Exploiting Hub States in Automatic Verification

Giuseppe Della Penna[1,⋆], Igor Melatti[1], Benedetto Intrigila[2], and Enrico Tronci[3]

[1] Dip. di Informatica, Università di L'Aquila, Coppito 67100, L'Aquila, Italy
{dellapenna, melatti}@di.univaq.it
[2] Dip. di Matematica, Università di Roma "Tor Vergata", Via della Ricerca Scientifica, 00133 Roma, Italy
intrigil@mat.uniroma2.it
[3] Dip. di Informatica, Università di Roma "La Sapienza", Via Salaria 113, 00198 Roma, Italy
tronci@di.uniroma1.it

Abstract. In this paper we present a new algorithm to counteract *state explosion* when using *Explicit State Space Exploration* to verify protocol-like systems.

We sketch the implementation of our algorithm within the Caching Murφ verifier and give experimental results showing its effectiveness.

We show experimentally that, when memory is a scarce resource, our algorithm improves on the time performances of Caching Murφ verification algorithm, saving between 16% and 68% (45% on average) in computation time.

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 and hybrid systems verification, *Explicit* State Space Exploration often outperforms *Symbolic* (i.e. OBDD based, [4,5]) State Space Exploration [1,13,8]. Since here we are mainly interested in protocol verification we focus on explicit state space exploration. Tools based on explicit state space exploration are, e.g., SPIN [17,23] and Murφ [11,19].

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 [30,7,18,28,26,16,12].

⋆ Corresponding Author: Giuseppe Della Penna. Tel: +39 0862 43 3130 Fax: +39 0862 43 3057.

D.A. Peled and Y.-K. Tsay (Eds.): ATVA 2005, LNCS 3707, pp. 54–68, 2005.

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 [24,25].

1.1 Background

In [27,10,9] we presented verification algorithms exploiting statistical properties of protocol transition graphs to save on RAM usage as well as to speed up disk based *Breadth First* (BF) state space exploration. Our algorithms have been implemented within the Murφ verifier. We call CMurφ (Caching Murφ [6]) the resulting verifier.

Shortly, CMurφ takes advantage of a statistical property of protocol transition graphs, namely the *transition locality*. That is, w.r.t the levels of a BF state space exploration, state transitions tend to be between states belonging to close levels of the transition graph. Thus, CMurφ 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. The rationale behind this approach is that a cache maintains only *recently visited* states. Thanks to transition locality this is sufficient, in most cases, to complete the verification task. Our experimental results [27,9] show that, with the same amount of RAM, CMurφ can verify systems more than 40% larger than those that can be handled using a hash table based approach. On the other hand, CMurφ verification time can be up to twice that of standard Murφ.

Note that CMurφ caching techniques is not an alternative to state compression techniques (e.g. hash compaction [28,26,16,12]) or to state space reduction techniques (e.g. symmetry and multiset reduction [7,18], partial order reduction [22]). On the contrary, caching is intended to be used together with the available reduction options [27,9]. The only thing that *caching* does is storing data in the cache. Such data can be full states, state signatures, or anything else. This is not relevant to the caching schema. This, of course, may be relevant for the effectiveness of the caching schema. As long as the implemented BF search uses a hash table to store visited states (or their signatures) CMurφ caching scheme can be used. For this reason CMurφ can reuse all state reduction procedures implemented in the standard Murφ verifier [27].

1.2 Goal

CMurφ memory saving stems from the fact the *most* transitions are local. On the other hand, CMurφ time penalty stems from the fact the *not all* transitions are local. A nonlocal transition leading to a rather old state that has been overwritten (and thus *forgotten*) in CMurφ cache may trigger revisit of large portions of the transition graph and may even lead to nontermination because of loops in the transition graph. The higher CMurφ cache *collision rate* (i.e. the ratio between collisions and insertions) the higher the probability of revisiting already visited states because of nonlocal transitions.

When the collision rate is high (i.e. close to 1) it means that we do not have enough RAM to hold all visited states. So our only hope to decrease the

time penalty due to revisiting in such a situation is to make a better use of the available RAM.

Quite clearly a (large) fraction of the available RAM must be used to store recently visited states. This is indeed what CMurφ already does. Here we propose to use a (small) fraction of the available RAM to store *hub states*, that is states that have an *in-degree* (i.e. number of incoming transitions) much greater than the average in-degree of the set of reachable states. The rationale behind such proposal is that *many* nonlocal transitions will lead to hub states. Thus avoiding revisiting hub states (and so their successors) may be an effective way to reduce CMurφ time penalty when the collision rate is high.

Note that when the collision rate is low (close to 0) it means that we have (almost) enough RAM to store *all* reachable states. In such a case CMurφ does not incur any time penalty. That is, verification with CMurφ takes the same amount of time as with standard Murφ [19].

Unfortunately our goal of storing hub states faces a substantial obstruction: we do not know *before hand* if a state is a hub or not. Thus, to carry out our goal we need a both time and memory effective way to select hub states among the states visited so far. In other words, the obstruction here is not in storing (the few) hub states, but rather in recognizing that a state seen during the visit is indeed a hub state.

In this paper we show that protocol-like systems do have hub states and present an effective algorithm to select hub states among the states visited so far.

Intuitively, we use a *hard to write* cache `L2`, that is a cache in which an insertion request is actually carried out with (a small) insertion probability p. This means that states that are frequently *seen* during our visit will have a greater chance than seldom seen states of actually making their way into `L2`. As a result, statistically speaking, `L2` will tend to store the hub states among the states visited so far. Of course not all hub states will be in `L2` nor all states in `L2` will be hubs. Still, we can show experimentally that `L2` is an effective way to catch hub states.

1.3 Main Results

Our main results can be summarized as follows.

In *Section 3* we show experimentally that protocol-like systems do have hub states. We support our claim by measuring the distribution of the in-degree of the reachable states for the set of protocols included in the Murφ verifier distribution [19].

In *Section 4* we present our algorithm to select hub states among the states visited so far.

In *Section 5* we show how an appropriate value for the insertion probability p in `L2` can be computed.

We implemented our algorithm within the CMurφ [6] verifier. We call HubCMurφ the resulting verifier. In *Section 6* we give experimental results on HubCMurφ as compared to CMurφ. Our experimental results show that when

the collision rate is high typically HubCMurφ allows between 16% and 68% (45% on average) of saving in the verification time. Of course when the collision rate is low HubCMurφ behaves essentially as CMurφ.

1.4 Related Works

A rather systematic study of statistical properties of transition graphs is presented in [21]. The author of [21] concludes that there are no hubs in transition graphs. Note however that the definition of hub state used in [21] is different form ours. For us a reachable state s is a hub state if its in-degree is *much higher* than the average in-degree of the reachable states whereas [21] also requires the s in-degree to be *not too smaller* than the number of (reachable) states.

Of course what is the *right* definition of hub depends on the intended application. Anyway, because of this different definition of hub states there is no contradiction between our results about hub existence and those in [21].

Moreover the focus of our paper is not proving or disproving hub existence but rather finding ways to exploit the fact that there are states whose in-degree is much higher that the average one. Finally, the issue of exploiting statistical properties of transition graphs is not investigated in [21].

A survey on caching schemes is presented in [15]. Note however that [15] studies *Depth First* (DF) search with a linked list based hash table. Caching Murφ [27,9] instead uses a BF search with an open addressing hash table. As remarked in [15] this is a quite different scenario. In fact, CMurφ caching schema works quite well [27,9] with open addressing *and* BF search and does not seem to work with a DF search (SPIN like).

Note that we do not reduce the state space using our hub states. So our approach has nothing to do with *Partial Order* (PO) reduction [22] techniques. On the other hand we can exploit hub states (if any) in a PO reduced state space.

Finally, [3], using static analysis techniques, studies the issue of which states should be stored in order to save RAM. The results in [3] are orthogonal to ours, note however that the two approaches can be usefully combined.

2 Background

In this section we give some basic definitions that will be useful in the following.

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, $\mathcal{A}$, R) where: S is a finite set (of states), $I \subseteq S$ is the set of initial states, $\mathcal{A}$ 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* if and only if $R(s, a, s')$ holds. The set of successors of state s (notation $\texttt{next}(s)$) is the set of states s' such that there exists $a \in A$ such that $R(s, a, s')$ holds.

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 such that
   1. $I \subseteq \mathbf{Reach}(\mathcal{S})$, 2. for all $s \in \mathbf{Reach}(\mathcal{S})$, $\texttt{next}(s) \subseteq \mathbf{Reach}(\mathcal{S})$.

```
FIFOQueue Q; HashTable T;
bfs(FSS S) { let S = (S,I,A,R);
 foreach s in I {Enqueue(Q, s); Insert(T, s);} /*init*/
 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. Basic Breadth First Search

In the following we will always refer to a given system $\mathcal{S} = (S, I, \mathcal{A}, R)$. 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*, that is 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., by using a *Depth–First* (DF) search or a *Breadth–First* (BF) search. For example, Murφ [19] and (the latest version of) SPIN [23] may use a DF as well as a BF search.

In the following we will focus on BF search. The Murφ algorithm for the BF visit is shown in Figure 1. Namely, function `bfs` of Figure 1 takes as input a FSS $\mathcal{S}$ and performs a BF visit of $\mathcal{S}$ transition graph. To this end, it uses a FIFO queue `Q` and a hash table `T`. The first maintains the BF front (i.e. the states to be expanded), while the latter stores the visited states, so avoiding to revisit the same states. Thus, state explosion occurs on `T` and `Q`. Finally, note that, if `T` and `Q` fit in the available memory, `bfs` will surely terminate, since the set of reachable states is finite.

3 Hub States

Inspired by [14,2,29] we call *hub* a reachable state which in-degree is much higher than the average in-degree of all reachable states. Note that, as discussed in Section 1.4, our definition of hub state is different from the one used in [21].

In this section we show experimentally that for protocol-like systems hub states do exist. We do this by showing that all our benchmark protocols indeed have hub states. We use as benchmark protocols all those available in the Murφ verifier distribution [19]. 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 Measuring Hub States Presence

In this section we give the basic definitions needed to understand the experimental results in Section 3.2.

Definition 2. *Let $\mathcal{S}$ = (S, I, A, R) be an FSS, and let $s \in S$ be a state. We call* in-degree *of the state s the number* $\texttt{indeg}(s)$ *of transitions leading to s. That is:* $\texttt{indeg}(s) = |\{(r,a) \in \mathbf{Reach}(\mathcal{S}) \times A \mid R(r,a,s)\}|$.

Our goal is to study the in-degree distribution in protocol-like systems. As usual when reporting statistical results, to make distributions relative to different systems easily comparable we replace the absolute number of states with the fraction x of reachable states and the actual in-degree value with its fraction of the maximum in-degree. In this way all quantities lie in the interval $[0,1]$.

To build the in-degree distribution we proceed in the standard way. Namely, we divide the interval $[0,1]$ in $\lceil \frac{1}{\Delta} \rceil$ subintervals of length Δ and, for each subinterval k, we compute the fraction of the reachable states whose fraction of the maximum in-degree falls in interval k. The following definition gives the formal details.

Definition 3. *Let $\mathcal{S}$ = (S, I, A, R) be an FSS,* $M_{\texttt{indeg}} = \max\{\texttt{indeg}(t) | t \in \mathbf{Reach}(\mathcal{S})\}$ *be the maximum in-degree of $\mathcal{S}$ and $\Delta \in [0,1]$.*

- *We define function $\theta : (0,1] \times \mathbb{N} \to [0,1]$ as follows:*

$$\theta(\Delta, k) = \frac{|\ \{s \in \mathbf{Reach}(\mathcal{S}) \mid (k-1)\Delta M_{\texttt{indeg}} < \texttt{indeg}(s) \leq k\Delta M_{\texttt{indeg}}\}\ |}{|\mathbf{Reach}(\mathcal{S})|}$$

  Function $\theta(\Delta, k)$ returns the fraction of reachable states whose in-degree is a fraction $y \in ((k-1)\Delta, k\Delta]$ of the maximum in-degree. In other words, $\theta(\Delta, k)$ returns the probability that a reachable state has an in-degree which is a fraction $y \in ((k-1)\Delta, k\Delta]$ of the maximum in-degree. Thus, technically speaking, $\theta(\Delta, k)$ is a probability density*. Of course, for us, function $\theta(\Delta, k)$ is only interesting when $k \leq \frac{1}{\Delta}$.*
- *We define function $\tau : (0,1] \times [0,1] \to [0,1]$ as follows:*

$$\tau(\Delta, x) = \theta\left(\Delta, \left\lceil \frac{x}{\Delta} \right\rceil\right)$$

  We also write $\tau_\Delta(x)$ for $\tau(\Delta, x)$ and denote with τ_Δ function $\lambda x.\tau(\Delta, x)$. That is, $\tau_\Delta : (0,1] \to [0,1]$ is defined as $\tau_\Delta(x) = \tau(\Delta, x)$.
  Note that function τ_Δ is completely defined once we know the values $\tau_\Delta(\Delta)$, $\tau_\Delta(2\Delta)$, $\ldots \tau_\Delta(1)$.

3.2 Experimental Results About Hub States

To carry out our plan we modified the Murφ verifier so as to compute function τ in Definition 3. Namely, we compute $\tau_{\frac{1}{n}}(\frac{1}{n}), \ldots, \tau_{\frac{1}{n}}(\frac{n-1}{n}), \tau_{\frac{1}{n}}(1)$ while performing state space exploration. In our experiments, we set $n = 20$.

Our results are shown in Figure 2 where, for each protocol in our benchmark, we plot $\tau(\frac{1}{n}, x)$ (y-axes) versus the fraction x of the maximum in-degree (x-axes).

The graphs in Figure 2 show that most reachable states have an in-degree that is a rather small fraction of the maximum in-degree. However there is a small fraction of states that have an in-degree that is close to the maximum in-degree.

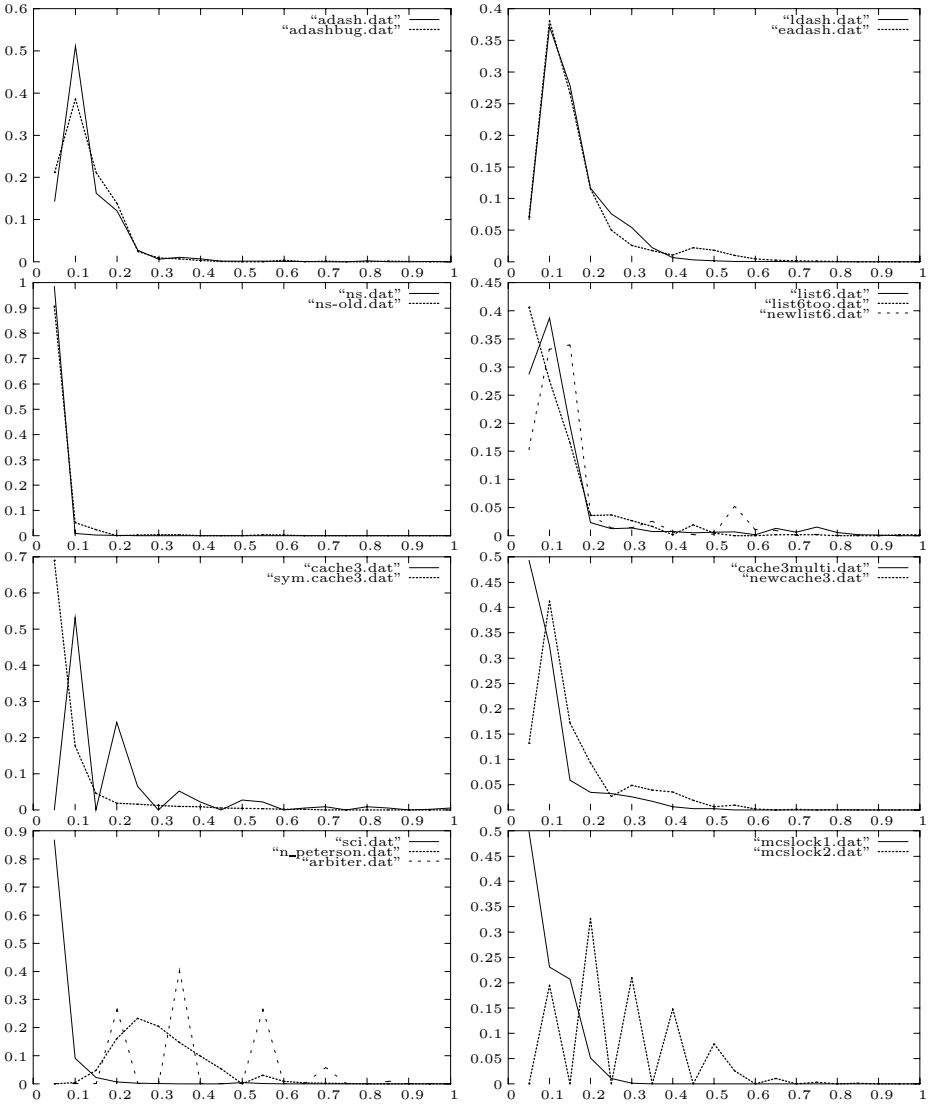

Fig. 2. Density of probability $\tau_{\frac{1}{20}}$ graphs for protocols included in the Murφ distribution. The curves show the fraction of reachable states y which in-degree is a fraction x of the max in-degree. Thus, by definition $y > 0$ when $x = 1$. Note log scale on y axes.

4 Exploiting Hub States in State Space Exploration

In this section we present an algorithm that is able to effectively select hub states among the states visited so far. Note that the correctness of our algorithm does not depend on the results in Section 3. However such results will help us to understand why the proposed algorithm is effective on protocol-like systems.

Before describing our algorithm, in Figure 3 we briefly recall the CMurφ [9] one. With respect to Figure 1, we have that in Figure 3 the queue is now implemented on disk, and the hash table `T` is replaced by a cache. That is, if the insertion of a state `s'` causes a collision because of a state `t` already in cache `T`, then `t` is overwritten (and thus *forgotten*). This implies that, if `t` is reached again, it will be revisited, since it is not in our cache anymore. This means that in general cache `T` may not be able to prevent nontermination of our visit. As shown in Figure 3, to guarantee termination in CMurφ, the main `while` cycle is guarded by the *collision rate*, i.e. the ratio between the number of collisions and the number of insertions in cache `T`. In fact, when the collision rate becomes too high, we are visiting over and over the same set of states. In this case we should give up our verification task because of lack of memory.

In order to improve CMurφ time performances, we present a new two-level caching algorithm. The rationale behind this algorithm is the one discussed in Section 1.2.

To implement the ideas in Section 1.2 we proceed as follows. We modify the cache based BF algorithm in Figure 3 as shown in Figures 4 and 5. Namely, we split cache `T` in two parts, `L1` and `L2`, with a split ratio $0 < p_h < 1$. Thus, if M was the amount of RAM dedicated to `T`, then $p_h M$ is now dedicated to `L1` and $(1 - p_h)M$ to `L2`. In our experiments, we set $p_h = 0.7$. This is a reasonable value, since hub states are always a very small subset of the reachable states (see Figure 2).

The idea is to use `L1` to store the recently visited states, so inheriting the goal of `T` (i.e. to exploit transition locality) in CMurφ, and `L2` (our *hard to write* cache) to store the hub states. To this end, the algorithm now stores the visited states in `L1` (function `Insert` in Figure 4) and, when the insertion of a state `s'` causes a collision in `L1` on state `t`, `t` is passed to `L2` *before* being overwritten. If this causes a collision also in `L2` on a state `r`, `r` will be overwritten by `t` with a fixed probability `p_ovrwrt` (functions `Insert_L2` and `prob_decide` in Figure 5). Of course a state is considered visited if it can be found in `L1` or `L2` (see functions `Insert` and `Lookup_L2` in Figures 4 and 5, respectively).

In this way, if state `t` is a hub, it will have a high probability of being eventually inserted in `L2` and remaining there. In fact, since `t` will be reached more often than the other states, it will be often present and overwritten in `L1` and, as a result, it will attempt insertion in `L2` many times. This gives `t` more chances of entering `L2` since it will compete more times for the insertion.

We implemented the algorithm of Figures 4 and 5 within the CMurφ verifier [6], calling HubCMurφ the resulting verifier.

```
FIFOQueue Q;        Cache T;
collision_rate = 0.0; /* #collisions on T / #insertions in T */
cbfs(FSS S) { let S = (S, I, A, R);
 foreach s in I {Enqueue(Q, s); 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. 3. Cache based Breadth First Search

```
Insert(s) {h = hash_key(s);
 if (L1[h] == s) { /*cache hit (state found)*/
  return true; /* report a cache hit */
 } else { /* s not in L1 */
  if (Lookup_L2(s)) { /* but s is in L2 */
   return true; /* report a cache hit */
  } else { /*s is neither in L1 nor in L2, insert it*/
   if (L1[h] is empty) {L1[h] = s;
   } else { /* the slot is full, overwrite it */
    s' = L1[h];
    /* before overwriting s', pass it to L2 */
    Insert_L2(s'); L1[h] = s; }}
  return false; /* report a cache miss */ }}
```

Fig. 4. Function `Insert`

```
Lookup_L2(s) { h = hash_key2(s);
 if (L2[h] == s) return true; else return false; }

Insert_L2(s) { h = hash_key2(s);
 if (L2[h] == s) return true; /* report a cache hit */
 else if (L2[h] is empty) L2[h] = s;
  else /* slot full, we may choose to overwrite */
  if (prob_decide(p_ovrwrt)) L2[h] = s;
 return false; /* report a cache miss */ }

prob_decide(p) {
  return true with probability p, false otherwise;}
```

Fig. 5. Functions `Lookup_L2`, `Insert_L2` and `prob_decide`

5 Tuning the Overwrite Probability

As already said in Section 4, a state that causes a collision in the `L2` cache is overwritten with a fixed probability `p_ovrwrt`. To make `L2` effective in finding and retaining hub states, it is important to choose a suitable value of `p_ovrwrt`.

We carried out a set of experiments to determine a reasonable value for `p_ovrwrt`. In particular, Figure 6 shows the collision rate as a function of the fraction of visited reachable states for values of `p_ovrwrt` in $\{10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1\}$. When `p_ovrwrt` $\leq 10^{-5}$ the collision rate becomes soon pretty high and the visits stops. This is because when `p_ovrwrt` s *too small* it is almost like not having `L2` at all. For this reason we only plotted `p_ovrwrt` in the range $\{10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1\}$. Note that the protocol set used in these experiments is the same one used in Section 6 to assess performances of our algorithm.

Figure 6 shows that when `p_ovrwrt` is 1 there are cases in which verification does not terminate. For example this happens for protocols `mcslock1`, `mcslock2` and `newlists6` in Figure 6.

Note that setting `p_ovrwrt` to 1 is equivalent two using the standard *victim cache* approach in processor design [20]. However, this does not work in our setting, since in this way the algorithm will overwrite too many states (hubs included) thus leading to nontermination.

On the other hand if `p_ovrwrt` is *too small* (namely less than 10^{-4}) then `L2` will (almost) never be used and, all in all, we have wasted a fraction p_h (see Section 4) of our RAM.

Finally, if `p_ovrwrt` is *small enough*, only states that are encountered many times during the exploration process will make their way to `L2`. Summing up, in our experiments we choose to set `p_ovrwrt` $= 10^{-4}$.

6 Experimental Results

We report the experimental results we obtained using HubCMurφ (Section 4).

We want to measure how much time and (RAM) memory we can save by using our 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 [19]).

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`). However, since in our scenario

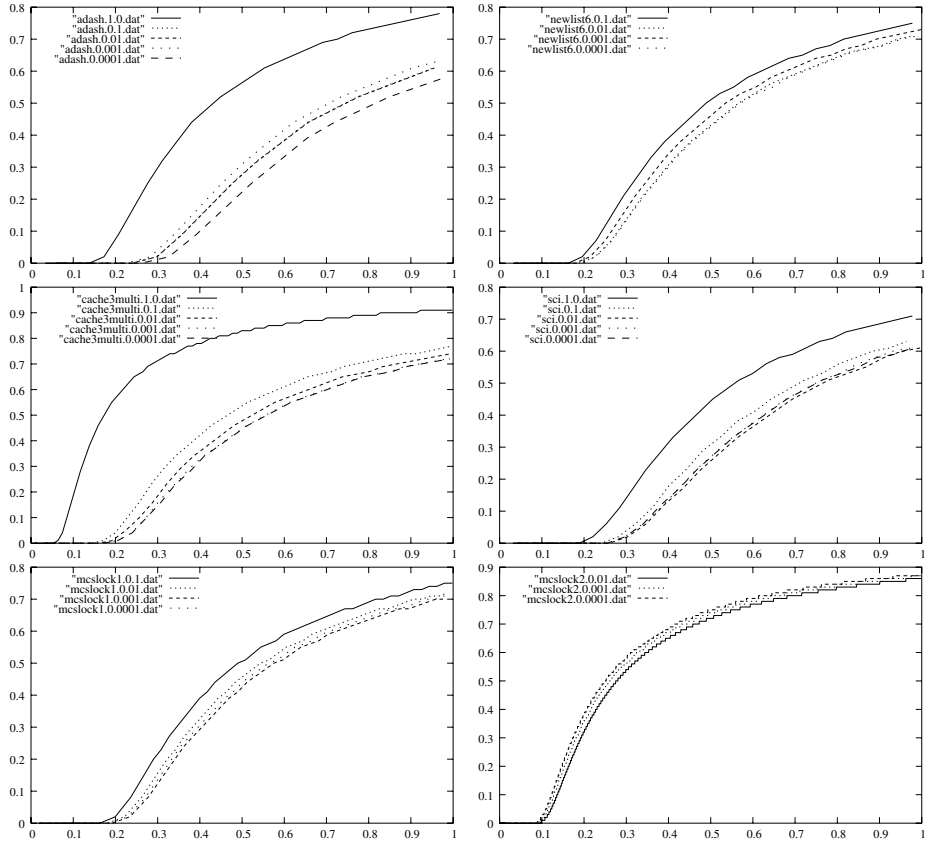

Fig. 6. Collision rate as a function of the fraction of visited protocol states. Each graph shows the collision rate for values of `p_ovrwrt` in $\{10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1\}$. A missing line indicates that the verifier was unable to complete the verification with the corresponding value of `p_ovrwrt`.

RAM is a scarce resource, in the following we only consider the case in which both options are enabled on all verifiers (i.e. Murφ, CMurφ, HubCMurφ). Moreover, in order to visit all reachable states, all experiments have been carried out with deadlock detection disabled (`-ndl`).

Our results are in Figure 7, where we only show protocols requiring at least 10 kilobytes of RAM and a nonnegligible amount of time to complete state space exploration. In Figure 7, column ***M*** gives the minimum amount of memory (in kilobytes) needed to complete state space exploration and column ***T*** gives the time (in seconds) to complete state space exploration when using memory M. Finally, column ***Reach*** gives the number of reachable states.

Our next step is to run each protocol p with less and less memory using both HubCMurφ and CMurφ. That is we run protocol p with memory limits $\alpha M(p)$, $\alpha \in [0, 1]$, with the new (`L1`+`L2`) and the old (just CMurφ `L1`) cache

based algorithm. This approach allows us to easily compare the experimental results obtained from different protocols.

The results obtained in such a way are in Fig. 9. Note that in these experiments the value used for g (`gPercentActive`) is not relevant since the queue is implemented on disk. We give the meaning of rows and columns in Fig. 9.

Column α (with $\alpha \in [0, 1]$) gives information about the run of protocol p with memory $\alpha M(p)$ (for this reason, the row heading is ***Mem***).

Row ***States*** gives $\frac{N_{hub}}{N_{nohub}}$, where N_{nohub} is the number of visited states using CMurφ and N_{hub} is the number of visited states using HubCMurφ.

Row ***Time*** gives $\frac{T_{hub}}{T_{nohub}}$, where T_{nohub} is the computation time needed by CMurφ and T_{hub} is the computation time needed by HubCMurφ.

A verifier (CMurφ or HubCMurφ) is stopped when its collision rate becomes greater than 0.99. We mark with a $*$ superscript the data obtained when CMurφ gives up state space exploration because its collision rate exceeds the given threshold (0.99) and, on the contrary, HubCMurφ succeeds in completing the verification. In such cases, instead of giving a ratio, rows ***States*** and ***Time*** display, respectively, the absolute values for the visited states and the computation time (in seconds) of HubCMurφ. Note that there was no case in which only CMurφ completed the verification.

We are interested in the case in which the collision rate is high, since this means that we do not have enough RAM to store all visited states. For this reason when comparing CMurφ and HubCMurφ performances we only consider the results obtained from the experiments relative to the least α in which both CMurφ and HubCMurφ terminate. This means that column (α - 0.01) is marked with a $*$ (only HubCMurφ terminates). When the collision rate is low (i.e. we do have enough memory to store *most* of the visited states) CMurφ and HubCMurφ have similar performances. This can be seen from Figure 9 by looking at the column with the largest value of α (namely the leftmost column).

The experimental results in Figure 9 show that, with respect to CMurφ, HubCMurφ typically saves from 16% to 68% (45% on average) in computation time. Note also that for all protocols there are cases in which, with the available memory, only HubCMurφ is able to terminate.

Of course there are protocols (e.g. `n_peterson` in Figure 9) where HubCMurφ is less efficient than CMurφ. We conjecture that this is due to the shape of the in-degree distribution curves in Figure 2. First, we should note that, technically speaking, the curves in Figure 2 are *density* of probabilities. Now, for each protocol p we can compare the curve for p in Figure 2 with HubCMurφ performances for p as from Figure 9. From this we see that if the curve of p is rather *concentrated* (i.e., has a small *variance*) then HubCMurφ performs well on p (e.g., as for protocol `sci`). On the other hand, if p curve has a large variance (e.g. as for `mcslock2` and `n_peterson`) then HubCMurφ does not perform well on p.

We also wanted to test our approach with a large protocol that heavily loads our machine. The results are in Fig. 8. We used protocol `sci` with parameter `MemorySize` set to 5. As shown in [10], this protocol has 75,081,011 reachable

Protocol	Reach	M	T
n_peterson	163298	813	273.32
adash	10466	55	62.98
cache3multi	13738	73	35.11
newlist6	13044	67	18.34
mcslock1	23644	120	16.76
mcslock2	540219	2693	237.48
sci	18193	94	28.17

Fig. 7. Results on a SUN Sparc machine with 512M RAM

Mem	**0.41**	**0.37**
States	80430178*	84045856*
Time	47129*	46604*
Mem	**0.33**	**0.29**
States	92322597*	120543398*
Time	51009*	66676*

Fig. 8. HubCMurφ experimental results for protocol `sci-31151` with parameter `MemorySize` = 5

mcslock1	**Mem**	**0.60**	**0.59**	**0.58**	**0.57**	**0.56**	**0.55**	**0.54**	**0.53**	**0.52**	**0.51**	**0.50**	**0.49**
	States	0.603	0.352	72358*	104019*	134834*							
	Time	0.69	0.42	3.89*	5.62*	7.36*							
cache3multi	**Mem**	**0.60**	**0.59**	**0.58**	**0.57**	**0.56**	**0.55**	**0.54**	**0.53**	**0.52**	**0.51**	**0.50**	**0.49**
	States	0.828	0.89	0.769	0.77	66687*	87096*						
	Time	0.83	0.92	0.78	0.79	19.39*	25.22*						
mcslock2	**Mem**	**0.60**	**0.59**	**0.58**	**0.57**	**0.56**	**0.55**	**0.54**	**0.53**	**0.52**	**0.51**	**0.50**	**0.49**
	States	0.99	0.976	0.956	0.93	0.919	0.885	0.805	0.714	1164348*	1397335*	2085105*	
	Time	1.19	1.16	1.14	1.12	1.09	1.05	0.96	0.84	39.67*	47.92*	72.06*	
newlist6	**Mem**	**0.60**	**0.59**	**0.58**	**0.57**	**0.56**	**0.55**	**0.54**	**0.53**	**0.52**	**0.51**	**0.50**	**0.49**
	States	0.697	0.296	48882*	63843*								
	Time	0.75	0.32	5.14*	6.69*								
adash	**Mem**	**0.60**	**0.59**	**0.58**	**0.57**	**0.56**	**0.55**	**0.54**	**0.53**	**0.52**	**0.51**	**0.50**	**0.49**
	States	0.87	0.873	0.421	0.584	21362*	32166*						
	Time	0.88	0.91	0.44	0.61	11.52*	17.27*						
sci	**Mem**	**0.49**	**0.48**	**0.47**	**0.46**	**0.45**	**0.44**	**0.43**	**0.42**	**0.41**	**0.40**	**0.39**	**0.38**
	States	0.914	0.799	0.833	0.797	0.693	0.818	0.626	0.305	34019*	41096*	47845*	91666*
	Time	0.95	0.8	0.84	0.8	0.69	0.86	0.64	0.32	5.28*	6.42*	7.39*	14.17*
n_peterson	**Mem**	**0.70**	**0.69**	**0.68**	**0.67**	**0.66**	**0.65**	**0.64**	**0.63**	**0.62**	**0.61**	**0.60**	**0.59**
	States	1.079	1.078	1.028	1.311	1.071	0.663	1.635	1.188	1.032	5588575*		
	Time	1.2	1.2	1.14	1.46	1.18	0.73	1.82	1.32	1.14	368.17*		

Fig. 9. Comparison of CMurφ and HubCMurφ experimental results on an INTEL Pentium 3.2GHz machine with 512M RAM

states and requires 563 Megabytes of memory to be verified with standard Murφ in 35,905 seconds.

CMurφ was not able to complete verification with less than 225 Megabytes, that is 40% of the required (563MB) memory.

On the other hand, as shown in Fig. 8, HubCMurφ completed the verification with about 163 MB, that is 29% of the required memory, and a time penalty (w.r.t. standard Murφ with 563MB of RAM) of 85%.

This suggests that for large protocols HubCMurφ can achieve huge (about 71% in our example above) memory savings, possibly at the expense of time. This is better than being left with an *out of memory* message after hours of computation.

7 Conclusions

We presented a novel explicit verification algorithm that exploits hub states (Section 3) to save on memory usage (Sections 4, 5). We implemented our algorithm within the CMurφ verifier [6] and call HubCMurφ the resulting verifier.

Our experimental results (Section 6) show that, with respect to CMurφ, HubCMurφ typically saves from 16% to 68% (45% on average) in computation time.

Acknowledgments

We gratefully acknowledge discussing with Alan Hu during CHARME 2001 conference the possibility of using a small *victim cache* in order to improve CMurφ performances. Although in its basic form a victim cache does not meet our goals here, it is also quite clear that our *hard to write* second level cache is a sort of (*lucky*) victim cache.

References

1. A.J. Hu, G. York, and D.L. Dill. New Techniques for Efficient Verification with Implicitly Conjoined BDDs. In *31st ACM/IEEE Design Automation Conference (DAC)*, San Diego, CA, USA, 1994.
2. Albert-Laszlo Barabasi. *Linked*. Perseus Publishing, 2002.
3. G. Behrmann, K. G. Larsen, and R. Pelánek. To store or not to store. In *Proc. of 15th Int. Conf. on: Computer Aided Verification (CAV)*, volume 2725 of *Lecture Notes in Computer Science*, Boulder, CO, USA, July 2033. Springer.
4. R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, C-35(8):677–691, Aug 1986.
5. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Inf. Comput.*, 98(2):142–170, 1992.
6. Caching murphi web page: http://www.dsi.uniroma1.it/~tronci/cached.murphi.html, 2004.
7. C.N. Ip and D.L. Dill. Better verification through symmetry. In *Computer Hardware Description Languages and their Applications*, Ottawa, Canada, 1993. Elsevier Science Publishers B.V., Amsterdam, Netherland.
8. G. Della Penna, B. Intrigila, I. Melatti, M. Minichino, E. Ciancamerla, A. Parisse, E. Tronci, and M. Venturini Zilli. Automatic verification of a turbogas control system with the murφ verifier. In *Hybrid Systems: Computation and Control, HSCC, Proc.*, volume 2623 of *Lecture Notes in Computer Science*. Springer, 2003.
9. G. Della Penna, B. Intrigila, I. Melatti, E. Tronci, and M. Venturini Zilli. Exploiting transition locality in automatic verification of finite state concurrent systems. *STTT*, 6(4), 2004.
10. G. Della Penna, B. Intrigila, E. Tronci, and M. Venturini Zilli. Exploiting transition locality in the disk based murφ verifier. In *Formal Methods in Computer-Aided Design, FMCAD'02, Proc.*, volume 2517 of *Lecture Notes in Computer Science*. Springer, 2002.
11. D. L. Dill, A. J. Drexler, A. J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *Proc. of the 1991 IEEE Int. Conf. on Computer Design on VLSI in Computer & Processors*. IEEE Computer Society, 1992.
12. Peter C. Dillinger and Panagiotis Manolios. Bloom filters in probabilistic verification. In *Formal Methods in Computer-Aided Design, FMCAD, Proc.*, volume 3312 of *Lecture Notes in Computer Science*. Springer, Nov 2004.
13. Cindy Eisner and Doron Peled. Comparing symbolic and explicit model checking of a software system. In *SPIN Workshop, Proc.*, Lecture Notes in Computer Science. Springer, 2002.
14. Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. In *SIGCOMM '99: Proc. of the Conf. on Applications, technologies, architectures, and protocols for computer communication*, New York, NY, USA, 1999. ACM Press.

15. Jaco Geldenhuys. State caching reconsidered. In *SPIN Workshop, Proc.*, volume 2989 of *Lecture Notes in Computer Science*. Springer, 2004.
16. Gerard J. Holzmann. An analysis of bitstate hashing. *Form. Methods Syst. Des.*, 13(3):289–307, 1998.
17. Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison Wesley Professional, 2004.
18. C. N. Ip and D. L. Dill. Efficient verification of symmetric concurrent systems. In *Proc. of the IEEE Int. Conf. on Computer Design: VLSI in Computers and Processors*, Cambridge, MA, October 1993. IEEE Computer Society Press.
19. Murphi web page: http://sprout.stanford.edu/dill/murphi.html, 2004.
20. David A. Patterson and John L. Hennessy. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers Inc., 1996.
21. Radek Pelánek. Typical structural properties of state spaces. In *SPIN Workshop, Proc.*, volume 2989 of *Lecture Notes in Computer Science*. Springer, 2004.
22. D. Peled. Ten years of partial order reduction. In *Computer Aided Verification, CAV, Proc.*, volume 1427 of *Lecture Notes in Computer Science*. Springer, 1998.
23. Spin web page: http://spinroot.com, 2004.
24. U. Stern and D. Dill. Parallelizing the murφ verifier. In *Computer Aided Verification (CAV), Proc*, volume 1254 of *Lecture Notes in Computer Science*. Springer, 1997.
25. U. Stern and D. Dill. Using magnetic disk instead of main memory in the murφ verifier. In *Computer Aided Verification (CAV), Proc.*, volume 1427 of *Lecture Notes in Computer Science*. Springer, 1998.
26. U. Stern and D. L. Dill. A new scheme for memory-efficient probabilistic verification. In *IFIP TC6/WG6.1 Joint Int. Conf. on: Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification*, volume 69 of *IFIP Conference Proceedings*. Kluwer, 1996.
27. E. Tronci, G. Della Penna, B. Intrigila, and M. Venturini Zilli. Exploiting transition locality in automatic verification. In *Correct Hardware Design and Verification Methods, CHARME, Proc.*, volume 2144 of *Lecture Notes in Computer Science*. Springer, 2001.
28. U. Stern and D.L. Dill. Improved Probabilistic Verification by Hash Compaction. In *Correct Hardware Design and Verification Methods*, volume 987, Stanford University, USA, 1995. Springer-Verlag.
29. D. J. Watts. *Small Worlds: The Dynamics of Networks Between Order and Randomness*. Princeton Univ. Press, Princeton, NJ, USA, 1999.
30. Pierre Wolper and Dennis Leroy. Reliable hashing without collision detection. In *Computer Aided Verification (CAV), Proc.*, volume 697 of *Lecture Notes in Computer Science*. Springer, 1993.