FI MU

Faculty of Informatics
Masaryk University Brno

Evaluation of State Caching and State Compression Techniques

by

Radek Pelánek
Václav Rosecký
Jaroslav Šeděnka

FI MU Report Series FIMU-RS-2008-02

Publications in the FI MU Report Series are in general accessible via WWW:

`http://www.fi.muni.cz/reports/`

Further information can be obtained by contacting:

Faculty of Informatics
Masaryk University
Botanická 68a
602 00 Brno
Czech Republic

Evaluation of State Caching and State Compression Techniques[*]

Radek Pelánek Václav Rosecký
Jaroslav Šeděnka
Faculty of Informatics
Masaryk University Brno, Czech Republic

February 28, 2008

Abstract

We study two techniques for reducing memory consumption of explicit model checking — state caching and state compression. In order to evaluate these techniques we review the literature, discuss trends in relevant research, and perform experiments over a large benchmark set (more than 100 models). The conclusion of our evaluation is that it is more important to combine several simple techniques in an appropriate way rather than to tune a single sophisticated technique.

1 Introduction

In this work we are concerned with explicit model checking and verification of safety properties. This approach is principally very simple — it is based on the straightforward construction of the whole state space and on a simple reachability analysis of this state space. The technique is successful for verification of asynchronous systems, particularly protocols.

The main problem of explicit model checking is the state space explosion problem and hence large memory requirements of the technique. Researchers proposed during the last 15 years many techniques aimed at reduction of the memory consumption of explicit model checking. At this moment, there is a large number of reduction technique proposals. However, from a practitioner point of view, the situation is not satisfying:

[*]Partially supported by GA ČR grant no. 201/07/P035.

- Research papers usually include only few experiments on selected models on which the techniques bring (non-trivial) improvement. Our evaluation [Pel05] of on-the-fly reduction techniques shows that the improvement is often more humble than claimed in research papers.
- Most reduction techniques involve some kind of trade-off (usually time-memory) and they bring improvement on only some models. The trade-off and the dependence of performance on model properties is not well understood.
- Research papers usually compare a newly proposed technique only to the standard algorithm and not to other reduction technique. It is also not clear whether the impact of different reduction techniques combine.

1.1 Contribution

In this paper we focus on two techniques for reducing the memory consumption of the explicit model checking: state caching and state compression. State caching saves memory by deleting some visited states from memory; state compression saves memory by compressing individual states in memory. We evaluate these techniques in the following way:

- We review the literature and discuss trends in the relevant research.
- We perform experiments with caching and compression techniques over a large benchmark set (BEEM [Pel07a]) and we present results of these experiments and their interpretation.
- We analyze how the performance of the techniques depends on model and state space parameters.

1.2 Context

This work is a part of our long term effort to make the verification process more automatic, i.e., to automate the selection of reduction techniques and parameters [Pel07b]. Our other works which contribute to this goal are the following:

- a general overview and evaluation of on-the-fly reduction techniques [Pel05],
- evaluation of techniques for error detection [PRM08],

- analysis of properties of state spaces [Pel04],
- description and evaluation of techniques for estimating state space size and other parameters [PŠ08].

2 Review of Literature

Before the discussion of related work we clarify the terminology that we use to discuss the effect of techniques for reduction of memory consumption. We use the notion 'reduction ratio' to denote the ratio between the memory consumption of the reachability search with a memory reduction technique and the memory consumption of the standard reachability. Some authors report 'reduced by' factor, i.e., if we report 'reduction ratio' 80%, it means that the memory consumption was 'reduced by' 20%.

State caching and compression techniques have been studied rather extensively. The main works are the following:

- Holzmann [Hol85, Hol87] was the first to propose the state caching technique, but he did not perform realistic evaluation of the technique.
- Godefroid et al. [GHP92] focus on the relation of state caching and partial order reduction. Their experiments are, however, limited to only few models.
- Geldenhuys [Gel04] performs more extensive evaluation and compares many different caching strategies. However, this evaluation is done partly on random graphs, which can be misleading (for argumentation against the use of random graph see [Pel04]).
- Similar approaches to state caching are selective storing of states during the search [BLP03] and the sweep line technique [CKM01], which deletes only states that are guaranteed to not be revisited.
- Holzmann [Hol97] describes the state compression algorithm with training runs in Spin model checker. Each part of the state space is compressed according to a local table.
- Visser [Vis96] describes a similar compression techniques as Holzmann [Hol97] and combines the compression technique with OBDD storage.

```
proc EXPLORE(M)
   add $s_0$ to Wait
   while Wait $\neq \emptyset$ do
         remove s from Wait
         explore (s)
         foreach $s \rightarrow s'$ do
            if $s' \notin$ Visited then add $s'$ to Wait fi od
   od
end
```

Figure 1: The basic algorithm for exploring the state space.

Table 1 gives the overview of these and several other most relevant papers. From this table we can see the following general trends:

- There is a steady flow of publications about the topic (1-2 every year).
- At first, techniques were implemented in SPIN (and its predecessors), from 2000 onwards the scope of used tools is rather diverse.
- We expected that the (reported) reduction ratio would increase in time, however there is no clear trend with time. The reported reduction ratio is usually in the interval 5% to 80%.
- The quality of experiments (number and quality of used model and performed experiments) is nearly constant, despite the improving availability of realistic models.

3 Techniques

In this section we describe formally the context of our work and the state caching and state compression techniques.

3.1 State Space Exploration

Figure 1 gives the basic state space exploration algorithms. It is just a simple graph traversal algorithm, which uses two important data structures:

paper	year	technique	tool	number of models	reported reduction ratio
[Hol85]	1985	caching	Trace	1	
[Hol87]	1987	caching	Argos	2	
[JJ92]	1991	caching	unknown	0	
[HGP92]	1992	compression	SPIN	5	~ 80%
[GHP92]	1992	caching	SPIN	4	1-3% (*)
[Vis96]	1996	compression	SPIN	5	5-25%
[Gre96]	1996	compression	SPIN	3	~ 80%
[PY97]	1997	caching-like	ARC	11	15-50%
[Hol97]	1997	compression	SPIN	17	15-50%
[LLPY97]	1997	compression, caching-like	Uppaal	10	5-25%
[Par98]	1998	compression	SPIN	?	20-60%
[HP98]	1999	compression	SPIN	14	~ 15%
[GdV99]	1999	compression	SPIN	7	40-60%
[CKM01]	2001	caching-like	Design/CPN	3	5-80%
[PIM$^+$04]	2001	caching-like	Murphi	20	~ 60%
[LV01]	2001	compression	JPF	2	~ 5%
[BLP03]	2003	caching-like	Uppaal	9	5-80%
[GV03]	2003	compression	TVT	4	15-50%
[Gel04]	2004	caching	SPIN	18	5-50%
[EPP05]	2005	compression	Helena	8	30-60%

Table 1: Review of literature. Experiments on random graphs are not taken into account. The 'reported reduction ratio' is only an approximate due to the use of different metrics to measure memory consumption; the reduction ratio is given only if enough experiments are reported in the paper. (*) The small reduction ratio is due to combination with partial order reduction method.

- *Visited* is a set of states that were visited during the exploration. Since we need to perform a test of membership in this data structure, the set is usually represented by hash table.
- *Wait* is a set of states that need to be explored. The implementation of this data structure determines the search order of the algorithm — usually either breadth-first search (BFS) or depth-first search (DFS).

States are represented as vectors of bytes, which code the current location of individual processes and values of variables.

3.2 State Caching

The basic idea of state caching is simple: if we run out of memory then we remove some states from the data structure *Visited*. This, of course, has the consequence of revisits of states and thus time increase. If we use depth-first search, the method is still guaranteed to terminate. With breadth-first search order we do not have such a guarantee in general.

Note that the name of the approach is slightly misleading, since it is not caching in the usual sense of the word — states are *not* moved lower in the memory hierarchy, they are simply deleted. However, in model checking the technique is traditionally called this way [GHP92, Gel04].

3.2.1 Caching Strategies

The main issue in application of state caching is to determine states for removal from the cache. We call an algorithm for selection of states a *caching strategy*. In our experiments we consider the following strategies (see also [Gel04]):

O (out-degree) States are removed according to the number of successors (out-degree) of the state. The intuition behind the strategy is that states with higher number of successors have higher probability that some of its successor was forgotten.

I (in-degree) States are removed according to the number of visits (actual in-degree) of the state. The intuition behind the strategy is that often visited states will also be more visited in the future.

OI (out-degree, in-degree) The strategy takes into account both the out-degree and the actual in-degree.

RAND (random) States are removed randomly.

SC (stratified caching) We assign to every stored state the depth on which it was discovered (the length of the path from the initial state). States lying on the same depth form stratas. We erase state s when the following predicate is true (d is depth of a state s, k is a constant, initially $k = 2$):

$$d \bmod k \neq k - 1$$

When no such state exists, the constant k is doubled, so there are additional available stratas with states for erasing.

3.3 State Compression

State vectors contain significant redundancy. Researchers proposed several techniques that try to reduce the size of state vectors (see Section 2). Previous studies suggest that these techniques achieve similar results, therefor we focus on the most typical compression techniques — huffman coding.

Huffman coding [Huf52] is a general loss-less compression method that is proven to be memory-optimal when exact probabilities of each value usage are known. The Huffman code can be constructed by the well-known algorithm [Huf52].

3.3.1 Static Code

In order to construct the Huffman code we need to know the frequencies of individual values. Since prior to the reachability analysis we do not have the knowledge of the frequencies, we cannot construct the optimal Huffman code. Nevertheless, we can construct at least some Huffman code using approximated frequencies.

A straightforward approach to compute this estimated frequencies is to take a representative set of small models, compute frequencies of values in these models and then compute a “static” Hufmann code. Part of this static Huffman code (that we use in the experiments) is shown in Table 2.

0	00	...	...
1	01	248	111111111010
2	1000	249	111111111011
3	1001	250	1111111111000
4	1010	251	1111111111001
5	1011	252	111111111101
6	11001	253	111111111110
7	11000	254	111111111111
8	1101	255	11100

Table 2: Hufmann code used for the compression — part of the static code.

3.3.2 Training Runs

A more precise way of gathering value frequencies is to go through a small part of the state space, i.e., to perform a short reachability. We call this preliminary reachability a *training run*, as it is not supposed to walk through the whole state space. After such training run we create Huffman trees based on inaccurate probabilities.

There are several possibilities how to perform the training run. We consider two basic approaches to sampling a state space during the training run: DFS and BFS, each with a specified limit on a number of states to be searched.

4 Evaluation

Reported techniques are implemented in the DiVinE environment [BBČ06]. Experiments were performed on 2GHz Intel Xeon Linux workstation with 16 GB RAM.

Evaluation was performed over the BEEM benchmark set [Pel07a]. The web portal of the benchmark set contains all the used models and it also presents detailed information about models. For our experiments we used 120 models, which fall into following (BEEM) categories: 29 communication protocols, 18 controllers, 6 leader election algorithms, 26 mutual exclusion algorithms, 22 scheduling problems, 19 other; 26 complex case studies, 70 simple models, 24 toy examples.

In this section we report only summaries of results and their interpretation; all results can be found on the following web page:

`http://anna.fi.muni.cz/~xrosecky/mem_reduction/`

We measured (computed) both the real memory consumption and the (theoretical) consumption of the storage data structures. The real memory consumption is higher due to the additional implementation overheads. The distinction between these two metrics is important because the used techniques can reduce only the memory consumed by storage data structures. Each of these metrics have its (dis)advantages — theoretical consumption of storage structures is less implementation dependent, however real memory consumption is, at the end, what really counts.

Nevertheless, our experiments show that these two metrics are rather well correlated and that the main conclusions of our experiments are not dependent on the used metric. In the following we use the real memory consumption. More specifically, we report relative memory consumption with respect to standard reachability.

4.1 Caching

For state caching there is a clear trade-off between memory consumption and runtime. Figure 2 gives several examples of this trade-off. Since our focus is on memory consumption, we use the following approach. We use a relative time limit — 6 times the runtime of standard reachability. We run reachability with caching with different cache sizes, cache sizes are also set relatively to the full reachability ($i \cdot 10\%$ of state space size). We consider the run successful if it finishes within the given time limit.

4.1.1 Comparison of strategies

We run the experiment for all caching strategies. Since the performance of caching is dependent on the search order (BFS or DFS), we also combined each strategy with both BFS and DFS order. We use the following notation: BFS-SC means reachability search with BFS order and SC caching strategy. Here we report only on 7 of these combined strategies, the results are representative of the overall results:

- The overall results of different strategies are similar, nevertheless there is a difference between different strategies (Fig. 3).
- There is no universally best strategy, strategies are to a certain degree complementary; Figure 4 shows which techniques have similar behaviour.

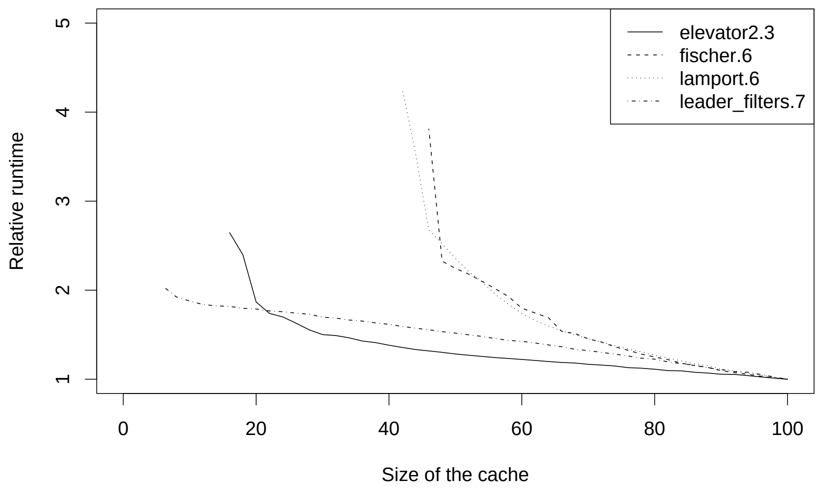

Figure 2: Trade-off between size of the cache (reduction of memory consumption) and runtime increase.

- The most successful strategies are the following (numbers indicate how many times a technique achieves the best result): BFS-SC (56), DFS-SC (37), BFS-OI (14), DFS-OI (7).
- In 71% of cases the best strategy is successful with cache size 20% of less of the full state space.

4.1.2 Dependence on a model

The performance of caching depends on the model — for some models we can use small cache, for other models we need to store nearly all states. Which characteristics of the model influence the performance of caching?

- Type of a model: for controllers, leader election algorithms, and communication protocols caching works well (10% cache is sufficient for more than a half of these models), whereas for mutual exclusion algorithms caching works poorly (at least 40% cache is necessary for more than a half of these models).
- Average degree of the state space (correlation coefficient is 0.53): caching works better for sparse state spaces (see Figure 5).

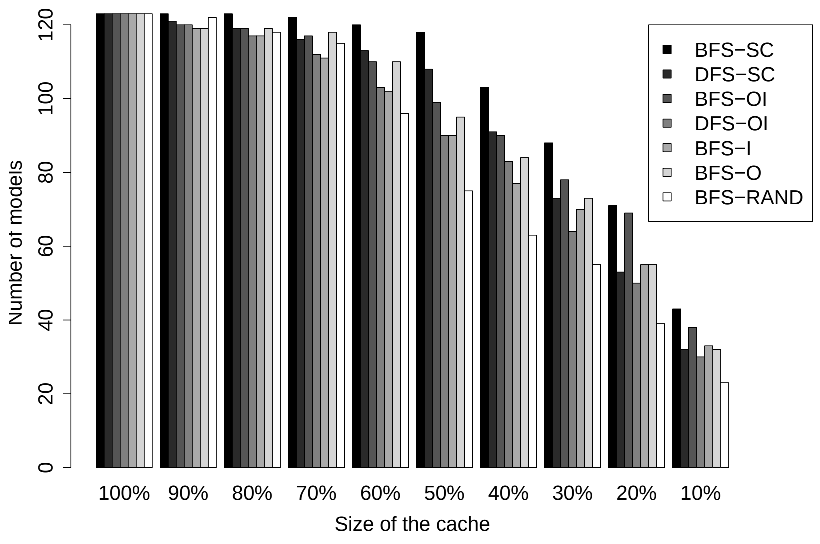

Figure 3: Comparison of caching strategies. The graph shows the number of successes for different strategies and cache sizes.

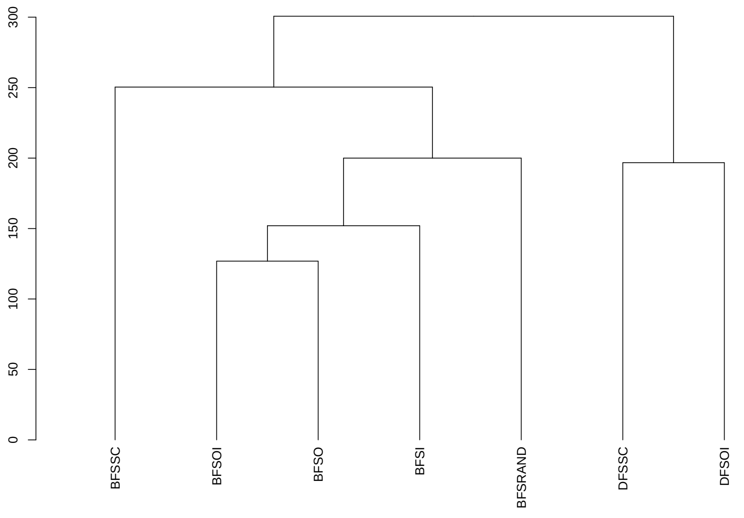

Figure 4: Clustering tree which shows similarity between caching techniques.

- BFS height of the state space (correlation coefficient is -0.43 with respect to logarithm of the height): caching works better for state spaces with many BFS levels (see Figure 5).

4.2 Compression

The performance of state compression technique is not dependent on the search order (we use BFS). For each model we run the compression technique with static code and with codes obtained by analysis of 4 different training runs: BFS (respectively DFS) with limit of 2% (respectively 15 %) of the state space. Results of these experiments can be summarized as follows:

- The reduction ratio is in range 40% to 90%, typically 60% (see Figure 6).
- The reduction ratio is slightly better with the huffman code obtained from training runs than with the static code (approximately 10% better) (see Figure 6).
- Neither the type of the training run (BFS, DFS) nor the size of the training run (2% or 15% of the state space) are important — the reduction ratio is very similar (see Figure 6).
- The performance of state compression technique depends on state size — there is very good linear correlation with logarithm of state size (see Figure 7, correlation coefficient is -0.84).
- There is nearly no relation between the runtime increase and reduction of memory consumption (cf. caching techniques).
- The performance of state compression technique is not related to the type of model (cf. caching techniques).

4.3 Combination

Finally, we implemented and evaluated the combination of caching and compression techniques. For the evaluation we used fixed strategies: BFS search order with stratified caching and compression with dictionary computed by a BFS training run (15% of the state space). The results are following (see also Figure 8):

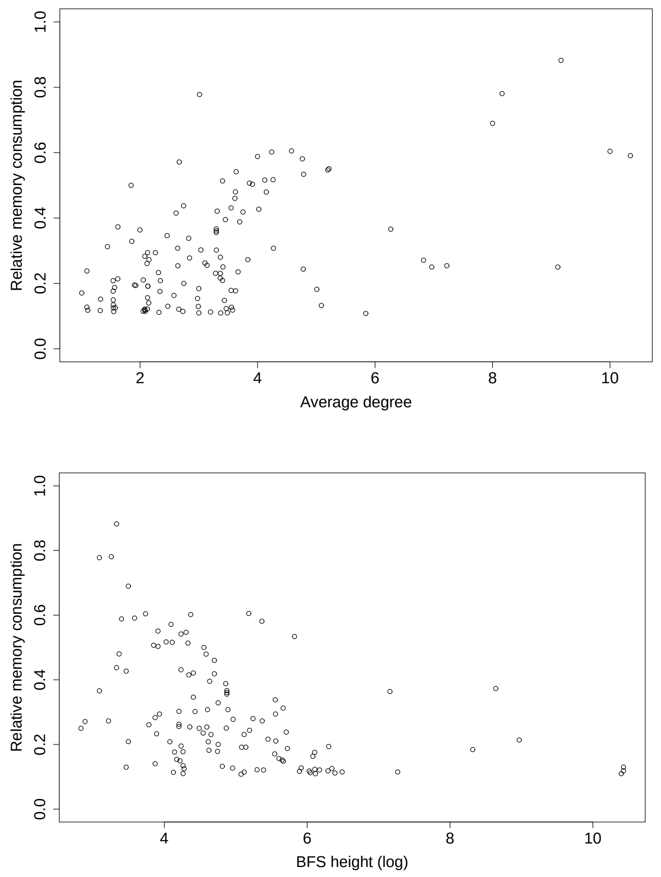

Figure 5: Best results of caching techniques and their relation to state space parameters.

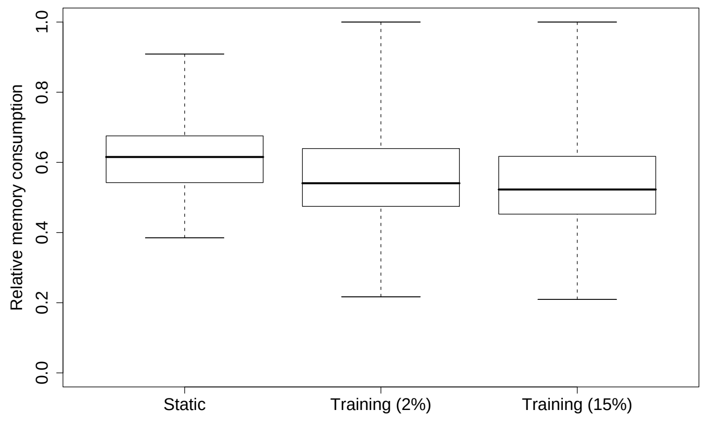

Figure 6: Compression: static dictionary, training runs

- Since the techniques are rather orthogonal, they combine well.
- The achieved reduction ratio is between 5% and 70% with median value 25%.
- Better results are achieved for complex case studies, particularly for leader election and communication protocols, worse results are for mutual exclusion algorithms.

5 Conclusions

In this paper we focus on two techniques for reducing memory consumption of model checking algorithms: state caching and state compression.

5.1 Evaluation of Effectiveness

We evaluate these techniques over a large set of models and reach the following conclusions about their effectiveness:

- Caching strategies are to a certain degree complementary. Using an appropriate state caching strategy, the memory consumption can be in most cases reduced to 10% to 30%. Using a fixed strategy (stratified caching with BFS order), cache of the size 30% of the state space is sufficient in 3/4 of cases.
- Using state compression the memory consumption can be usually reduced to approximately 60%. Using training runs (a short preliminary reachability analysis) the performance of compression can be improved, but only slightly.
- The two techniques combine well.
- Effectiveness of state caching is related to average degree, height of the BFS tree and the type of a model. Effectiveness of state caching differs for toy models and real case studies.
- Effectiveness of state compression is very well correlated with state size, it does not depend on other parameters.

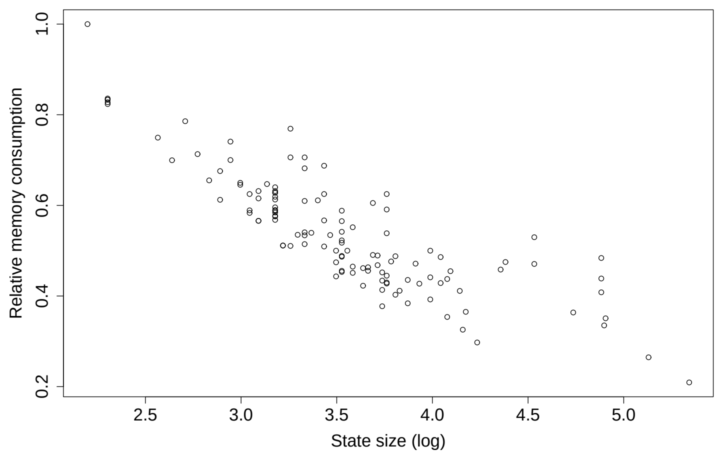

Figure 7: Correlation between state size and memory consumption

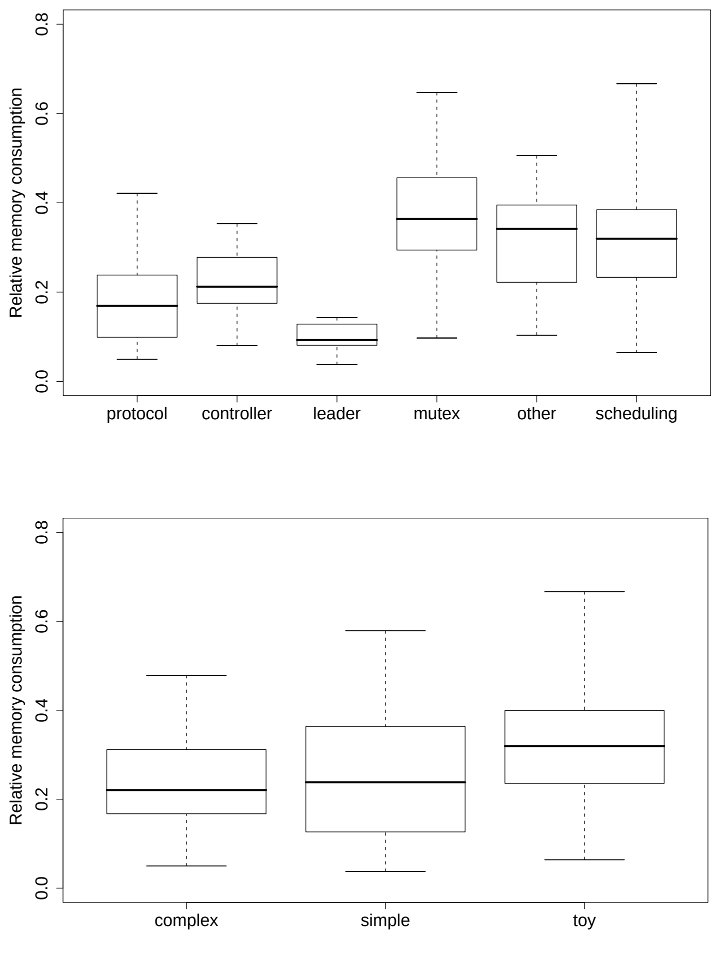

Figure 8: Results for combination of caching and compression

5.2 Comparison with Related Work

Let us also put our work in the context of the numerous related work. Rather then tuning a single implementation (as is the case of most of the related work), we try several simple strategies and parameter values. We also use significantly larger number of models than other studies and compare the performance on different types of models. In this context, the results of our study are the following:

- Our review of the literature as well as our experimental results show that the reduction ratio obtained by state caching and state compression techniques is in most cases in the interval 10-80%, i.e., with the use of these techniques it may be possible to traverse up to 10 times larger state space than by standard search.
- Using simple and easy-to-implement techniques, we are able to achieve very similar results as reported in other works which use more sophisticated approaches.
- The performance depends on used models — the choice of models does not change the overall results fundamentally (smaller number of models may be sufficient to get the basic insight), but with respect to comparison of techniques the choice of models can be important. Caching and compression techniques work better on realistic models than on academic toy examples.

5.3 Outlook

In the case of state caching and state compression techniques it seems better to implement several simple technique than a single sophisticated one. Rather than tuning a single technique, we should focus on methods for selecting an appropriate simple techniques and choosing suitable parameter values.

References

[BBČ06] J. Barnat, L. Brim, I. Černá, P. Moravec, P. Rockai, and P. Šimeček. Divine - a tool for distributed verification. In *Proc. of Computer Aided Verification (CAV'06)*, volume 4144 of *LNCS*, pages 278–281. Springer, 2006. The tool is available at `http://anna.fi.muni.cz/divine`.

[BLP03] G. Behrmann, K. G. Larsen, and R. Pelánek. To store or not to store. In *Proc. of Computer Aided Verification (CAV 2003)*, volume 2725 of *LNCS*. Springer, 2003.

[CKM01] S. Christensen, L.M. Kristensen, and T. Mailund. A Sweep-Line Method for State Space Exploration. In *Proc. of Tools and Algorithms for Construction and Analysis of Systems (TACAS 2001)*, volume 2031 of *LNCS*, pages 450–464. Springer, 2001.

[EPP05] S. Evangelista and J.-F. Pradat-Peyre. Memory efficient state space storage in explicit software model checking. In *Proc. of Model Checking Software (SPIN)*, volume 3639 of *LNCS*, pages 43–57. Springer, 2005.

[GdV99] J. Geldenhuys and P. J. A. de Villiers. Runtime efficient state compaction in SPIN. In *Proc. of SPIN Workshop*, volume 1680 of *LNCS*, pages 12–21. Springer, 1999.

[Gel04] J. Geldenhuys. State caching reconsidered. In *Proc. of SPIN Workshop*, volume 2989 of *LNCS*, pages 23–39. Springer, 2004.

[GHP92] P. Godefroid, G. J. Holzmann, and D. Pirottin. State space caching revisited. In *Proc. of Computer Aided Verification (CAV 1992)*, volume 663 of *LNCS*, pages 178–191. Springer, 1992.

[Gre96] J. Gregoire. State space compression in spin with GETSs. In *Proc. Second SPIN Workshop*. Rutgers University, New Brunswick, New Jersey, 1996.

[GV03] J. Geldenhuys and A. Valmari. A nearly memory-optimal data structure for sets and mappings. In *Proc. of Model Checking Software (SPIN)*, volume 2648 of *LNCS*, pages 136–150. Springer, 2003.

[HGP92] G. J. Holzmann, P. Godefroid, and D. Pirottin. Coverage preserving reduction strategies for reachability analysis. In *Proc. of Protocol Specification, Testing, and Verification*, 1992.

[Hol85] G. J. Holzmann. Tracing protocols. *Bell System Technical Journal*, 64(2413-2434):336, 1985.

[Hol87] G. J. Holzmann. Automated protocol validation in argos: Assertion proving and scatter searching. *IEEE Trans. Softw. Eng.*, 13(6):683–696, 1987.

[Hol97] G. J. Holzmann. State compression in SPIN: Recursive indexing and compression training runs. In *Proc. of SPIN Workshop*, 1997.

[HP98] G. J. Holzmann and A. Puri. A minimized automaton representation of reachable states. *Software Tools for Technology Transfer (STTT)*, 3(1):270–278, 1998.

[Huf52] D. A. Huffman. A method for the construction of minimum redundancy codes. *Proc. of the Institute of Radio Engineers*, 40(9):1098–1101, Sep 1952.

[JJ92] C. Jard and T. Jéron. Bounded-memory algorithms for verification on-the-fly. In *Proc. Computer Aided Verification (CAV'(91)*, volume 575 of *LNCS*, pages 192–202. Springer, 1992.

[LLPY97] K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Efficient verification of real-time systems: compact data structure and state-space reduction. In *Proc. of IEEE Real-Time Systems Symposium*, pages 14–24. IEEE Computer Society, 1997.

[LV01] F. Lerda and W. Visser. Addressing dynamic issues of program model checking. In *Proc. of SPIN Workshop*, volume 2057 of *LNCS*, pages 80–102. Springer, 2001.

[Par98] B. Parreaux. Difference compression in spin. In *Proc. of Workshop on automata theoric verification with the SPIN model checker (SPIN'98)*, 1998.

[Pel04] R. Pelánek. Typical structural properties of state spaces. In *Proc. of SPIN Workshop*, volume 2989 of *LNCS*, pages 5–22. Springer, 2004.

[Pel05] R. Pelánek. Evaluation of on-the-fly state space reductions. In *Proc. of Mathematical and Engineering Methods in Computer Science (MEMICS'05)*, pages 121–127, 2005.

[Pel07a] R. Pelánek. Beem: Benchmarks for explicit model checkers. In *Proc. of SPIN Workshop*, LNCS. Springer, 2007. To appear.

[Pel07b] R. Pelánek. Model classifications and automated verification. In *Proc. of Formal Methods for Industrial Critical Systems (FMICS'07)*, 2007. To appear.

[PIM$^+$04] G. D. Penna, B. Intrigila, I. Melatti, E. Tronci, and M. V. Zilli. Exploiting transition locality in automatic verification of finite state concurrent systems. *Software Tools for Technology Transfer (STTT)*, 6(4):320–341, 2004.

[PRM08] R. Pelánek, V. Rosecký, and P. Moravec. Complementarity of error detection techniques. In *Proc. of Parallel and Distributed Methods in verifiCation (PDMC)*, 2008. To appear.

[PŠ08] R. Pelánek and P. Šimeček. Estimating state space parameters. Technical Report FIMU-RS-2008-01, Masaryk University Brno, 2008.

[PY97] A. N. Parashkevov and J. Yantchev. Space efficient reachability analysis through use of pseudo-root states. In *Proc. of Tools and Algorithms for Construction and Analysis of Systems (TACAS '97)*, volume 1217 of *LNCS*, pages 50–64. Springer, 1997.

[Vis96] W. Visser. Memory efficient state storage in SPIN. In *Proc. of SPIN Workshop*, pages 21–35, 1996.