Hierarchical Adaptive State Space Caching based on Level Sampling

Radu Mateescu, Anton Wijs

Hierarchical Adaptive State Space Caching based on Level Sampling

Radu Mateescu and Anton Wijs

INRIA / VASY, 655, avenue de l'Europe, F-38330 Montbonnot St Martin, France
{Radu.Mateescu, Anton.Wijs}@inria.fr

Abstract. In the past, several attempts have been made to deal with the state space explosion problem by equipping a depth-first search (DFS) algorithm with a state cache, or by avoiding collision detection, thereby keeping the state hash table at a fixed size. Most of these attempts are tailored specifically for DFS, and are often not guaranteed to terminate and/or to exhaustively visit all the states. In this paper, we propose a general framework of hierarchical caches which can also be used by breadth-first searches (BFS). Our method, based on an adequate sampling of BFS levels during the traversal, guarantees that the BFS terminates and traverses all transitions of the state space. We define several (static or adaptive) configurations of hierarchical caches and we study experimentally their effectiveness on benchmark examples of state spaces and on several communication protocols, using a generic implementation of the cache framework that we developed within the CADP toolbox.

1 Introduction

In model checking, the *state space explosion* problem is the most important issue. It stems from the fact that a linear growth of the number of concurrent processes in a specification leads to an exponential growth of the number of states in the resulting state space. This problem strongly limits the possibilities to verify large systems, since state space generation algorithms typically need to keep all generated states in memory, thereby exhausting it quickly. Over the years, many techniques have been introduced to fight it, e.g., *partial order reduction* [17, 5], using *secondary storage* [9, 18], *distributed model checking* [14, 3, 2], and *directed model checking* [10, 11].

Another research branch is to consider partial storage of the previously explored states. This idea has lead, roughly speaking, to two classes of approaches: one where exhaustive exploration of the state space is guaranteed, and one where it is not. Since we are concerned with state space *generation*, i.e., the traversal of all the transitions in a state space, we focus on the first class; the reader is referred to Section 5 for the second one. The first class contains most work on state space *caching* [20], where a cache is employed which can never contain more than *n* states, and a technique which can be referred to as *covering set determination* [1] where, by means of static analysis, the goal is to identify which states to store in order to guarantee termination of the exploration. The caching approach is usually reserved for *depth-first search* (DFS) exploration, since termination can be guaranteed by efficient cycle detection, as opposed to when using *breadth-first search* (BFS). However, partial storage of explored states for BFS is desirable as well, since BFS (unlike DFS) can be efficiently distributed in order to perform the state space exploration using clusters of machines.

In this paper, we focus on state space generation with the goal of storing the state space on disk, meaning that besides minimising the memory use, we also aim at reducing the number of state revisits, which determines the amount of redundant information in the generated state space. This distinguishes our work from earlier work on state space caching, where the goal was to visit all states (but not necessarily traverse all transitions) and hence memory use was the main factor of importance. First, we propose a framework allowing hierarchies of caches in addition to single caches. To our knowledge, this has not been investigated yet in this context, although it is quite common practice in the field of hardware architectures [19]. The main idea is that several caches together, each having different characteristics, can be more efficient than one single cache. We study the performance of several hierarchical cache configurations using DFS, some of them improving on results shown by single cache setups. Second, we explain how these caches can be used for BFS by storing *search levels* instead of individual states in them. This provides a generic (language-independent) *on-the-fly* covering set estimation, instead of a (language-dependent) one based on static analysis. Our main result in this setting guarantees termination of the BFS with caches if appropriate *sampling functions* are used to decide which search levels must be stored. Finally, we propose a learning mechanism allowing the BFS algorithm to adapt on-the-fly to the state space structure by detecting its earlier mistakes.

Using the CADP toolbox [13] and the underlying OPEN/CÆSAR environment [12] for state space manipulation, we have implemented generic libraries for hierarchical caches, as well as cache-equipped BFS and DFS exploration tools, which allow to finely tune the memory use. We applied these tools on many examples of state spaces, taken either from benchmarks such as VLTS [32], or produced from communication protocols. These experiments allowed us to identify the most effective cache configurations, which can lead to memory reductions of over one order of magnitude, and even to speedups of the generation. Our results improve over existing ones by being language-independent and robust to the variations of the so-called *locality* (roughly, the size of back jumps during BFS) of state spaces.

The paper is organized as follows. Section 2 defines the terminology we use for state space exploration. Section 3 presents our cache-based BFS algorithm based on sampling of levels, proves its termination, and defines several instances of it using hierarchical caches and adaptive mechanisms. Section 4 briefly describes the implementation within CADP and gives various performance measures. Finally, Section 5 compares our approach with previous work and Section 6 gives concluding remarks and directions for future work.

2 Preliminaries

Labelled transition systems (LTSs) capture the operational behaviour of concurrent systems. An LTS consists of transitions $s \xrightarrow{\ell} s'$, meaning that being in a state s, an action ℓ can be executed, after which a state s' is reached.

Definition 1. *A* labelled transition system (LTS) *is a tuple $\mathscr{M} = (\mathscr{S}, \mathscr{A}, \mathscr{T}, s_0)$, where $\mathscr{S}$ is a set of states, $\mathscr{A}$ a set of transition labels, $\mathscr{T} : \mathscr{S} \times \mathscr{A} \times \mathscr{S}$ a transition relation, and s_0 the initial state. A transition $(s, \ell, s') \in \mathscr{T}$ is denoted by $s \xrightarrow{\ell} s'$.*

Furthermore, we express that a state s' is reachable from s with $s \rightarrow^* s'$, where $\rightarrow^*$ is the reflexive, transitive closure of $\rightarrow$. Likewise, we express with $s \rightarrow^+ s'$ that s' is reachable from s by traversing at least one transition. The set of enabled transitions of s is $en(s) = \{(s, \ell, s') \mid \exists \ell \in \mathscr{A}, s' \in \mathscr{S} . s \xrightarrow{\ell} s'\}$, and the set of successor states $succ(s) = \{s' \in \mathscr{S} \mid \exists \ell \in \mathscr{A} . (s, \ell, s') \in en(s)\}$ consists of all states reachable from s via a single transition.

Before generation, the structure of an LTS is not known. A generation algorithm is given an *implicit* LTS $\mathscr{M}_{im} = (s_0, en)$, with s_0 an initial state, and *en* the enabled function which can be used to generate the other states and the transitions between them, thereby creating an *explicit* LTS in line with Definition 1. We call an LTS *finite* iff $\mathscr{S}$ and $\mathscr{A}$ are of a finite size. Whenever a state is newly discovered, we call it a *visited* state; once we have generated all transitions and successors of s by employing the enabled function, we call s an *explored* state. Recognising when a newly visited state is already explored is called *duplicate detection*. In a standard BFS, all previously explored states are stored in memory, in what is called the *Closed* set, and all visited states yet to be explored are stored in the *Open* set, also called the *search horizon*. Given a nonempty *Open* set, the exploration of all the states therein can be seen as an *iteration* of the BFS algorithm, which produces a new *search level* consisting of all the newly generated successors. Subsequently, these states without the duplicates make up the new *Open* set, which can be subjected to a new iteration. Duplicate detection will happen whenever applicable, and the resulting LTS will contain no redundant states, i.e., states which are de facto equal to other states in the LTS. To make things clear when it comes to *partial* duplicate detection, we represent LTS generation explicitly by first constructing a generation tree of nodes. Figure 1 depicts the generation of an LTS as the traversal of some system behaviour, using both BFS with and without a *Closed* set. Left and right of the behaviour are two trees, where the numbering of the nodes firstly indicates the search level in which the node is encountered, and secondly, in what order the nodes are encountered. The dotted lines visualise which system state each node maps to. Using a *Closed* set, all explored nodes remain in memory, therefore, in the left tree, once nodes 2.4 and 3.5 are visited, they are recognised as essentially being equal to nodes 2.3 and 1.1, respectively, via their associated system states. If we consider BFS without a *Closed* set, on the right of the figure, we observe partial duplicate detection. Once node 2.4 is encountered, it is recognised as being essentially equal to node 2.3, since 2.3 at that time resides in the *Open* set. However, later on, node 3.5 is not recognised as being equal to node 1.1, since the latter has not been kept in memory, and the generation continues endlessly. From a generation tree, an LTS is derived by creating a state for each node deemed unique. For the right tree in the figure, this results in redundant states.

For checking duplicates, in practice, often a hash table is used to quickly search the contents of *Closed*. Such a hash table uses a hash function $h : \mathscr{S} \rightarrow \mathbb{N}$ to store $h(s)$ for all $s \in Closed$. If h leads to collisions, that is there are $s, s' \in \mathscr{S}$ such that $s \neq s'$ and $h(s) = h(s')$, then *collision detection* is a method to recognise this, which usually means storing multiple states at the same index in a hash array.

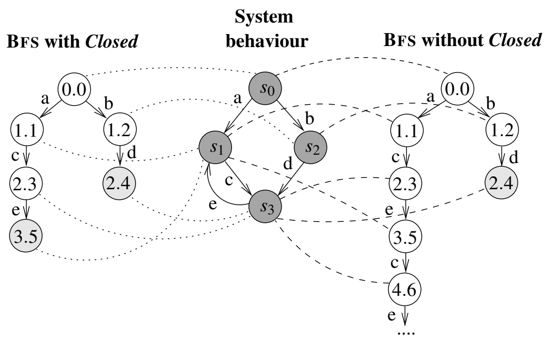


Fig. 1. BFS with and without a *Closed* set

3 BFS with State Space Caching

Caches have a fixed size, and a so-called *replacement strategy*, which determines which element should be removed next, whenever a new element needs to be added to an already full cache. There have been caching attempts with DFS [21, 6, 28, 29, 20, 16, 22, 1], and a few with BFS, which, however, are not guaranteed to terminate [30, 31].

As with related work on caching with DFS, we intend to restrict the growth of the *Closed* set. This, usually, has a negative effect on two other aspects of LTS generation, namely the execution time and the size of the output LTS. Unlike most previous work, besides the fact that we focus on BFS, we also try to minimise this negative effect. More precisely, we aim at satisfying three criteria: (a) The approach should be independent of any modelling paradigm, that is, it should not rely on analysis of the specification from which the LTS is derived, e.g., by using techniques such as static analysis, as in [1]; (b) The resulting LTS should contain as few duplicates as possible; restriction of the growth of the *Closed* set, however, tends to deteriorate duplicate detection, unless the LTS has a tree structure, or we are able to store exactly the right states in *Closed*, which would require prior knowledge of the LTS structure; (c) Termination of the generation algorithm must be guaranteed. This is not trivial for BFS, and possibly the main reason that few attempts exist of partial duplicate detection with BFS; as regards DFS, cycle detection by keeping the DFS stack in memory suffices to guarantee termination.

Without knowledge of the LTS structure, how can we decide which states to store and which to ignore? Many attempts have been made in the past to predict state space structures, e.g., [1, 30, 7, 26], but theoretically, any state may lead to any other state. Some have observed that, although LTSs may have any conceivable structure, the ones stemming from real specifications, which form our main target, seem to share some structural properties [30, 7]. Later, we return to this. First, we determine under which conditions the partial storage of explored states in the *Closed* set does not remove the termination guarantee of BFS.

3.1 Partial Storage of Explored States and Termination

Let us consider the relation between partial storage of explored states and termination of BFS. Earlier approaches using BFS all seem to be probabilistic in this respect; termination is at best highly likely [31, 30, 7]. Consider the right search tree from Figure 1, now with $Closed = \{1.1\}$. Note that this is sufficient to generate the LTS without any redundancy. But, focusing on termination, there are more possibilities; consider $Closed = \{2.3\}$. Even though node 3.5 is not recognised as identical to node 1.1, leading to redundancy in the LTS, the generation will terminate, as node 4.6 will be recognised as identical to the, stored, node 2.3. In other words, the fact that we store search level 2 means that the algorithm terminates. [1] calls a set of vertices in an automaton which ensures termination if states associated with them are stored a *covering set*. Similarly, we call a set of states a covering set if stored related nodes ensure termination. Periodically storing levels seems sufficient to always have a represented covering set in memory. In those cases where a state s is explored again, some of its 'descendants', either immediate or remote successors, will be recognised as part of a stored level, hence the redundant work resulting from the detection failure at s is finite. The levels should be stored *completely*, otherwise redundant traces may never be recognised as such (more precisely, levels need to be stored without any internal redundancy: in the example tree, note that node 2.4 does not need to be stored, as it is identical to node 2.3.). We call these stored levels *snapshots*.

Periodically storing levels allows us the construction of a covering set on-the-fly, as long as there is no bound on the number of snapshots in memory. We investigate this setup in more detail in Section 3.2. However, we are also interested in bounding the number of snapshots in memory, since it allows us to be more rigorous at removing states. Algorithm 1 shows this technique, which we call 'BFS with Snapshots' (BFSWS), where a sampling function f is used to determine at which levels to make snapshots, and n is the maximum number of snapshots in memory. When $n \longrightarrow \infty$, we are able to store an unbounded number of snapshots.

It is important to note that although duplicates are removed from *Next* before the new horizon is created, this is not done when making a new snapshot, in order to keep the levels complete. As explained in Section 4, though, storage of states in snapshots can be implemented such that duplicate occurrences cost little extra memory to keep. Next,

Algorithm 1 BFS with Snapshots

```
Require: Sampling function $f : \mathbb{N} \longrightarrow \mathbb{B}$, number of snapshots $n$
  procedure BFSWS($s_0$)
    $i, j \leftarrow 0$, $Open \leftarrow \{s_0\}$, $S_0, \ldots, S_{n-1} \leftarrow \emptyset$        {Initial state added to horizon}
    $S_j \leftarrow Open$                                    {First snapshot contains initial state}
    while $Open \neq \emptyset$ do                          {Repeat until there are no more states to explore}
      $i \leftarrow i+1$, $Next \leftarrow \emptyset$                          {The next level $(i+1)$ is currently empty}
      for all $s \in Open$ do                                  {Explore all states in the horizon}
        $Next \leftarrow Next \cup \{s' \mid \exists \ell.(s, \ell, s') \in en(s)\}$
      $Open \leftarrow Next \setminus \bigcup_{k=0}^{n-1} S_k$                          {Add new states to horizon}
      if $f(i)$ then $j \leftarrow j+1 \bmod n$, $S_j \leftarrow Next$                {Should this level be sampled?}
```

Lemma 1 shows under which conditions we can guarantee termination of BFSWS even if the number of snapshots is limited. We prove that BFSWS terminates as long as the sampling period, i.e., the number of levels between the taking of snapshots, increases along the search. From f, we can derive a function p_f as follows:

- $p_f(0) = 0$, (as we assume $f(0)$ is true)
- $p_f(i) = d$, with $i, d \in \mathbb{N}, i, d > 0$, $f(p_f(0) + \ldots + p_f(i-1) + d) \wedge \forall 0 < d' < d. \neg f(p_f(0) + \ldots + p_f(i-1) + d')$

In words, p_f expresses the subsequent sampling periods observable in f. Now, we need to prove that BFSWS is terminating for finite LTSs if p_f is increasing.

Lemma 1. BFSWS *of a finite* LTS *with a finite number of snapshots $n > 0$, is terminating if p_f is increasing.*

Proof. Let us consider a cycle of size k in an LTS. In a generation tree of BFSWS without duplicate detection, this cycle will be generated infinitely often, leading to $s_0, s_1, \ldots, s_{k-1}, s_k \ldots, s_{2k-1}, s_{2k}, \ldots$ etc.[1], with $\forall 0 \leq i \leq k-1, \forall j \in \mathbb{N}. s_i = s_{i+j \cdot k}$. We need to prove that by taking n snapshots, with p_f increasing, the cycle will be detected. First, we consider the case $n = 1$. Other cases follow from this.

Without loss of generality, we say that the first snapshot including a state from the cycle is taken while traversing the cycle for the first time. We call this state $\hat{s}_0 = s_d$, with $0 \leq d \leq k-1$. Let us first consider a p_f with $\forall i \in \mathbb{N}. p_f(i) = p$, and $p < k$. It follows that for subsequent snapshots $\hat{s}_i = s_{d+i \cdot p}$, with $i \in \mathbb{N}$. Observe that with $\hat{s}_0$, duplicate detection will succeed when reaching state s_{d+k}. But, since $p < k$, we have $s_{d+p} \rightarrow^+ s_{d+k}$, i.e., $\hat{s}_1 \rightarrow^+ s_{d+k}$. Because $n = 1$, we lose $\hat{s}_0$ after creating $\hat{s}_1$, hence there is no duplicate detection when we reach s_{d+k}. Similarly, with $\hat{s}_1$, duplicate detection can happen when reaching s_{d+p+k}, but $\hat{s}_2 \rightarrow^+ s_{d+p+k}$. In general, with $\hat{s}_i$, detection may happen at state $s_{d+i \cdot p+k}$, but $\hat{s}_{i+1} \rightarrow^+ s_{d+i \cdot p+k}$. However, if $p \geq k$, we have with $\hat{s}_i$ that $s_{d+i \cdot p+k} \rightarrow^* s_{d+(i+1) \cdot p}$, i.e., $s_{d+i \cdot p+k} \rightarrow^* \hat{s}_{i+1}$, so the next snapshot would be created some time after the exploration of $s_{d+i \cdot p+k}$, but when $s_{d+i \cdot p+k}$ is reached, duplicate detection takes place and the cycle traversal is terminated. If p_f is increasing, it follows that there exists $i \in \mathbb{N}$ such that $p_f(i) \geq k$, hence BFSWS can, in that case, deal with any cycle of arbitrary size. The case of BFSWS with $n > 1$ is a generalisation of case $n = 1$. For a p_f with $\forall i \in \mathbb{N}. p_f(i) = p$, if $n \cdot p < k$, for all $\hat{s}_i$, detection may happen at state $s_{d+i \cdot p+k}$, but $s_{d+(i+n) \cdot p} \rightarrow^+ s_{d+i \cdot p+k}$, i.e., $\hat{s}_{i+n} \rightarrow^+ s_{d+i \cdot p+k}$, and $\hat{s}_{i+n}$ replaces $\hat{s}_i$. If $n \cdot p \geq k$, then for $\hat{s}_i$, $s_{d+i \cdot p+k} \rightarrow^* s_{d+(i+n) \cdot p}$, i.e., $s_{d+i \cdot p+k} \rightarrow^* \hat{s}_{i+n}$, hence duplicate detection will take place at $s_{d+i \cdot p+k}$. If p_f is increasing, clearly BFSWS with $n > 1$ also terminates.

BFSWS is guaranteed to terminate if the sampling period is bigger than the size of the largest cycle in the LTS. Since we do not know this size a priori, constantly increasing the period ensures that eventually, the sampling period will be big enough. Now that we know under which conditions BFSWS always terminates, we can look at additional techniques to minimise the amount of redundant work, in order to keep the LTSs on disk as small as possible, and the execution time.

[1] As we enumerate the states of the cycle here, s_0 is not necessarily the initial state of the LTS.

3.2 Maximising the Efficiency of Partial Duplicate Detection

BFS With Snapshot Caches As explained in Section 1, a cache may contain a finite number of elements. For BFSWS, we can use a cache to store snapshots, as opposed to individual states, which is more common in related work. By choosing complete snapshots as elements, BFSWS is guaranteed to terminate, but it is impossible to enforce a fixed size of the cache. This seems at odds with the principle of a cache, but the caches of webbrowsers work in a somewhat similar manner; such a cache must always contain complete files, not parts of files, even though the files vary in size. For state space caching, this does not cause real problems, since in practice it shows that the gain in memory is still considerable. The size, together with the replacement strategy, which dictates which element to remove in case the cache is full, typically defines a cache. To this, we add the sampling function, which deals with the input of elements, resulting in the following definition.

Definition 2 (state space generation cache). *A* state space generation cache C_i *is a triple* (f_i, R_i, n_i)*, with* $f_i : \mathbb{N} \to \mathbb{B}$ *the sampling function,* $R_i : 2^{2^{\mathscr{S}} \times 2^{\mathbb{N}}} \to 2^{\mathscr{S}}$ *the replacement function, taking a set of snapshots together with meta-data about the snapshots (in the form of natural numbers), and returning a snapshot to be removed next, and* n_i *the maximal number of snapshots in the cache.*

Figure 2 illustrates a stream hierarchy of state space generation caches, in a way in which also in hardware architectures, multiple caches can be linked together. The f_i of a cache C_i decides which snapshots to accept for storage, n_i is the maximum number of snapshots it contains, and R_i is the replacement strategy. The removal of a snapshot from C_i leads to the input of a snapshot in C_{i+1}, that is, if f_{i+1} accepts it, etc. In general, R_i computes the cost of every snapshot in the cache based on a cost function $c : 2^{\mathbb{N}} \to \mathbb{N}$, and picks the snapshot with the lowest cost for the next removal. The cost function can use any accumulated data during the generation, e.g., (a) size of the snapshot, (b) snapshot level number, i.e., the time the snapshot was created, (c) the last time duplicate detection succeeded due to the snapshot, and (d) hit ratio of the snapshot, i.e., how many times duplicate detection succeeded due to the snapshot.

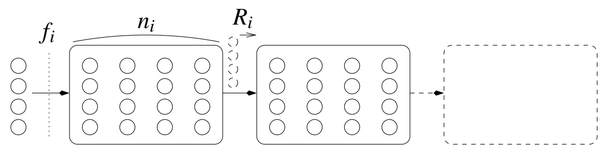


Fig. 2. A stream hierarchy of state space generation caches

This machinery allows for a wide range of configurations, since it involves at least four new parameters: the number of caches, and per cache a sampling function, a size, and a replacement strategy. Next, we explain which configurations make sense for the generation of LTSs stemming from real specifications.

Exploiting Transition Locality Related attempts to search LTSs with partial duplicate detection often use a notion called *transition locality* l of states [30, 7, 25]. This is a property of an LTS together with a corresponding traversal tree of *traditional* BFS, i.e., without caching. It expresses the biggest distance, measured in levels, between any two nodes which are considered equal. In Figure 1, for the LTS together with the left tree, $l = 2$, as nodes 1.1 and 3.5 are considered equal. It has been claimed [30, 7, 25] that l is extremely low for LTSs resulting from real protocol specifications. This can be exploited by only keeping the last l levels of the tree in memory, which yields a version of BFS called *frontier search* [25][2]. Algorithm 1 describes frontier search if $f(i)$ equals *true* for all $i \in \mathbb{N}$, and $n = l$. Termination is only guaranteed if $n \geq l$, $n = l$ being ideal, since it saves as much memory as possible with this technique. However, in practice, l is not known before traversal of the LTS, therefore termination cannot be guaranteed. In addition, our experience points out that, although the majority of equalities concern nodes which are indeed very close to each other, there are usually also some equal nodes which are much further removed from each other, e.g., more than half the total depth of the tree. This differs from the results reported by [30, 7, 25], possibly because we look at a large set of specifications of varying types of systems. In case l is close to the full depth of the tree, which is likely to happen for e.g., cyclic processes, one cannot gain that much memory.

However, the fact remains that often, most nodes which are considered equal are very close to each other in the tree. We choose to exploit this by setting up a stream of caches, the first one sampling frequently, and subsequent ones sampling less often, as identification with 'older' nodes is less likely to happen. Then, for the first cache, we can choose $n_1 < l$, which not only removes the necessity to know l a priori, but for LTSs where l is large, we can also save more memory compared to frontier search and related techniques. Related to this, Tronci et al. [30] deal with distances larger than l in a probabilistic way, while we guarantee eventual detection, hence termination. Streams of caches can be set up in many different ways; we consider two, the results of which will be discussed in the next section:

1. The first cache samples often, in fixed periods, and a second cache employs a sampling function with increasing period, i.e., conform Lemma 1. We call this setup *Frontier Safety Net*, since in addition to having a frontier cache, we also have some safety nets to fall back on.
2. Initially, there is one cache, C_1, sampling often, in fixed periods. As soon as the cache is full, another one, C_2, with the same setup is created and connected behind C_1. Whenever this cache is full, a third one is created, etc. If the sampling period of the caches is not 0, i.e., not every level is sampled, then the further down the stream, the fewer levels are accepted by a cache, hence the longer it takes before the cache is full. Since the number of snapshots allowed in memory is not bounded, this setup guarantees termination of the algorithm. We call this setup *Pebble Search*, since the distribution of the snapshots over a generation tree resembles the waves produced

[2] In the original setting of Artificial Intelligence, a predecessor function is assumed to be available to have access to the recent predecessors. Lacking such a function in on-the-fly model checking, the last l levels should be stored [11].

by a pebble when dropped in a pool, i.e., the further away from the point of impact, i.e., the horizon, the further the distance between waves.

The Backtracking Set In our experience, Frontier Safety Net and Pebble Search lead to good reductions of the memory use, ranging from 50% to sometimes less than 10%, as will be shown in the next section. However, if there are many duplicates to be detected at a distance greater than $n_1 \times p_1$, with p_1 the constant sampling period of cache C_1, then this tends to lead to a big increase of redundant work, negatively affecting both the execution time and the output LTS. Consider Figure 3, where in the *Open* set, node 0 is equal to node 1, which has been explored before, but removed from memory by now. Therefore, node 0 will be re-explored, leading to nodes equal to nodes 2 and 3, the successors of node 1. Since nodes 2 and 3 are also removed, these new nodes are explored as well, and their successors are finally identified as equal to the successors of nodes 2 and 3, since these are present in a snapshot, i.e., the grey bar. All in all, failure to recognise that node 0 has essentially been seen before leads to the traversal of 6 redundant transitions. In state space traversal, the traversal of transitions is the most time-consuming operation, therefore this redundant work has a real impact on the overall execution time. In both our setups, the older the levels, the fewer remain in memory, hence, the larger the distance between equal nodes, the more likely it is that failure of duplicate detection leads to the exploration of many nodes before a snapshot is 'reached'. On the one hand, only keeping a few very old levels makes sense, since new nodes do not often refer back to very old nodes. On the other hand, on those occasions where they do, we often obtain a significant amount of redundant work. Let us call the branching factor, i.e., the average number of successors of a state in the LTS, b, and the distance between an old node and the nearest subsequent snapshot d, then every detection failure leads to approximately $\sum_{i=1}^{d} b^i$ additional traversals. In practice, it turns out that if a much older node, i.e., older than only a few levels ago, is referred to again once, then it tends to be referred to several times more later on, in our case each time leading to *at least* $\sum_{i=1}^{d} b^i$ extra traversals (in subsequent re-explorations, the nearest snapshot may well have been removed from memory, thereby increasing the distance to the next snapshot).[3]

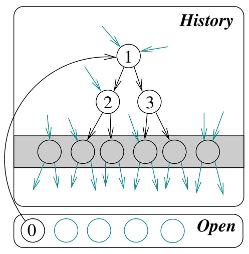


Fig. 3. Duplicate work

These nodes seem to represent states which are very common for the specified systems, imagine e.g., the specification of a car; there are many ways to use the car, but eventually you return to the state representing 'off'. By keeping the node representing 'off' in memory, we can avoid a lot of redundant work. Recognising these important nodes is very hard, but we propose a mechanism for BFSWS which can guess which nodes are important. Every time a node is revisited, the mechanism gets closer to discovering this revisiting. For this we introduce an extra, unbounded, set of nodes to be kept in memory, the *Backtrack* set. While traversing, this set is filled with nodes by

[3] [20] reports that no relation is found between the number of previous visits to a state and the likelihood that it will be visited again in the future. We found that revisits are likely to states which had 'late' revisits, i.e., revisits many levels after the first visit.

following two rules, where very old snapshots are defined as 'not in cache C_1': 1) given a node N in the *Open* set with $\mid succ(N) \mid > 1$, if there exists a snapshot S_i not in C_1 such that for all $N' \in succ(N)$, there exists $N'' \in S_i$ with $N' = N''$ (i.e., N' is considered equal to N''), then we add N to *Backtrack*, and 2) given a node N in the *Open* set with $\mid succ(N) \mid > 1$, if for all $N' \in succ(N)$, there exists $N'' \in Backtrack$ with $N' = N''$, then we add N to *Backtrack*. The first rule states that if all the successors of a node are detected as duplicates due to a single very old snapshot, then it is very likely that we have explored their parent before. The more successors the node has, the more likely this is, hence we exclude the case here of a single successor. Failure to detect duplicates which only have one successor does not directly lead to much redundant work anyway. The second rule is a continuation of this: if all the successors of a node are suspected of having been re-explored, then we suspect this node as well. With this technique, we bound and lessen the amount of redundant work with each revisit of a node; the n^{th} revisit leads to $\sum_{i=0}^{|d-(n-1)|} b^i - 1$ extra traversals. Practice shows that this learning mechanism is very successful, as seen next. By only keeping a few extra nodes in memory according to these rules, we sometimes reduce the amount of redundant work considerably.

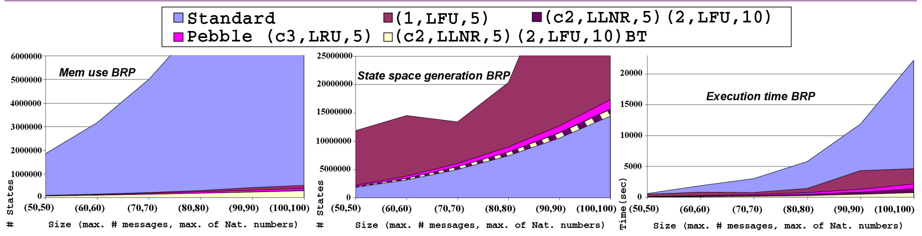
4 Implementation, Caching Setups, and Experiments

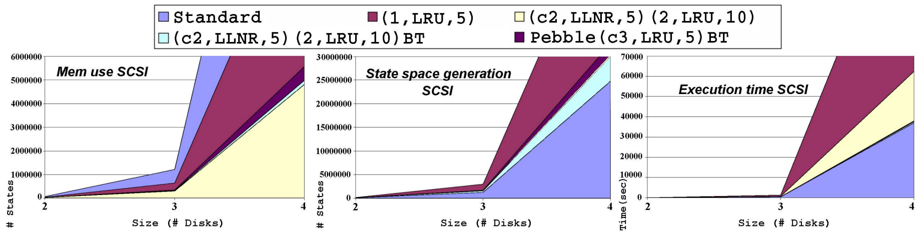
We built a generic, application-independent implementation of the caching machinery using the OPEN/CÆSAR [12] environment of the CADP toolbox [13], which provides various primitives for state space storage and exploration (hash tables, edge lists, stacks, etc.). The cache library allows to define caches containing a fixed number of elements, either states or snapshots, each one being possibly assorted with user-defined information allowing, e.g., to calculate the cost associated to the element. The replacement strategy used by a cache can be user provided, or selected among five built-in strategies: least/most recently used (LRU/MRU), least/most frequently used (LFU/MFU), and random (RND). The elements of a cache are stored in a balanced heap equipped with a hash table in order to allow fast retrievals of the lowest-cost element and fast searches of elements. A special primitive retrieves the last element replaced after an insertion took place in an already full cache; this allows to manage hierarchical caches (organized, e.g., as trees) by retrieving an element from one cache and inserting it into another.

The most basic usage of the cache library is for storing visited states, assorted with their id's, during a DFS traversal of the state space. A more complex usage is made by the BFSWS approach, where elements stored in the cache are snapshots (lists of states) but the searches carried out for duplicate detection concern individual states. To reduce memory consumption, states belonging to the set of snapshots currently present in a cache are stored uniquely and referenced through pointers; state deletion is done efficiently using a reference counting scheme borrowed from garbage collection [23]. Next, we study the performance of several BFS and DFS setups experimentally. For this, we used around 35 LTSs from the VLTS benchmark suite [32] stemming from real, industrial case studies, and also several communication protocols [8]. The experiments were run on a LINUX machine with a 2.2GHz CPU and 1 GB memory.

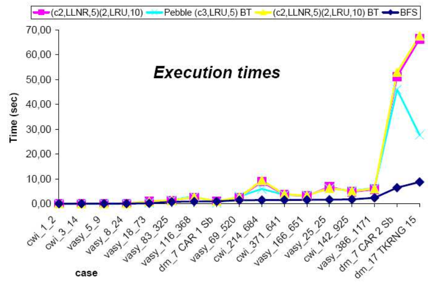
4.1 BFS Experiments

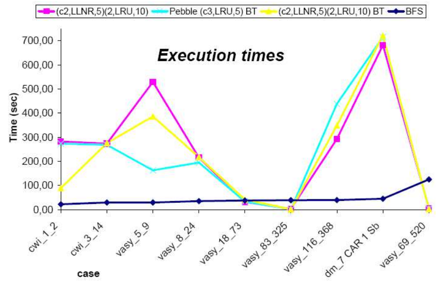
Here, we show the results of a representative selection of cases, generated by standard BFS and by BFSWS using some of the most successful cache setups. Caches are described as triples, a sampling function which increases its period by *n* after every sampling being written as *n*, a function with constant period *n* being written as *cn*, LLNR being a replacement function based on lowest level number, and BT indicating the backtracking mechanism. The top three graphs visualise the results on several instances of the Bounded Retransmission Protocol (BRP), varying in both the size of messages and the number of retransmissions. Here, the techniques are extremely effective, allowing not only to reduce the memory use drastically, but also to make the generation much faster. This is due to the hash table being very small, which speeds up duplicate detection. Usually, this gain is countered by failed detections, leading to more (time consuming) transition traversals, but here such failures hardly occur. Additional tests showed that we could generate the LTS for BRP $\langle 300, 300\rangle$, consisting of more than $410,000,000$ states, in 37 hours with $(c2, \text{LLNR}, 5)(2, \text{LFU}, 10)$, and in 22 hours with backtracking. The bottom three graphs show the generation results for the SCSI-2 bus arbitration protocol, varying the number of competing disks. Here, memory use can be reduced to 20% compared to standard BFS, using a Frontier Safety Net with backtracking. Moreover, both backtracking setups show practically no increase in execution time, and the number of redundant states produced is reasonable.

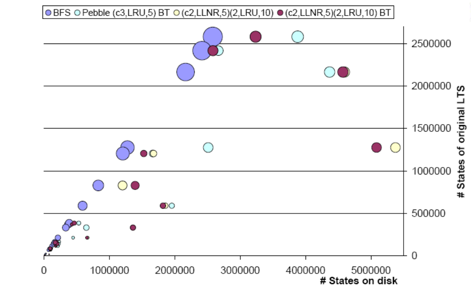


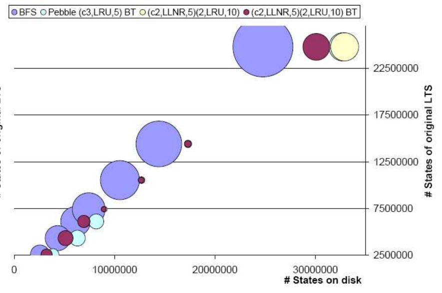


The graphs below compare execution times and state generation of BFS and 3 sampling setups. Execution times of the setups are often much longer than for BFS, due to using more complex data structures, and the redundant work. However, this effect becomes mainly apparent with small examples. The bottom two graphs relate output LTS sizes with the original sizes, indicating the number of states in memory by the size of the bubbles. Bubbles on the same horizontal line relate to the same case. Pebble Search with backtracking sometimes produces remarkably smaller LTSs than the other methods, and faster, suggesting that for these cases, keeping snapshots which are more evenly distributed over the history of the generation pays off. In Frontier Safety Net, the second cache samples more and more infrequently, eventually leaving a big 'hole' in the stored history. There is still room for improvement, though, which we plan as future work. For most cases, memory use can be reduced to about 30% using the sampling mechanism. The graphs show that either backtracking has a very positive effect (e.g. in execution time), or no effect; a negative effect hardly ever occurs. This makes backtracking a useful feature to enable by default in the BFSWS state space generator.



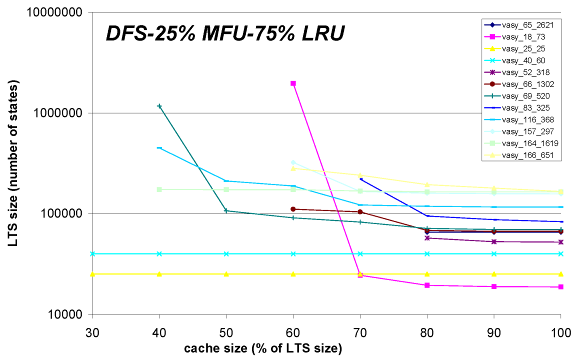






4.2 DFS with caches

Our generic cache machinery can also be used in conjunction with other graph traversals in order to reduce the amount of memory needed for generating an LTS. The figure below shows the behaviour of DFS equipped with cycle detection (by searching states on the DFS stack) and a hierarchical $\langle$MFU, LRU$\rangle$ cache, executed on a subset of the VLTS benchmark suite. For each example, the figure gives the minimal size of the cache yielding an output LTS of size at most double w.r.t. the input LTS. Among the five built-in replacement strategies, LRU performs best on all examples (reducing the cache size down to 40% of the number of states), followed closely by MFU and RND. LRU is close to the strategy removing the oldest states, which was rated best among the



strategies analysed in [20]. If we split the cache into two cascading subcaches of varying sizes and different replacement strategies, the best cache size reduction (down to 30% of the number of states) was achieved by using an MFU or RND subcache followed by an LRU subcache (of about 25% of the whole cache). Overall performance (see the figure above) was further improved by increasing the LRU subcache to 75%. Compared to BFSWS, the success of DFS setups differs a lot from one case to another. Moreover, a difficulty with 'fixed size' DFS caching is to determine the right size of the cache, which, in order to be effective, should lie between $30-60\%$ of the (*a priori* unknown) LTS size; BFSWS, on the other hand, simply takes whatever memory it needs.

5 Related Work

As mentioned in the introduction, existing approaches for state space generation can roughly be divided in two classes. In the second class, where exhaustiveness is not guaranteed, hashing without collision detection is often used, which is a way to ensure that the hash table stores a fixed number of states. Concerning collisions, [21, 6] assume that whenever a collision happens in the hash table, the new state has already been visited. In [6], this is used for a nested DFS to search for errors in LTSs. Collision avoidance in this way guarantees termination. [30, 7] take the opposite approach concerning this. They fully depend on LTSs of protocols having small localities. They avoid collision detection, by removing a previously stored state if there is a hash collision with a new state, arguing that in most cases this means removing a state beyond the locality region. Termination is handled by stopping the search once the collision rate is sufficiently high, or a maximum search depth has been reached. In [31], a fixed size cache and *Open* set is used for a probabilistic, randomised BFS, where states are replaced at random. These last two cases report memory savings of 40% on average, the first case achieving this partly by keeping the *Open* set on disk. In [28], the probability of failures is reduced by using open addressing and t-limited lookups and insertions in the hash table, where a hash function provides not one position for a state in the hash table, but a sequence of t possible locations. In addition to this, [29] includes the level number of a state in a BFS in calculating its omission probability.

The other class, guaranteeing exhaustiveness, includes most state space caching work, which has been done for DFS [20, 16, 22, 15]. In [20], several replacement strategies are used for a single LTS, and the conclusion is that the one selecting the oldest states is the best. [22] continue on this, believing the conclusion to be a random strategy. [15] reinvestigates this, employing many other strategies on many examples, some of them from practical cases. In those cases, a proposed strategy called stratified caching with random replacement performs best. Here, *strata* are created, very similar to our snapshots. The states of certain strata, however, are candidates for *removal* from memory, as opposed to storing in memory. Caching is combined with static analysis and partial order reduction in [16]. Our setting allows hierarchies of caches, and they can be used to store snapshots in case of BFS. In [1], an unbounded set of states is stored based on a storing strategy, usually based on static analysis. They claim that up to 90% can be saved in memory use, but, as reported by [18], this leads to an extensive amount of redundant work. In [18], a hash table is initially used in a traditional manner, until it fills up the whole memory, at which point older states are moved to secondary storage.

The number of lookups on disk is reduced by using a Bloom filter. Instead of trying to use memory in a smarter way, they want the generation to be able to continue once the memory is filled. We experience that trying to avoid a big *Closed* set pays off in terms of execution time, since the hash table is kept small. However, our BFS method may still run out of memory. It could be interesting to look at a combination of the two.

In Artificial Intelligence (AI), connecting to directed model checking, multiple methods are presented to bound the memory use of exhaustive search algorithms, e.g., *IDA** [24], *MA** [4], and extensions *IE* and *SMA** [27]. As is common in this field, the algorithms employ a cost function, mapping states to costs. For memory-bounding, this function is used to decide which states to remove from memory. The cost function provides knowledge of the LTS structure a priori, and its main purpose is to guide the search to a goal state. In this paper, we are neither concerned with a subset of goal states, nor have any structural knowledge. It is, however, possible to incorporate the AI algorithms in our framework, like DFS and BFS, in order to obtain more memory efficient variants.

6 Conclusion and Future Work

We presented generic machinery for hierarchical, adaptive state space caching, implemented using the OPEN/CÆSAR environment [12] of the CADP toolbox [13]. This machinery can be used in a very flexible manner for state space generation, in conjunction with DFS and BFS traversals. Our algorithm BFSWS is exhaustive and guaranteed to terminate, and its behaviour can be finely tuned using the caching and learning mechanisms introduced in Section 3. These techniques compete favourably with earlier ones, such as hashing without collision detection and frontier search, which only concern LTS search (and not generation), lack termination or exhaustiveness (or both), or are dedicated to LTSs with small localities. The learning mechanism for BFSWS strongly reduces the amount of redundant work, and is also able to speed up the generation.

As future work, we will study other BFSWS configurations on further examples (e.g., the BEEM benchmark [26]), and try to design additional mechanisms to deal more efficiently with different LTS structures. Secondly, we plan to adapt the machinery for distributed state space generation [14]. Finally, we want to investigate its use in conjunction with on-the-fly LTS reduction modulo τ-confluence and branching bisimulation.

References

1. G. Behrmann, K.G. Larsen, and R. Pélanek. To Store or Not To Store. In *CAV*, volume 2725 of *LNCS*, pages 433–445, 2003.
2. S.C.C. Blom, J.R. Calamé, B. Lisser, S. Orzan, J. Pang, J.C. van de Pol, M. Torabi Dashti, and A.J. Wijs. Distributed Analysis with μCRL: A Compendium of Case Studies. In *TACAS*, volume 4424 of *LNCS*, pages 683–689, 2007.
3. S.C.C. Blom and S. Orzan. Distributed State Space Minimization. *STTT*, 7(3):280–291, 2005.
4. P.P. Chakrabarti, S. Ghose, A. Acharya, and S.C. de Sarkas. Heuristic Search in Restricted Memory. *Artificial Intelligence*, 41(2):197–222, 1989.
5. E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, 1999.
6. C. Courcoubetis, M.Y. Vardi, P. Wolper, and M. Yannakakis. Memory-Efficient Algorithms for the Verification of Temporal Properties. *FMSD*, 1(2/3):275–288, 1992.

7. G. Della Penna, B. Intrigila, E. Tronci, and M. Venturini Zilli. Exploiting Transition Locality in the Disk Based Murphi Verifier. In *FMCAD*, volume 2517 of *LNCS*, pages 202–219, 2002.
8. CADP demos. CADP online demo examples. `http://www.inrialpes.fr/vasy/cadp/demos.html`, 2008. Last visited on October 7, 2008.
9. S. Edelkamp and S. Jabbar. Real-Time Model Checking on Secondary Storage. In *MoChArt*, volume 4428 of *LNAI*, pages 68–84, 2007.
10. S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *STTT*, 5:247 – 267, 2004.
11. S. Edelkamp, V. Schuppan, D. Bošnački, A.J. Wijs, A. Fehnker, and H. Aljazzar. Survey on Directed Model Checking. In *MoChArt*, volume 5348 of *LNAI*, 2009. To Appear.
12. H. Garavel. OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing. In *TACAS*, volume 1384 of *LNCS*, pages 68–84, 1998.
13. H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes. In *CAV*, volume 4590 of *LNCS*, pages 158–163, 2007.
14. H. Garavel, R. Mateescu, and I. Smarandache. Parallel State Space Construction for Model-Checking. In *SPIN*, volume 2057 of *LNCS*, pages 217–234, 2001.
15. J. Geldenhuys. State Caching Reconsidered. In *SPIN*, volume 2989 of *LNCS*, pages 23–38, 2004.
16. P. Godefroid, G.J. Holzmann, and D. Pirottin. State-Space Caching Revisited. *FMSD*, 7(3):227–241, 1995.
17. P. Godefroid and P. Wolper. Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties. In *CAV*, volume 575 of *LNCS*, pages 410–429, 1991.
18. M. Hammer and M. Weber. "To Store or Not To Store" Reloaded: Reclaiming Memory on Demand. In *FMICS*, volume 4346 of *LNCS*, pages 51–66, 2006.
19. J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, fourth edition, 2006.
20. G.J. Holzmann. Automated Protocol Validation in Argos, assertion proving and scatter searching. *IEEE Trans. on Software Engineering*, 13(6):683–696, 1987.
21. G.J. Holzmann. An Improved Protocol Reachability Analysis Technique. *Software - Practice and Experience*, 18(2):137–161, 1988.
22. C. Jard and T. Jéron. Bounded-memory Agorithms for Verification On-the-fly. In *CAV*, volume 575 of *LNCS*, pages 192–202, 1991.
23. Donald E. Knuth. *The Art of Computer Programming — Volume III: Sorting and Searching*. Computer Science and Information Processing. Addison-Wesley, 1973.
24. R. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.
25. R. Korf, W. Zhang, I. Thayer, and H. Hohwald. Frontier search. *Journal of the ACM*, 52(5):715–748, 2005.
26. R. Pelánek. Properties of state spaces and their applications. *STTT*, 10(5):443–454, 2008.
27. S. Russell. Efficient memory-bounded search methods. In *ECAI*, pages 1–5. Wiley, 1992.
28. U. Stern and D.L. Dill. Combining State Space Caching and Hash Compaction. In *4. GI/ITG/GME Workshop*, pages 81–90, 1996.
29. U. Stern and D.L. Dill. A New Scheme for Memory-Efficient Probabilistic Verification. In *FORTE*, pages 333–348, 1996.
30. E. Tronci, G. Della Penna, B. Intrigila, and M. Venturini Zilli. Exploiting Transition Locality in Automatic Verification. In *CHARME*, volume 2144 of *LNCS*, pages 259–273, 2001.
31. E. Tronci, G. Della Penna, B. Intrigila, and M. Venturini Zilli. A Probabilistic Approach to Automatic Verification of Concurrent Systems. In *APSEC*, pages 317–324, 2001.
32. VLTS. The VLTS Benchmark Suite. `http://www.inrialpes.fr/vasy/cadp/resources/benchmark_bcg.html`, 2008. Last visited on June 6, 2008.