	0.062	0.141	0.272	0.458	0.752	0.908^{*}	0.909^{∞}	0.908^{∞}	0.911^{∞}	0.900^{∞}
n_peterson	states	1.004	1.011	1.028	1.242	4.366^{*}	5.028^{*}	4.036^{*}	3.037^{∞}	2.039^{∞}	1.035^{∞}
	fired	1.004	1.011	1.028	1.242	4.364^{*}	4.994^{*}	3.997^{*}	2.992^{∞}	1.992^{∞}	0.993^{∞}
	time	0.972	0.981	1.004	1.201	4.222^{*}	4.888^{*}	3.914^{*}	2.959^{∞}	1.975^{∞}	1.017^{∞}
	coll	0.062	0.128	0.225	0.437	0.863^{*}	0.901^{*}	0.901^{*}	0.901^{∞}	0.902^{∞}	0.904^{∞}
ldash	states	1.019	1.058	1.295	7.495^{*}	6.114^{∞}	6.106^{∞}	4.067^{∞}	3.753^{∞}	2.361^{∞}	1.020^{∞}
	rules	1.018	1.056	1.287	6.698^{*}	5.084^{∞}	4.847^{∞}	3.086^{∞}	2.726^{∞}	1.633^{∞}	0.664^{∞}
	time	1.043	1.078	1.324	6.758^{*}	5.112^{∞}	4.984^{∞}	3.177^{∞}	2.828^{∞}	1.703^{∞}	0.701^{∞}
	coll	0.069	0.161	0.382	0.907^{*}	0.902^{∞}	0.918^{∞}	0.902^{∞}	0.920^{∞}	0.915^{∞}	0.902^{∞}
Min	Time	0.972	0.981	0.987	0.994	1.004	1.015	1.041	1.090	1.221	//
Avg	Time	1.010	1.022	1.071	1.155	1.595	1.192	1.173	1.977	1.221	//
Max	Time	1.085	1.078	1.324	1.337	2.564	1.392	1.305	2.864	1.221	//

Table 6. Cache-based BF search with bit compression and hash compaction

Model	Memory fraction	1	0.9	0.8	0.7	0.6	0.5	0.4	0.3	0.2	0.1
`kerb`	states	1.000	1.001	1.002	1.006	1.013	1.026	1.047	1.093	1.223	1.116^{∞}
	rules	1.000	1.003	1.008	1.018	1.034	1.060	1.090	1.146	1.280	1.020^{∞}
	time	0.983	0.990	1.002	1.011	1.026	1.054	1.078	1.135	1.274	0.891^{∞}
	coll	0.007	0.101	0.202	0.305	0.407	0.513	0.618	0.726	0.836	0.911^{∞}
`sci`	states	1.000	1.005	1.013	1.037	1.087	1.313	4.013^{*}	3.133^{∞}	2.309^{∞}	1.154^{∞}
	rules	1.000	1.006	1.014	1.041	1.093	1.325	3.950^{*}	2.793^{∞}	1.916^{∞}	0.894^{∞}
	time	1.027	1.043	1.061	1.105	1.180	1.458	4.544^{*}	3.225^{∞}	2.226^{∞}	1.047^{∞}
	coll	0.005	0.108	0.210	0.331	0.450	0.618	0.902^{*}	0.906^{∞}	0.917^{∞}	0.920^{∞}
`adash`	states	1.000	1.009	1.025	1.133	1.565	5.064^{*}	4.491^{∞}	3.631^{∞}	2.293^{∞}	0.860^{∞}
	rules	1.000	1.009	1.025	1.133	1.568	4.684^{*}	3.895^{∞}	2.931^{∞}	1.763^{∞}	0.572^{∞}
	time	0.973	0.994	1.016	1.134	1.593	4.855^{*}	4.046^{∞}	3.081^{∞}	1.863^{∞}	0.609^{∞}
	coll	0.002	0.103	0.207	0.380	0.611	0.902^{*}	0.911^{∞}	0.918^{∞}	0.912^{∞}	0.900^{∞}
`cache3multi`	states	1.000	1.013	1.089	1.297	2.939	5.168^{*}	4.440^{∞}	3.421^{∞}	2.329^{∞}	0.946^{∞}
	rules	1.000	1.017	1.104	1.330	3.078	4.929^{*}	3.985^{∞}	2.862^{∞}	1.835^{∞}	0.713^{∞}
	time	0.978	1.005	1.107	1.363	3.344	5.418^{*}	4.399^{∞}	3.157^{∞}	2.026^{∞}	0.780^{∞}
	coll	0.003	0.114	0.264	0.457	0.798	0.905^{*}	0.911^{∞}	0.915^{∞}	0.918^{∞}	0.903^{∞}
`newcache3`	states	1.000	1.022	1.053	1.313	2.155	5.049^{*}	4.131^{∞}	3.213^{∞}	1.836^{∞}	1.148^{∞}
	rules	1.000	1.020	1.052	1.330	2.192	4.646^{*}	3.412^{∞}	2.564^{∞}	1.347^{∞}	0.742^{∞}
	time	1.032	1.061	1.111	1.450	2.483	5.462^{*}	4.026^{∞}	3.021^{∞}	1.604^{∞}	0.919^{∞}
	coll	0.003	0.136	0.239	0.488	0.725	0.903^{*}	0.908^{∞}	0.911^{∞}	0.909^{∞}	0.936^{∞}
`newlist6`	states	1.000	1.007	1.050	1.226	1.897	5.443^{*}	4.140^{∞}	3.373^{∞}	2.300^{∞}	1.073^{∞}
	rules	1.000	1.008	1.053	1.224	1.882	4.863^{*}	3.434^{∞}	2.674^{∞}	1.738^{∞}	0.776^{∞}
	time	1.046	1.066	1.140	1.360	2.192	6.009^{*}	4.263^{∞}	3.335^{∞}	2.155^{∞}	0.944^{∞}
	coll	0.002	0.100	0.237	0.434	0.682	0.909^{*}	0.905^{∞}	0.912^{∞}	0.917^{∞}	0.917^{∞}
`mcslock1`	states	1.000	1.010	1.046	1.163	1.848	5.160^{*}	4.145^{∞}	3.341^{∞}	2.326^{∞}	1.100^{∞}
	rules	1.000	1.010	1.046	1.163	1.848	4.817^{*}	3.680^{∞}	2.849^{∞}	1.926^{∞}	0.893^{∞}
	time	0.982	1.025	1.106	1.284	2.217	6.304^{*}	4.776^{∞}	3.652^{∞}	2.410^{∞}	1.085^{∞}
	coll	0.006	0.104	0.232	0.395	0.674	0.903^{*}	0.903^{∞}	0.911^{∞}	0.915^{∞}	0.910^{∞}
`mcslock2`	states	1.000	1.006	1.022	1.055	1.177	4.444^{*}	4.206^{∞}	3.038^{∞}	2.279^{∞}	1.063^{∞}
	rules	1.000	1.006	1.022	1.055	1.177	4.427^{*}	3.915^{∞}	2.734^{∞}	1.993^{∞}	0.905^{∞}
	time	1.023	1.092	1.181	1.301	1.538	6.956^{*}	6.202^{∞}	4.325^{∞}	3.043^{∞}	1.335^{∞}
	coll	0.009	0.105	0.217	0.337	0.490	0.888^{*}	0.905^{∞}	0.901^{∞}	0.912^{∞}	0.906^{∞}
`n_peterson`	states	1.000	1.005	1.024	1.225	4.262^{*}	5.021^{*}	4.036^{*}	2.988^{∞}	1.990^{∞}	0.998^{∞}
	rules	1.000	1.005	1.024	1.225	4.262^{*}	4.988^{*}	3.996^{*}	2.942^{∞}	1.944^{∞}	0.957^{∞}
	time	1.008	1.033	1.079	1.320	5.014^{*}	5.993^{*}	4.797^{*}	3.560^{∞}	2.374^{∞}	1.207^{∞}
	coll	0.006	0.104	0.219	0.428	0.859^{*}	0.901^{*}	0.901^{*}	0.900^{∞}	0.900^{∞}	0.900^{∞}
`ldash`	states	1.000	1.028	1.284	7.189^{*}	7.047^{∞}	5.953^{∞}	4.016^{∞}	3.722^{∞}	2.349^{∞}	1.016^{∞}
	rules	1.000	1.027	1.278	6.394^{*}	5.861^{∞}	4.719^{∞}	3.048^{∞}	2.705^{∞}	1.622^{∞}	0.062^{∞}
	time	1.013	1.075	1.346	6.851^{*}	6.177^{∞}	4.987^{∞}	3.278^{∞}	2.946^{∞}	1.762^{∞}	0.727^{∞}
	coll	0.007	0.125	0.377	0.903^{*}	0.915^{∞}	0.916^{∞}	0.901^{∞}	0.920^{∞}	0.915^{∞}	0.902^{∞}
Min	Time	0.973	0.990	1.002	1.011	1.026	1.054	1.078	1.135	1.274	//
Avg	Time	1.007	1.038	1.115	1.259	1.947	1.256	1.078	1.135	1.274	//
Max	Time	1.046	1.092	1.346	1.450	3.344	1.458	1.078	1.135	1.274	//

has been stopped because the collision rate exceeded 0.9 and there are reachable states that have not been visited.

The **Time** row shows the time overhead due to revisiting already visited states. Using only bit compression (Table 5) time penalty for column 0.6 (40% memory saving) ranges from 0 to 150% with an average value of about 50%. Using bit compression and hash compaction (Table 6) time penalty for column 0.6 ranges from 0 to 234% with an average value of about 100%.

Therefore, our experimental results show that our cache-based approach typically saves about 40% of memory with respect to a hash-table-based approach. This holds when using bit compression alone as well as when using bit compression and hash compaction.

The results of Table 6 are also plotted in Fig. 13, where we have the *Memory fraction* on the x-axis and *Time* on the y-axis.

Note that no memory saving may be possible for non-local protocols. For example, the state space exploration of the NLS system in Fig. 2 does not terminate unless we give the verifier enough memory to store all the reachable states.

6.3 A large protocol with CBFS

We also tested our cache-based approach with a large protocol that heavily loads our machine.

We used the `ns` protocol with parameters `NumInitiators` = 2, `NumResponders` = 1, `NumIntruders` = 2, `NetworkSize` = 2, `MaxKnowledge` = 10, giving to CMurφ 200, 150, and 130 MB of memory.

The results are in Table 7. Columns **Cache memory** and **Cache size** give, respectively, the memory reserved for the cache and the max number of states that fit in that memory (each state takes 40 bits with hash compaction).

Columns **Queue memory** and **Queue size** give, respectively, the memory reserved for the queue and the max number of states that fit in that memory (each state takes 68 bytes in the queue).

Columns **Max states queued** and **Max queue space used** give, respectively, the max number of states actually contained in the BF queue (memory + disk) during the verification and the space they would require in memory. The other columns have their usual meaning.

When **Memory** = 200 (first row of Table 7), the collision rate is low; thus the number of visited states is about the number of reachable states. This means that completing verification for this protocol would require more than 1313 MB of memory using standard Murφ (i.e., 200 MB for the hash table, 1113 MB for the queue). This is more than our machine can handle: however, using our cache-based Murφ, we were able to complete verification using only 130 MB of memory (last row of Table 7).

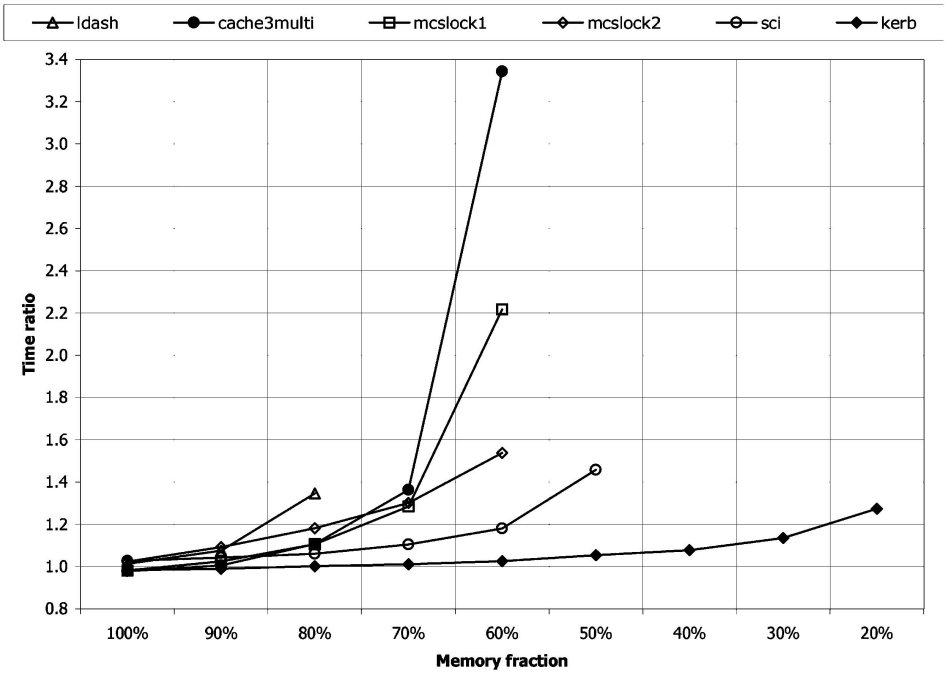

Fig. 13. *Time* (time ratio) vs. *Memory fraction* from Table 6

Table 7. Cache-based BF search for protocol `ns` with parameters: `NumInitiators` = 2, `NumResponders` = 1, `NumIntruders` = 2, `NetworkSize` = 2, `MaxKnowledge` = 10. Results obtained on a Pentium III 866-MHz machine with 256 MB RAM using bit compression, 40-bit hash compaction, and no deadlock detection

Memory (Mb)	Cache size	Cache memory (Mb)	Queue size	Queue memory (Mb)	Visited states	Time	Collision rate	Max states queued	Max queue space used (Mb)
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

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

6.4 Results for LDBFS

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 exploiting locality there is indeed some gain with respect to the algorithm proposed in [34]. Second, we want to measure DMurφ time overhead with respect to standard Murφ performing a RAM–BFS.

We proceed using the same method used for CBFS in Sect. 6.2: we run each protocol p with our disk-based Murφ, using decreasing fractions of the minimal memory determined in Sect. 6, i.e., we run protocol p with memory limits $M(p)$, $0.5M(p)$ and $0.1M(p)$.

The results we obtained comparing our disk-based Murφ with standard Murφ are in Table 8. Column meanings are the same as those in Sect. 6.2. Note that, with the disk-based algorithm, the time to complete the verification is the time elapsed between the start and the end of the state space exploration process: that is, it in-

Table 8. Comparing DMurφ with standard Murφ (using bit compression and hash compaction)

Model	Parameters	Memory fraction	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

cludes the CPU time as well as the time spent on disk accesses.

For the *biggest* models in our benchmark, requiring more than 512 MB of memory, we could not run the experiments with memory fraction $\alpha = 1$ on our machine with 512 MB of memory. However, the most interesting column for us is the one with $\alpha = 0.1$. Indeed, the experimental results 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 memory to complete the verification task.

The results of Table 8 are also plotted in Fig. 14, where we have the *Memory fraction* on the x-axis and *Time* (time fraction) on the y-axis.

6.5 DMurφ vs. disk-based Murφ

To measure the time speedup we obtain by exploiting locality, we are also interested in comparing our locality-based disk algorithm DMurφ with the disk-based Murφ presented in [34]. The latter algorithm is not available in the standard Murφ distribution, so we obtained it from DMurφ by omitting the calibration step in the `Checktable()` function and always using all disk blocks to clean up the unchecked queue `Q_unck` and table `M` (Sect. 5.3). In the following discussion, we will refer to this algorithm as disk-based Murφ.

Although we had to use our own implementation of the algorithm described in [34], we think that our comparison of it with our DMurφ is quite fair since our implementation of the algorithm in [34] shares all optimizations we implemented in DMurφ except the calibration step, which is what allows us to save disk accesses (and thus verification time) by exploiting transition locality.

For disk-based Murφ, we repeated the same set of experiments we ran for DMurφ in Sect. 6.4. However, the *biggest* models of Table 4 took too long. Thus we did not include them in our set of experiments.

The results are in Table 9. Computations taking too much longer than the time in Table 4 were aborted. In such cases, we get a lower bound to the time overhead with respect to standard Murφ. This is indicated with a $>$ sign before the lower bound. For these 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.

Finally, Table 10 compares performances of our DMurφ with those of the disk-based Murφ.

Of course, the interesting cases for us are those with $\alpha = 0.1$, when there is not enough memory to complete verification using a RAM–BFS. For such cases, from the results in Table 10 we see that our algorithm is typically more than ten times faster than the one presented in [34].

6.6 A large protocol with LDBFS

We also tested our disk-based approach on a protocol out of reach for both standard Murφ and CMurφ on our 512-MB machine.

We found that the protocol `mcslock2` (with `N` $= 4$) suited our needs. Our results are in Table 11.

Columns **Max states queued** and **Max queue space used** give, respectively, the max number of states actually contained in the BF queue (memory + disk)

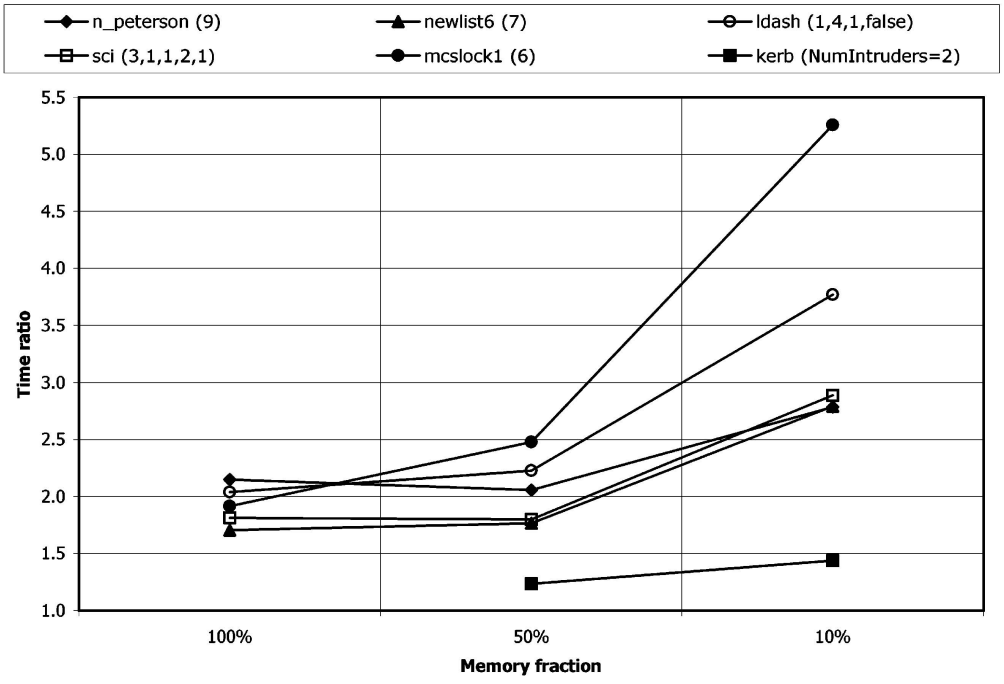

Fig. 14. *Time* (time ratio) vs. *Memory fraction* from Table 8

Table 9. Comparing disk-based Murφ with standard Murφ (using bit compression and hash compaction)

Model	Parameters	Memory fraction	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

Table 10. Comparing DMurφ with disk-based Murφ (using bit compression and hash compaction)

Model	Parameters	Memory fraction	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

during the verification and the space they would require in memory. Column **Hash table space used** gives the hash table size that would be needed if all reachable states were stored in a memory hash table. Column **Total space used** gives the memory (in kilobytes) needed to complete the verification task using a RAM–BFS with standard Murφ (that is, **Max queue space used** + **Hash table space used**). The other columns have their usual meaning.

7 Conclusions

We showed experimentally (Sect. 3) that protocols exhibit *transition locality*. We supported our claim by measuring transition locality for the set of protocols included in the Murφ verifier distribution.

We presented (Sect. 4) a (memory-based) *BF explicit state space exploration* algorithm to exploit transition locality and implemented it within the Murφ verifier. Essentially our algorithm replaces the hash table used in a BF state exploration with a fixed-size cache (without collision detection) and uses disk storage for the BF queue. Our experimental results (Sect. 6.2) show that, with respect to 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%.

We also presented (Sect. 5) a disk-based *breadth-first explicit state space exploration* algorithm as well as an

Table 11. Disk-based BF search for protocol `mcslock2` with parameter `N` = 4. Results obtained on a 1-GHz Pentium IV PC with 512 MB RAM using bit compression, 40-bit hash compaction, and no deadlock detection

Memory (Mb)	Visited states	Rules fired	Diam	Time	Bytes per state	Hash table space used (Kb)	Max states queued	Max queue space used (Kb)	Total space used (Kb)
300	945 950 806	3 783 803 224	153	406 275	16	4 729 754	30 091 568	481 465	5 211 219

implementation of it within the Murφ verifier. Our algorithm has been obtained from the one in [34] by exploiting *transition locality* to decrease disk usage (namely, disk read accesses). Our experimental results (Sect. 6.4) show that our algorithm is typically more than ten times faster than the disk-based algorithm proposed [34]. Moreover, even when using 1/10 of the memory needed to complete verification, our algorithm is only between 40 and 530% (300% on average) slower than RAM–BFS (namely, standard Murφ) with enough 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 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.

References

1. A user guide to hytech: `http://www.eecs.berkeley.edu/~tah/HyTech`
2. Alur R, Henzinger TA, Ho PH (1996) Automatic symbolic verification of embedded systems. IEEE Trans Softw Eng 22:2–11
3. Bryant R (1986) Graph-based algorithms for boolean function manipulation. IEEE Trans Comput C-35(8):677–691
4. Burch JR, Clarke EM, McMillan KL, Dill DL, Hwang LJ (1992) Symbolic model checking: 10^{20} states and beyond. Inf Comput 98(2):142–170
5. Cached murphi Web page: `http://www.dsi.uniroma1.it/~tronci/cached.murphi.html`
6. Chernikova NV (1968) Algorithm for discovering the set of all solutions of a linear programming problem. USSR Comput Math Math Phys 8(6):282–293
7. Dill DL, Drexler AJ, Hu AJ, Yang CH (1992) Protocol verification as a hardware design aid. In: Proceedings of the IEEE international conference on computer design: VLSI in computers and processors, pp 522–525
8. Godefroid P, Holzmann GJ, Pirottin D (1992) State space caching revisited. In: Bochmann GV, Probst D (eds) Proceedings of the 4th international workshop on computer aided verification (CAV), Montreal. Lecture notes in computer science, vol 663. Springer, Berlin Heidelberg New York, pp 178–191
9. Halbwachs N (1993) Delay analysis in synchronous programs. In: Courcoubetis C (ed) Proceedings of Computer Aided Verification (CAV). Lecture notes in computer science, vol 697. Springer, Berlin Heidelberg New York, pp 333–346
10. Halbwachs N, Raymond P, Proy Y-E (1994) Verification of linear hybrid systems by means of convex approximation. In: LeCharlier B (ed) Proceedings of the symposium on static analysis (SAS). Lecture notes in computer science, vol 864. Springer, Berlin Heidelberg New York, pp 223–237
11. Henzinger TA, Ho P-H, Wong-Toi H (1995) Hytech: the next generation. In: Proceedings of the 16th annual IEEE real-time systems symposium (RTSS). IEEE, New York, pp 56–65
12. Henzinger TA, Ho P-H, Wong-Toi H (1997) Hytech: a model checker for hybrid systems. Int J Softw Tools Technol Transfer 1:110–122
13. Holzmann GJ (1985) Tracing protocols. AT&T Tech J 64(10):2413–2433
14. Holzmann GJ (1987) Automated protocol validation in argos, assertion proving and scatter searching. IEEE Trans Softw Eng 13(6):683–697
15. Holzmann GJ (1997) The spin model checker. IEEE Trans Softw Eng 23(5):279–295
16. Holzmann GJ (1998) An analysis of bitstate hashing. Formal Methods Sys Des 13(3):289–307
17. Holzmann GJ, Peled D (1995) An improvement in formal verification. In: Proceedings of the FORTE conference, Proceedings of IFIP. Chapman & Hall, London, 6:197–211
18. Hu AJ, York G, Dill DL (1994) New techniques for efficient verification with implicitily conjoined bdds. In: Proceedings of the 31st IEEE conference on design automation, pp 276–282
19. Ip CN, Dill DL (1993) Better verification through symmetry. In: Agnew D, Claesen L, Camposano R (eds) Proceedings of the 11th international conference on: computer hardware description languages and their applications. Elsevier, Amsterdam, pp 97–111
20. Ip CN, Dill DL (1993) Efficient verification of symmetric concurrent systems. In: Proceedings of the IEEE international conference on computer design: VLSI in computers and processors, pp 230–234
21. Larsen KG, Pettersson P, Yi W (1997) Uppaal: Status and developments. In: Grumberg O (ed) Proceedings of CAV97, June 1997. Lecture notes in computer science, vol 1254. Springer, Berlin Heidelberg New York, pp 456–459
22. Liaw H-T, Lin C-S (1992) On the obdd-representation of general boolean functions. IEEE Trans Comput C-41(6):661–664
23. Murphi Web page: `http://sprout.stanford.edu/dill/murphi.html`
24. Papoulis A (1965) Probability, random variables and stochastic processes. McGraw-Hill Series in System Sciences
25. Patterson DA, Hennessy JL (1996) Computer architecture: a quantitative approach. Morgan Kaufmann, San Francisco
26. Della Penna G, Intrigila B, Melatti I, Minichino M, Ciancamerla E, Parisse A, Tronci E, Zilli MV (2003) Automatic verification of a turbogas control system with the murφ verifier. In: Pnueli A, Maler O (eds) Proceedings of the 6th international workshop Hybrid Systems: Computation and Control (HSCC), Prague, Czech Republic, April 2003. Lecture notes in computer science, vol 2623. Springer, Berlin Heidelberg New York, pp 141–155
27. Della Penna G, Intrigila B, Tronci E, Venturini Zilli M (2002) Exploiting transition locality in the disk based murφ verifier. In: Aagaard MD, O'Leary JW (eds) Proceedings of the 4th international conference on formal methods in computer aided design (FMCAD), Portland, OR, November 2002. Lecture notes in computer science, vol 2517. Springer, Berlin Heidelberg New York, pp 202–219
28. Puri A, Holzmann GJ (2000) A minimized automaton representation of reachable states. Int J Softw Tools Technol Transfer 2(3):270–278
29. Ranjan RK, Sanghavi JV, Brayton RK, Sangiovanni-Vincentelli A (1996) Binary decision diagrams on network of workstations. In: Proceedings of the IEEE international conference on computer design, pp 358–364
30. Sanghavi JV, Ranjan RK, Brayton RK, Sangiovanni-Vincentelli A (1996) High performance bdd package by exploiting memory hierarchy. In: Proceedings of the 33rd IEEE conference on design automation, pp 635–640

31. Smv Web page: http://www.cs.cmu.edu/~modelcheck/
32. Spin Web page: http://spinroot.com
33. Stern U, Dill D (1997) Parallelizing the murφ verifier. In: Grumberg O (ed) Proceedings of the 9th international conference on computer aided verification, Haifa, Israel. Lecture notes in computer science, vol 1254. Springer, Berlin Heidelberg New York, pp 256–267
34. Stern U, Dill D (1998) Using magnetic disk instead of main memory in the murφ verifier. In: Hu AJ, Vardi MY (eds) Proceedings of the 10th international conference on computer aided verification, Vancouver, BC, Canada. Lecture notes in computer science, vol 1427. Springer, Berlin Heidelberg New York, pp 172–183
35. Stern U, Dill DL (1995) Improved probabilistic verification by hash compaction. In: Camurati P, Eveking H (eds) Proceedings of the IFIP WG 10.5 advanced research working conference on correct hardware design and verification methods (CHARME). Lecture notes in computer science, vol 987. Springer, Berlin Heidelberg New York, pp 206–224
36. Stern U, Dill DL (1996) A new scheme for memory-efficient probabilistic verification. In: Gotzhein R, Bredereke J (eds) Proceedings of the IFIP TC6/WG6.1 joint international conference on formal description techniques for distributed systems and communication protocols, and Protocol specification, testing, and verification, Proceedings of IFIP, vol 69. Kluwer, Dordrecht, pp 333–348
37. Steven Ulrich web page: http://verify.stanford.edu/uli/research.html
38. Stornetta T, Brewer F (1996) Implementation of an efficient parallel bdd package. In: Proceedings of the 33rd annual conference on design automation. ACM Press, New York, pp 641–644
39. Tronci E, Della Penna G, Intrigila B, Venturini Zilli M (2001) Exploiting transition locality in automatic verification. In: Margaria T, Melham T (eds) Proceedings of the IFIP WG 10.5 advanced research working conference on correct hardware design and verification methods (CHARME), September 2001. Lecture notes in computer science, vol 2144. Springer, Berlin Heidelberg New York, pp 259–274
40. Tronci E, Della Penna G, Intrigila B, Venturini Zilli M (2001) A probabilistic approach to space-time trading in automatic verification of concurrent systems. In: Proceedings of the 8th IEEE Asia-Pacific conference on software engineering (APSEC), Macau SAR, China, December 2001. IEEE Press, New York, pp 317–324
41. Uppaal web page: http://www.docs.uu.se/docs/rtmv/uppaal/
42. Wolper P, Leroy D (1993) Reliable hashing without collision detection. In: Courcoubetis C (ed) Proceedings of the 5th international conference on computer aided verification, Elounda, Greece. Lecture notes in computer science, vol 697. Springer, Berlin Heidelberg New York, pp 59–70
43. German S, private communication