State Caching Reconsidered

Jaco Geldenhuys

Tampere University of Technology, Institute of Software Systems
PO Box 553, FIN-33101 Tampere, FINLAND
email: jaco@cs.tut.fi

Abstract. *State caching* makes the full exploration of large state spaces possible by storing only a subset of the reachable states. While memory requirements are limited, the time consumption can increase dramatically if the subset is too small. It is often claimed that state caching is effective when the cache is larger than between 33% and 50% of the total state space, and that random replacement of cached states is the best strategy. Both these ideas are re-investigated in this paper. In addition, the paper introduces a new technique, *stratified caching*, that reduces time consumption by placing an upper bound on the extra work caused by state caching. This, and a variety of other strategies are evaluated for random graphs and graphs based on actual verification models. Measurements made with SPIN are presented.

1 Introduction

Model checking by explicit state enumeration has become increasingly successful in the last decade, but suffers from the well-known state space explosion problem. Techniques for palliating the problem abound, but once the size of a model crosses a certain line, the only hope is a partial search of the state space. While such probabilistic approaches are valuable in detecting violations of the specification, they cannot guarantee correctness; when possible, a full exploration of the state space is, of course, preferable.

This paper focuses on *state caching*, one of the earliest techniques to deal with state space explosion. During state space exploration the reached states are stored in a hash table or similar data structure. When a previously visited state is reached again, it is found among the stored states and does not have to be re-explored. When a new state is generated and the state store is full (i.e., available memory has been exhausted), an already stored state is selected and discarded to make room for the new state. By replacing an old state, the model checker commits itself to re-investigate the state, should it be reached again. This may entail re-doing previous work, but full exploration of the state space is guaranteed.

While this approach makes effective use of the available memory, the running time may increase dramatically if too few states are cached. The received wisdom is that state caching is effective when the cache is larger than somewhere between 33% and 50% of the total state space, and that random replacement of states

is the best strategy. These ideas are re-investigated and found to be somewhat misleading.

We propose a new replacement strategy called *stratified caching*. Broadly speaking, it places an upper limit on the amount of redundant work that is performed when an already visited but replaced state is reached again. It does this by only replacing states at predetermined depths, or *strata*, hence the name.

Section 2 examines how state caching has been presented in the literature, and in Section 3 stratified caching is introduced and discussed. Section 4 evaluates a range of caching strategies for both random and actual state graphs, some initial results for a single model are reported in Section 5, and, finally, conclusions are presented in Section 6.

2 State Space Exploration and State Caching

During state space exploration the reached states are stored in a table. When a previously visited state is reached again, it is found in the table and does not have to be re-explored. When a new state is generated and the table is full (i.e., the available memory has been exhausted), there are three possibilities:

A. Abandon the search, explaining that the memory has been exhausted.
B. Discard the new state and pretend that it has been seen before.
C. Select an old state and replace it with the new state.

It is not immediately clear why possibility A would ever be preferred over B or C. Of course, it is important to notify the user that the memory has been exhausted: she may wish to interrupt the exploration and investigate alternatives. It may be that the model is faulty in some way and that the memory exhaustion is a symptom of this. Or the user may wish to investigate a simplified model, alternative options for state exploration or other reduction strategies. But it seems sensible to always continue the search with either possibility B or C. However, the implementation of possibility B or C may require memory and time overheads which the user wishes to avoid in the "default" operating mode of the state space exploration tool. Memory considerations are usually not critical since, as we shall see, state caching works well even if the cache is slightly smaller than the state space. In other words, the memory overhead may not make any real difference to the user. Time overhead is another issue and it is difficult to say, in general, whether it is wise to use or not use state caching as a default.

Possibility B results in a partial exploration of the state space, and issues such as omission probability and coverage come into play. In this case, the search may yield false positives, claiming that the model satisfies the correctness specification when, in fact, it does not, while all violations of the correctness specification are valid. As indicated in the introduction, this paper looks only at full exploration.

Possibility C is known as *state caching*, the focus of this paper. By replacing the old state, the model checker commits itself to reinvestigate the old state should it be reached again. Although this may redo work that has been done before, it does not lead to incorrect results.

State caching works well when the cache is slightly smaller than the state space. If relatively few states are replaced, the probability of revisiting a replaced state is low and even if it happens, the state's successors will probably be found in the cache. As the cache grows smaller and smaller compared to the state space, the re-exploration of states grows more and more frequent, and when the cache is very small, the problem of redundant work becomes severe. Under these conditions, the probability of revisiting a replaced state, and having to revisit its successors and their successors is high. Furthermore, each state that is revisited is re-inserted in the cache (since the depth-first algorithm cannot tell that it is not a new state) and displaces another state in the cache, in this way making matters even worse.

Depth-first search is guaranteed to terminate when state caching is used as long as states on the depth-first search path are never replaced, and the cache is large enough. Because states on the depth-first search path cannot be replaced, it is possible that too small a cache can eventually "fill up" with such states, in which case the search cannot proceed and must terminate early. (For general graphs, state caching does not work at all for breadth-first search, and offers a limited improvement in performance for mixed depth- and breadth-first search strategies; in the rest of the paper only depth-first search is considered.)

2.1 State Caching in the Literature

The first discussion of state caching for model checking is by Holzmann [10]. As far as we are aware, there have been three sets of papers that report significant results about state caching.

- In [10] — which is an overview of a number of verification techniques and not an in-depth discussion of state caching — the author investigates state caching for a single model of 150000 states. States are replaced using "simple blind round robin selection", but it is not clear what data structure is used to store the states and whether all states are considered equally in the evaluation of this criterion. The conclusion of the paper is that a cache of roughly half the size of the state space can still provide acceptable performance.

In a later paper [11] (published after but written before [10]) Holzmann investigates five different replacement strategies as implemented in the *trace* tool, a forerunner of SPIN [12]. The strategies are based on replacing

H1. most frequently visited states;
H2. least frequently visited states;
H3. states in the currently largest class of states, where the class of a state is defined by the number of times it has been visited;
H4. oldest states (i.e., those states that have been in the cache longest); and
H5. states in the bottom half of the current search tree.

As before [10], no details about how states are stored, are given. This is significant, because the data structure affects the behaviour of the state cache with respect to redundant work, memory and time consumption. The strategies

are also not clearly defined: there is no indication of how to choose between possible candidate states in strategies H1 and H2, and the "bottom half" in strategy H5 can be interpreted in a number of different ways.

The strategies are investigated "for a range of medium sized protocols". Two examples are presented, one with 4523 and the other with 8139 unique states. The paper concludes that strategy H4 is consistently the fastest (resulting in a tolerable increase in running time) even though in one of the examples it performs the most unnecessary work by far (59% "double work" compared to a maximum of 0.5% by the other strategies). No conclusions about the minimum size of the state cache are reached. Confusingly, many future references refer to H4 as "random replacement", even though it is clearly not presented as such in the paper. Also, H5 is sometimes described as replacing the state corresponding to the smallest subtree of the depth-first tree; this may be its intention, but it is not how the strategy is defined in the paper.

- Jard and Jéron investigate state caching in [13] and in [14, 15]. Like the earlier work [10, 11], these papers do not focus on state caching *per se*. In the last two works, the authors generate random graphs that are explored using depth-first search and, based on the earlier findings [11], a state cache with random replacement. They report that, in a typical case, a cache using 40% of the normal memory yields 70% more visited states and a 50% increase in running time. In the best case, the cache size is reduced to 10% of the state space with only a 1% increase in the number of visited states.

- The effect of partial-order methods [8, 18, 21] on state caching is addressed by Godefroid, Holzmann, and Pirottin in [7]. *Sleep sets* [6] is a partial-order method that eliminates most of the interleaving of independent transitions without reducing the number of states. (The combination of sleep sets and *persistent sets* [8], which also reduces the number of states, is further investigated by Godefroid [9].) In [6] state caching (without sleep sets) is first investigated for four models of real-world protocols using the SPIN verifier [12]. The models have a transition/state ratio of roughly 3, and the authors report that the cache can be reduced to 33% to 50% the size of the state space. These findings confirm the general results of [10, 11].

  The details of the state caching can be determined fully because the authors have made their software publically available. States are stored in an open hash table with pointers to singly linked lists of states with the same hash value. A state is inserted by appending it to the linked list pointed to by its hash slot. After each insertion, a check is made to see if the state store is full. If so, an old state is selected and discarded to make room for the next insertion. Although the paper claims to use a random replacement strategy, it is clear from the code that this is not entirely accurate. The linked lists are scanned cyclically and within each list the state that has been in the cache longest, and therefore occurs towards the front of the list, is always chosen first.

  With sleep sets the performance of state caching improves dramatically. The transition/state ratio of one model decreases from 2.88 to 1.45 and a cache of about 25% the size of the state space suffices. For the other three models, the

use of sleeps sets reduces the ratio from 2.80, 3.54, and 2.58 to 1.12, 1.04, and 1.04, respectively. For these, the cache performs spectacularly well: a cache size of only 0.2% to 3% the size of the state space suffices for a complete exploration of the models. Unfortunately, the running time increases by a factor of between 2 and 4, but it is not clear whether this is caused by the implementation of the sleep sets or the state caching.

2.2 Other Work Related to State Caching

The performance of state caching using open addressing — also known as closed hashing — has also been investigated [5, 19], but, due to space constraints, this approach is not considered here. Other techniques that may improve the effectiveness of state caching have been suggested, sometimes in more general contexts. This includes the identification of states to replace preferentially [7, 10], heuristic state space exploration [4], probabilistic caching of states [3], and precomputation to enhance replacement strategies [17].

3 Stratified Caching

When a state cache contains only a few replaced states, the probability of revisiting a deleted state is small, the probability that the state has a deleted child is smaller, the probability that the child also has a deleted child is smaller still, and so on. As the number of replacements grows, these probabilities increase to the point where practically every deleted state has at least one deleted child, and most states have several. Revisiting a deleted state in this situation leads to a considerable amount of redundant work; not only do large subtrees require re-exploration, but each revisited state is also re-inserted in the cache, pushing out yet another state, and causing a cascade of revisits.

Stratified caching limits the redundant work by placing an upper bound on how much deeper than usual each branch of the depth-first tree needs to be explored, hereafter referred to as the *extra depth*. It does this by only replacing states at specified levels of the depth-first search tree. All states at a certain level form a "stratum" of states.

It is generally not possible to know how deep the depth-first search will go and how many strata there will be, so strata are classified as "available" or "unavailable" for replacement based on their level *modulo a certain number* m. When a stratified caching strategy specifies that all strata of level k modulo m are available for replacement, this means that the states at levels $k, k+m, k+2m, \ldots$ may be replaced while the states of other strata must remain in the cache. Figure 1 shows available strata boxed in gray for the $k = 1$, $m = 3$ case.

Assuming that the depth-first tree is deep enough, that the states of the state space are uniformly distributed over the different levels modulo m, and that the probability of a revisit is uniformly distributed over the states, the expected extra depth is at most $1/m$ and the maximum extra depth is 1. As the search progresses, an available state can be replaced by either another available state,

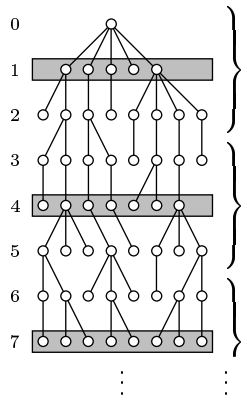

Fig. 1. Stratified caching for the $k = 1$, $m = 3$ case

in which case the expected extra depth remains constant, or by an unavailable state, in which case the expected extra depth decreases slightly. Unfortunately, because at most $1/m$ of the states are available for replacement, the cache is quickly exhausted (i.e., filled only with unreplaceable states) and the search must be aborted. Smaller values of m increase the number of available strata and therefore the fraction of replaceable states, but the minimum value for m is 2, meaning that at most $1/2$ of states can be available for replacement in this setting. Another way to increase the number of replaceable states is to increase the number of available strata within each modulo group. For example, if strata 2 and 3 modulo 5 are available for replacement, the expected extra depth is at most $3/5$ and the maximum extra depth is 2.

In this paper, the following approach is taken: Initially, $k = 1$ and $m = 2$. Once the available states are exhausted, all the states in the odd (available) strata are states currently on the depth-first search stack. (If a state from an odd stratum were not on the stack, it would have been available for replacement, but the cache has been exhausted and there are no more replaceable states.) Instead of aborting the search at this point, the value of m is doubled to 4, and k is changed to $1 \ldots 3$. This process may be repeated several times, as illustrated in Figure 2. After the nth doubling, $m = 2^{n+1}$, $k = 1 \ldots m-1$, the expected extra depth is at most $(m-1)/2$ and the maximum extra depth is $m-1$.

An idea similar to stratified caching has been described in [2]. There the authors investigate several heuristics that indicate whether or not a particular state should be stored at all. In fact, that work focuses on the minimal set of states that need to be stored, in other words, heuristics for selecting a covering set of vertices. Storing all states and replacing selectively replacing some holds the obvious advantage of avoiding a potentially significant amount of work, but it is also true that, for some models, storing only some of the states can improve performance even further.

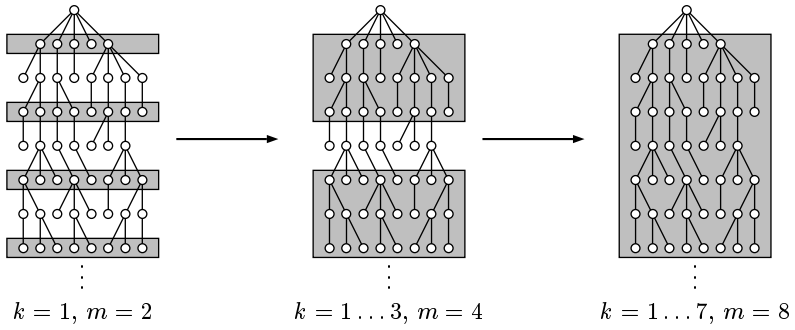

Fig. 2. Stratified caching with doubling modulo

4 Experiments with Random and Real Graphs

Our initial goal was to compare stratified caching and a random replacement strategy, but doubts arose about the optimality of random replacement. To resolve the matter, we conducted a set of experiments that explore graphs using state caching and a variety of state replacement strategies.

The replacement strategies are based on the following five state attributes: (`E`) stack entry time, (`X`) stack exit time, (`D`) search depth, (`I`) current indegree, and (`O`) current outdegree. A specific replacement strategy is a combination of attributes (denoted by uppercase letters) and negated attributes (denoted by lowercase letters). For example, the specification "`DiX`" indicates that states are first ordered by their ascending depth, then by their descending indegree, and finally by the ascending time of stack entry. When a state is selected for replacement, the least state according to this ordering is chosen.

Only attributes `E` and `X` are unique to each state. Therefore, a specification such as "`dE`" is unambiguous, as there is only one deepest, least recently entered state, while the specification "`Io`" is ambiguous, as there can be many states with minimal indegree and maximal outdegree. Note that the order of attributes is important: "`OIX`" and "`IOX`" describe two different replacement strategies.

In addition to the five attributes, two pseudo-attributes are used: (`R`) random selection, and (`S`) stratified caching. When the `R` pseudo-attribute is added to an ambiguous specification "*s*", states for replacement are chosen by randomly selecting one of the eligible states specified by *s*, and the resulting specification "*s*`R`" is unambiguous. The `S` pseudo-attribute can be combined with any unambiguous specification "*s*" to yield a strategy that selects a state within the available strata according to *s*, and employs the doubling-modulo stratified caching strategy discussed in the last section.

It is not easy to compare these strategies to those in [11]. Strategy H1 is clearly something of the form "`i...`", but in [11] a complete scan of the state store is performed to find the most frequently visited state, and it is therefore influenced by the hash function. Likewise, H2 corresponds to something of the form "`I...`". In [11] strategy H4 is said to select the oldest state in the cache, which is what "`E`" does, but H4 is often referred to as random replacement,

which corresponds to "R". H3 and H5 have no direct counterparts among the strategies investigated.

In total, 790 unambiguous specifications were investigated. The specifications are not only unambiguous, but also independent of the state storage scheme. (Note that it is not really possible to implement an ambiguous specification — the ambiguity must be resolved in some undisclosed way — and such specifications were not considered.)

The experiments in this section focus on the cache size and the amount of redundant work; memory and time consumption are not at issue here. Experience has shown that, if care is taken with the implementation, redundant work gives a good indication of the time consumption. Apart from empirical observation, there is also an intuitive argument to support this claim: The proportion of extra states explored brings about an equal proportion of extra transitions, and the number of transitions is the dominating factor in the execution time of a state exploration tool. Accurate calculation of the memory consumption of the different specifications is also possible, but, for lack of space, we do not discuss it here. It is important to note, however, that the optimality of the strategies should be judged on their memory consumption and not the amount of redundant work involved.

4.1 Experiments with Random Graphs

In the first set of experiments, random graphs were generated using the same method described in [14], and shown in Figure 3. Each random graph is determined by three parameters: the desired number of states S, the maximum outdegree D of any node, and a seed R for the random number generator. For both random graph generation and random replacement strategies, the $(2^{19937}-1)$-period Mersenne Twister random number generator [16] is used.

States are generated in a breadth-first fashion; for each new state an outdegree d is chosen uniformly in the range $0 \ldots D$; the probability that an outgoing transition leads to a new state is 0.5 for the first $\lfloor S/2 \rfloor$ states and then decreases linearly until it reaches 0 when the number of states reaches S; destination states of transitions that lead to old states are chosen uniformly from among the already generated states. This algorithm may terminate before enough states have been generated, if, for example, an outdegree $d = 0$ is chosen for the initial state. It is therefore repeated until at least $.9S$ states have been generated.

The graphs generated in this way are called *unweighted* and have a transition/state ratio of $D/2$. However, it is not only the average number but also the distribution of revisits that affect the performance of state caching. The algorithm was therefore adjusted to also generate *weighted* graphs: instead of choosing the revisited states by uniform selection (line 10 of Figure 3), each state was weighted by its number of incoming transitions. (The root state was given an additional incoming edge, since otherwise it would never be selected). Figure 4 shows the distribution of revisits for weighted and unweighted graphs; these match the distributions of many actual models.

```
GenerateGraph(S, D, R)
1   random seed ← R
2   repeat
3     Enqueue(q, 0) ; n ← 1
4     while NotEmpty(q) do
5       s ← Dequeue(q)
6       d ← Random(0 ... D)
7       for i ← 1 to d do
8         p ← 1 − Max(0.5, n/S)
9         if Random(0 ... 1) > p then {revisit an old state}
10          t ← Random(0 ... n − 1)
11        else {generate a new state}
12          t ← n ; n ← n + 1
13          Enqueue(q, t)
14        endif
15        AddTransition(s, t)
16      endfor
17    endwhile
18  until n ≥ .9S
```

Fig. 3. Code for generation of random graphs

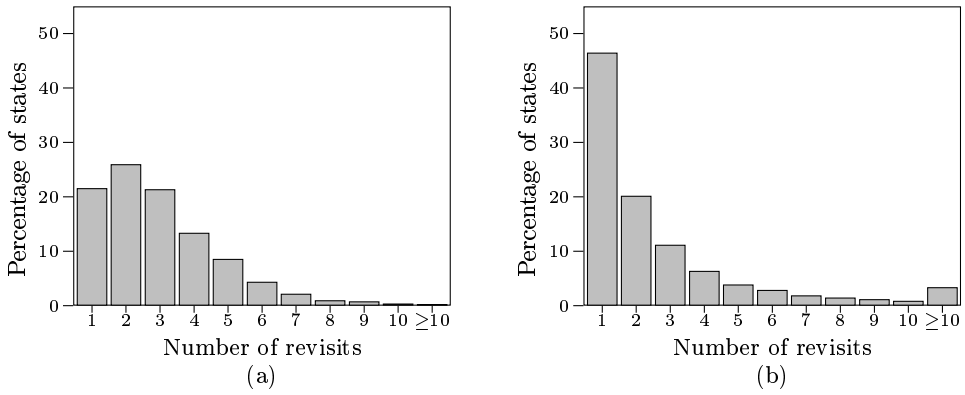

Fig. 4. The distribution of state revisits for (a) the unweighted random graph $S = 5000$, $D = 6$, $R = 22222222$, and (b) the weighted random graph $S = 5000$, $D = 6$, $R = 55555555$. In (a), for example, roughly 22% of states are visited only once, and roughly 26% of states are visited twice.

Thirty unweighted and thirty weighted random graphs were generated using the parameter values $S = 5000$, $D = 6, 10, 20$, and $R = 11111111n$ for $n = 0, 1, \ldots 9$. Different values of S had no discernible effect on the results, and 5000 was chosen as a representative value. Each random graph was explored using each of the unambiguous specifications as a cache replacement strategy. For the initial run, the cache size was the same as the state space size; during subsequent runs it was decremented in steps of $1 + \lfloor S/400 \rfloor$, roughly 0.25% of the state space size. This process terminated once the number of visited states exceeded 10 times the size of the state space, or once the cache reported that it was full.

A typical result is shown in Figure 5. The dot in the center shows, for example, that a cache of 48.3% the size of the state space produced 24975 reported visited states, while the actual number of states is only 4505. The reported/actual states ratio — which we call the *redundant work factor* — is therefore 5.54. The replacement strategies are difficult to compare: minimum cache size is important, but so is the redundant work factor and the relationship between the two variables. Moreover, the maximum tolerable redundant work factor is a subjective limit.

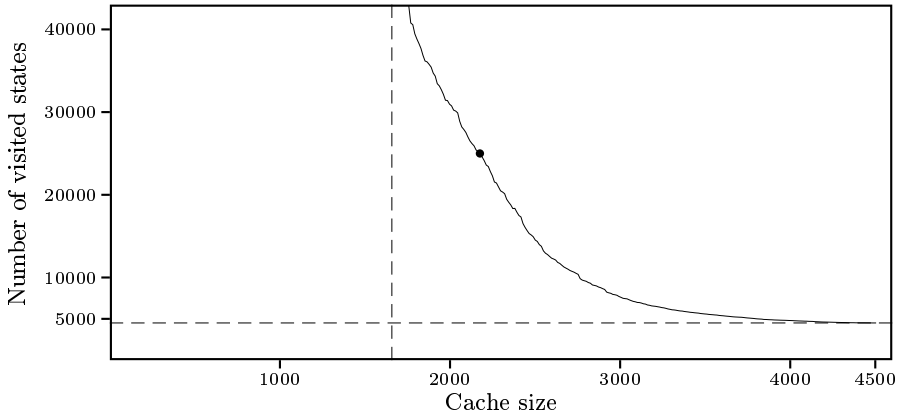

Fig. 5. The behaviour of the replacement strategy "`iDx`" (among the states with the highest indegree, select the shallowest states and, from these, choose the state that exited the depth-first stack first) on the unweighted random graph $S = 5000$, $D = 6$, $R = 0$. The graph has 4505 states (shown as the horizontal dashed line), 13249 transitions and a maximum search depth of 1658 (shown as the vertical dashed line).

Table 1 presents the results of the experiments with weighted and unweighted graphs separated. Different limits on the redundant work factor are shown in the first column, labeled f. In each row, the best strategies and their performance are shown for the ten graphs with $D = 6$, the ten graphs with $D = 10$, the ten graphs with $D = 20$, and, in the last major column, the thirty graphs combined. The best strategy was determined by averaging the minimum cache size data points for each strategy and selecting the strategy with the lowest average. In the

Table 1. State caching results for random graphs

Unweighted random graphs												
	$D = 6$			$D = 10$			$D = 20$			Combined		
f	$m = 37.43$			$m = 52.35$			$m = 70.13$			$m = 53.30$		
2	51.48	1.99	OIx	69.92	1.99	Ox	85.33	1.99	ODIe/x	68.95	1.99	OIx
3	44.22	2.96	OIR	62.43	2.99	OIDE/X	79.71	2.96	ODiR	62.15	2.97	OIDE/X
4	41.32	3.92	OIR	59.05	3.96	OIe	76.88	3.93	ODIe/x	59.12	3.94	OIDe/x
5	39.76	4.91	OIR	57.18	4.93	OIDe/x	75.16	4.90	ODIR	57.42	4.90	OIDe/x
6	38.79	5.78	OIR	56.00	5.85	OIDR	74.03	5.91	ODR	56.30	5.83	OIe
7	38.13	6.27	IOe	55.11	6.86	OIe	73.28	6.84	OIe	55.56	6.75	OIe
8	37.97	7.11	OIe	54.48	7.78	OIe	72.74	7.72	OIDR	55.06	7.54	OIe
9	37.94	7.15	OIe	54.03	8.65	OIe	72.29	8.75	OIDE/X	54.76	8.17	OIe
10	37.89	7.30	OIe	53.74	9.49	OIe	71.98	9.70	OIDR	54.55	8.79	OIe
Weighted random graphs												
	$D = 6$			$D = 10$			$D = 20$			Combined		
f	$m = 25.71$			$m = 35.51$			$m = 49.67$			$m = 36.96$		
2	37.06	1.98	IOE	51.05	1.98	IOE	66.24	1.99	IOE	51.45	1.98	IOE
3	31.78	2.94	IOR	45.08	2.96	IODR	60.47	2.96	IOE	45.81	2.95	IOR
4	29.20	3.90	IOe	42.20	3.96	IODe/x	57.89	3.90	IODe/x	43.13	3.92	IODe/x
5	27.27	4.87	IOe	39.83	4.94	IODe/x	56.10	4.89	IOR	41.15	4.89	IOe
6	26.61	5.56	IOe	38.20	5.90	IODe/x	54.41	5.92	IOx	39.78	5.80	IOe
7	26.37	5.94	IOe	37.02	6.89	IODR	53.11	6.89	IODe/x	38.89	6.56	IOe
8	26.34	5.99	IOe	36.34	7.66	IOx	52.07	7.85	IOe	38.29	7.28	IOx
9	26.34	5.99	IOe	36.13	8.07	IOx	51.39	8.68	IODe/x	37.96	7.66	IOe
10	26.34	5.99	IOe	36.10	8.16	IODE/X	50.90	9.46	IODR	37.81	8.08	IODR

case of more than one minimum, the strategy with the lowest average redundant work factor is shown. In a few cases two strategies of the form "*s*`e`" and "*s*`x`" (or "*s*`E`" and "*s*`X`") performed equally well. The minor columns give the averaged minimum cache size as a percentage of the state space size, the redundant work factor, and the strategy specifications selected as the best. Lastly, the value of m in the heading row is the maximum stack depth expressed as a percentage of the state space size; this is minimum possible size of the cache.

The first thing to note is that the table gives a very narrow view of the results, since there is no indication of how other strategies fared. For example, for unweighted graphs with $D = 20$ and $f = 4$, there are 14 other strategies that attain the 76.88% minimum cache size; they are not shown because their redundant work factor exceeded that of "`ODIe/x`". In total, 32 strategies came in below 77%, 65 strategies below 80%, and 230 strategies below 85%. The disadvantage of not showing all these results is, however, balanced by the consistency of the results, not to mention the problem of presenting such a volume of data.

The random replacement and the stratified caching strategies did not fare well enough to appear in the table. For unweighted graphs the average minimum cache size for random replacement ("`R`") is between 7.41% and 25.73% higher than that of the best strategy. Although stratified caching (specifically "`eS`") fared consistently better, its figures are still between 4.11% and 16.18% higher.

Instead, the "`OI...`" and "`IO...`" strategies dominate. The `I` attribute selects states with low indegree. Even if, as reported in [11], the number of past visits is not highly correlated with future visits, this approach works since, if many states are visited only once or twice, it is right more often than it is wrong. The `O` attribute selects states with low outdegree. If these states are revisited, there are fewer children that may have been replaced and need to be re-investigated.

The `D` attribute also appears several times among the more highly connected graphs. This suggests that back edges are more likely to lead to closer levels, making it pay off to replace the shallowest states first. This may be due to the generation process: since there are more states at the deeper levels, deeper states have a higher probability of being selected for revisits. As to the occurrence of the `e`/`x` and `E`/`X` patterns, we believe that the disambiguating attribute (one of `E`/`e`/`X`/`x`/`R`) in all of the specifications is somewhat arbitrary, although the `e` attribute seems to have a slight edge.

For the moment, all these observations are speculative, and warrant further investigation. It is, however, clear that a redundant work limit of $f = 10$ is too generous; even when $f = 5$ the best strategies came within 5% of the absolute minimum cache size.

4.2 Experiments with Real Graphs

In addition to random graphs, several Promela models were converted (after partial order reduction by the SPIN tool) to graphs and explored as above, but with a redundant work factor limit of $f = 5$; the results are shown in Table 2. The columns show the model name, the number of states (in column *states*), the transition/state ratio (column d), the maximum stack depth as a percentage

of S (column m), the minimum cache size as a percentage of S (column s), the redundant work factor (column *rwf*), and the number of best strategies and their specifications.

The results fall into roughly three groups: for the first group (dining, erasthostenes, mobile2, schedule, and gobackn2) "OX" (i.e., least outdegree and then least recently removed from stack) strategies work best, for the second group (dbm, X509, mobile1, petersonN, rap, pftp, slide, and gobackn) "RS" (stratified caching with random selection of available states) strategies work best, and for the last group (the other graphs) the best strategies were more varied. At present we are unable to explain this grouping, but hope to investigate it further.

Table 2. State caching results for graphs based on actual models

Model	*states*	d	m	s	*rwf*	Best strategies
mutex	428	2.0	15.42	24.53	3.16	3 IeS ieS eS
abp2	1440	1.3	23.26	23.40	1.02	5 iOe Oie OIe IOe Oe
cabp	2048	4.1	38.13	47.41	4.37	14 OdIeS OdIxS OdieS OdeS ...
dining	2071	3.6	71.32	74.02	3.99	5 OiXS IOXS OIXS OXS iOXS
erathostenes	2092	1.2	10.71	12.24	1.51	5 iOX OiX OIX IOX OX
mobile2	3300	2.0	23.36	23.79	4.94	5 iOX OiX OIX IOX OX
schedule	3328	1.3	16.29	16.77	1.37	5 iOX OiX OIX IOX OX
dbm	5111	4.0	1.96	15.07	4.58	1 iRS
X509	6093	2.0	0.89	13.52	4.68	1 RS
snoopy	9342	1.4	16.73	17.00	1.79	1 OiR
mobile1	9970	2.0	7.16	13.75	3.86	1 RS
gobackn2	14644	1.8	30.47	31.28	4.71	5 iOX OiX OIX IOX OX
petersonN	16719	1.8	29.40	31.17	3.82	1 RS
rap	26887	7.9	0.23	18.16	1.81	1 RS
pftp	47355	1.4	3.18	6.91	3.39	1 RS
riaan	67044	1.1	0.26	1.27	3.32	1 OR
slide	89910	1.5	11.07	12.91	4.48	1 ORS
gobackn	90209	1.5	11.03	12.32	4.65	1 OIdRS

When the performance (defined as $s - m$) of the strategies are averaged over the graphs and sorted, the stratified caching strategies occupy the top half of the list. In other words, the worst strategy with stratified caching outperforms the best strategy without, or to put it another way, of the replacement strategies previously considered in the literature, none achieved a better place than halfway down this list. At the top of the list is the "RS" (stratified caching with a doubling modulo in combination with random selection) strategy: its average minimum cache size is 5.41% higher than m, and its average redundant work factor is 3.12.

5 A SPIN Implementation

To further investigate the viability of some state caching schemes, the SPIN model checker was modified to include state caching. Three alternatives to out-of-the-box SPIN were investigated. The first alternative uses a cache replacement scheme identical to that of [7]. The second new version implements the `OX` strategy, identified as one of the "winners" in the previous section. It would have been instructive to investigate the `RS` strategy also, but true random selection is nontrivial to implement, especially in SPIN where state cache entries do not have a uniform length and cannot be determined a priori. It would be possible to ask the user to specify the size of the state vector, and this may not be unreasonable. However, to make the comparison as fair as possible, it was decided to implement the `XS` strategy as the third alternative SPIN version.

Table 3. SPIN state caching results for a model of leader election in a general graph

Memory	SPIN 4.1.0		+ [7]		+ `OX`		+ `XS`	
limit	*rwf*	*time*	*rwf*	*time*	*rwf*	*time*	*rwf*	*time*
No limit	1.00	2.16	–	–	–	–	–	–
100	–	–	1.00	2.29	1.00	2.26	1.00	2.26
80	–	–	1.00	5.55	1.06	2.49	1.00	2.36
60	–	–	1.02	17.82	1.13	2.71	1.00	2.78
50	–	–	1.04	29.84	1.16	3.10	1.00	2.50
40	–	–	1.07	47.21	1.19	2.91	1.01	2.55
30	–	–	1.16	76.09	1.22	3.01	1.01	2.64
20	–	–	2.28	231.97	1.25	3.09	1.04	2.70
10	–	–	–	–	26.19	63.89	1.14	3.03

Table 4. Performance of bitstate hashing with respect to partial/full exploration

`-w` *param.*	*states*	*trans.*	*time*
22	276112	350318	2.51
23	279982	354744	2.57
24	280880	355762	2.59
25	281155	356069	2.61
26	281207	356127	2.63
27	281261	356190	2.66
28	281263	356192	2.72

Table 3 shows the results of running the three new versions on SPIN on a single model of the echo algorithm with extinction for electing leaders in an

arbitrary network, as described in [20, Chapter 7]. The memory limit is given in 2^{20} bytes, the redundant work factor is defined as before, and the time is given in seconds. The experiments were performed on a 2.6GHz Pentium 4 machine with 512 megabytes of memory running Linux. These are only preliminary results and further experiments on more models are required to form a better idea of how well stratified caching behaves. From the figures in the table it is however clear that for this particular model this form of stratified caching outperforms the other two implementations.

While the model may be quite small (it has only 281263 states and 356192 transitions), it is interesting to look at the behaviour of bitstate hashing. Table 4 shows the value of the `-w` parameter which specifies the number of bits to use for bitstate hashing, the number of states and transitions reported by SPIN, and the time in seconds. Only when 2^{28} bits = 32 megabytes are used, does the system investigate the full state space. This does not reflect on bitstate hashing in general: even when it does not explore the full state space, it can find true errors in models. However, the important point is that state caching techniques can extend the limit beyond which we are forced to resort to partial exploration of state spaces.

6 Conclusions

We have shown, for both random graphs and graphs based on actual models, that for a substantial but not unlimited increase in running time, the cache can often be reduced to close to the minimum size imposed by the maximum stack depth. The maximum increase in running time is close to tenfold for the experiments in Table 1, and almost fivefold in Table 2; for the `XS` strategy in Table 3 the running time is only at most 1.4 times more than normal.

The results of the experiments are not as easy to interpret as we might have hoped. The strategies that worked best with random graphs are very different from those that worked best with actual models. For random graphs the results were more consistent than for actual models, perhaps indicating that the generated graphs represent only one "kind" of model. The experiments did however show that, by itself, random replacement is never the best strategy. Stratified caching in combination with random replacement emerged as the best strategy for almost half of the graphs based on actual models. In other cases, strategies based on the minimal outdegree and indegree of states proved successful.

We believe that stratified caching can be improved even further by selecting available strata based on the average outdegree, average indegree, or simply the number of states in a stratum. Its performance in the experiments makes it a strong candidate for the replacement strategy, whenever state caching is used.

Acknowledgments. This work was funded by the TISE graduate school and by the Academy of Finland.

References

1. G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, M. Protasi. *Complexity and Approximation.* Springer-Verlag, 1999.
2. G. Berhmann, K. G. Larsen, R. Pelánek. To store or not to store. *Proc. 15th Intl. Conf. on Computer-Aided Verification*, LNCS#2725, pp. 433–445, 2003.
3. L. Brim, I. Černá, M. Nečesal. Randomization helps in LTL model checking. *Proc. Joint Intl. Workshop on Process Algebra and Probabilistic Methods, Performance Modeling and Verification*, LNCS#2165, pp. 105–119, 2001.
4. S. Edelkamp, A. L. Lafuente, S. Leue. Directed explicit model checking with HSF-Spin. *Proc. 8th Intl.* Spin *Workshop on Model Checking of Software*, LNCS#2057, pp. 57–79, 2001.
5. J. Geldenhuys. *Efficiency Issues in the Design of a Model Checker.* Master's thesis, University of Stellenbosch, 1999.
6. P. Godefroid. Using partial orders to improve automatic verification methods. *Proc. 2nd Intl. Conf. on Computer-Aided Verification*, LNCS#531, pp. 176–185, 1990.
7. P. Godefroid, G. J. Holzmann, D. Pirottin. State space caching revisited. *Proc. 4th Intl. Conf. on Computer-Aided Verification*, LNCS#663, pp. 175–186, 1992. Also appeared in *Formal Methods in System Design*, 7(3):227–241, 1995. `http://www.montefiore.ulg.ac.be/services/verif/po-pack.html`
8. P. Godefroid. *Partial-order Methods for the Verification of Concurrent Systems, An Approach to the State-explosion Problem.* PhD thesis, University of Liège, Dec 1994. Also published as LNCS #1032, Springer-Verlag, 1996.
9. P. Godefroid. On the costs and benefits of using partial-order methods for the verification of concurrent systems. *Proc. Workshop on Partial Order Methods in Verification*, DIMACS Series vol. 29, pp. 289–303, 1996.
10. G. J. Holzmann. Tracing protocols. *AT&T Technical J.*, 64(10):2413–2433, 1985.
11. G. J. Holzmann. Automated protocol validation in *Argos*, assertion proving and scatter searching. *IEEE Trans. on Software Engineering*, 13(6):683–696, 1987.
12. G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.
13. C. Jard, T. Jéron. On-line model checking for finite linear temporal logic specifications. *Proc. Intl. Workshop on Automatic Verification Methods for Finite State Systems*, LNCS#407, pp. 275–285, 1989.
14. C. Jard, T. Jéron. Bounded memory algorithm for verification on-the-fly. *Proc. 3rd Intl. Conf. on Computer-Aided Verification*, LNCS#575, pp. 192–202, 1991.
15. C. Jard, T. Jéron, J.-C. Fernandez, L. Mounier. On-the-fly Verification of Finite Transition Systems. Tech. rep. #701, IRISA, 1993.
16. M. Matsumoto, T. Nishimura. Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Trans. on Modeling and Computer Simulation*, 8(1):3–30, 1998.
    `http://www.math.keio.ac.jp/~matumoto/emt.html`
17. A. N. Parashkevov, J. T. Yantchev. Space efficient reachability analysis through use of pseudo-root states. *Proc. 3rd Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS#1217, pp. 50–64, 1997.
18. D. Peled. All from one, one for all: On model checking using representatives. In *Proc. 5th Intl. Conf. Computer-Aided Verification*, LNCS #697, pp. 409–423, Jun 1993.
19. U. Stern, D. L. Dill. Combining state space caching and hash compaction. *Methoden des Entwurfs und der Verifikation digitaler Systeme, 4. GI/ITG/GME Workshop*, pp. 81–90, 1996.

20. G. Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 2nd edition, 2000.
21. A. Valmari. A stubborn attack on state explosion. *Formal Methods in System Design* 1(1), pp. 297–322, 1992.