State Compression in SPIN: Recursive Indexing and Compression Training Runs

Gerard J. Holzmann

Bell Laboratories, Lucent Technologies
Murray Hill, New Jersey 07974, U.S.A.

ABSTRACT

The verification algorithm of SPIN is based on an explicit enumeration of a subset of the reachable state-space of a system that is obtained through the formalization of a correctness requirement as an ω-automaton. This ω-automaton restricts the state-space to precisely the subset that may contain the counter-examples to the original correctness requirement, if they exist. This method of verification conforms to the method for automata-theoretic verification outlined in [VW86].

SPIN derives much of its power from an efficient implementation of the explicit state enumeration method in this context. The maximum number of reachable states that can be stored in main memory with this method, either explicitly or implicitly, then determines the maximum problem size that can be solved. We review some methods that have been tried to reduce the amount memory used per state, and describe some new methods that achieve relatively high compression rates.

Index Terms — Model Checking, Verification, State Compression.

1. INTRODUCTION

Given R reachable states, each consuming S bits of memory, no more than $log(R)$ bits of information need to be stored per state to distinguish between them. In theory, therefore, any system with $R.log(R) \leq M$ can be verified by exhaustive state enumeration. If the state-vectors are overlapped in memory, reusing common parts in a graph-like structure, the amount of memory used per state may even be less then $log(R)$.

To obtain good reductions with graph-like structures, one needs to find a reasonable *a priori* guess about the particulars of the reachable state space. If the guess is wrong, the memory requirements can increase, rather than decrease (cf. the variable ordering problem in BDD encodings [M93]). Similarly, for explicit state representations, one needs to find a mapping function that converts S bits of input into $log(R)$ bits of output, that is, a *minimal perfect hash function*, e.g. [MWHC96]. The number of reachable states R, however, is *a priori* unknown, and unguessable, and therefore also $log(R)$ is an unknown quantity, at least until after the verification has been completed, and the storage problem solved by other means.

Without compression techniques, the scope of verification runs by explicit state enumeration is limited by $R \leq M/S$, with S in general much larger than $log(R)$. Any reduction in the size of S will therefore increase this scope.

Every reachable state is uniquely characterized by a contiguous sequence of bits in memory. This sequence is called the *state-vector*. Above, we assumed that the state-vector is S bits long. In general, the length of the state-vectors that are stored can vary, as processes and channels are created or deleted during an execution of the system that is being verified. The problem we consider is to find a loss-less compression technique that allows us to store each state-vector in fewer than S bits.

We would like the compression to be as efficient as possible, but it is worth noting that there are no requirements on the efficiency of de-compression: in this application the compression *never* needs to be reversed.

Comparison of states, to determine if a state has previously been visited, for instance, can always be done in compressed form. SPIN's algorithm for cycle-detection [HPY96], and in general for the verification of temporal logic properties, is insensitive to the particular encoding of states, provided it is consistent (i.e., the encoding method does not change dynamically during the run).

Parts of the memory requirements of a verification system are not necessarily reduced by state-vector compression methods. For example, the memory required for the depth-first search stack, for the hash-table structure, and various other data structures that are used to store temporary information usually remain unchanged. We refer to this below as *extraneous data structures* or *overhead*. In a standard verification run without compression, this overhead is a relatively small part of the overall memory requirements. When aggressive state-vector compression methods are used, however, this overhead can easily become the larger part of the remaining memory requirements. In the results reported below, we always report the *total* memory requirements of verification runs before and after compression. Where possible, we will indicate what fraction of the final memory requirements is due to extraneous overhead, and what fraction due to state-vector storage.

In the following we look at the following methods.

- Predefined, static compression methods (Section 2).
- Recursive indexing methods (Section 3).
- Two-Phase compression methods (Section 4).
- Graph-Based encodings (Section 5).

2. Predefined, Static Compression

With this class of techniques, the state-vector is reduced in size with a single algorithm, that is independent of the particular verification problem being addressed. We look at three examples: run-length encoding, static Huffman compression, and byte masking. The performance of these three methods is compared in Table I.

2.1. Run-length Encoding

A straightforward technique is to apply direct run-length encoding to the contents of the state-vector. In all experiments discussed in this paper, the size of the state-vector is first rounded up to an integer multiple of bytes (the unit of storage). In the run-length encoding method, the compressed state-vector is encoded as a sequence of byte *pairs*, i.e., it is always an even number of bytes long. The first number of each pair gives the number of times that a byte value is repeated (as a number between 1 and 255). The second number gives the original byte value. Clearly, this method can save on memory only if *on average* each byte value appears in more than two consecutive positions in the state-vector. If this is not the case, the method can cause an increase in the size of the state-vector by up to a factor of two. This run-length encoding method was tried in [HGP92], and applied to a sample data transfer protocol (*dtp*) that we will also use in the experiments reported in this paper.

In [HGP92] it was found that the run-length encoding method significantly increased the run-time requirements for the search, in return for only a relatively small reduction of the memory requirements (Table I).

2.2. Static Huffman Compression

Another technique studied in [HGP92] was a predefined, static Huffman compression technique. To apply this technique, first the relative frequency of byte values in state-vectors was measured over a range of applications. These value were then used to predefine a dictionary for the Huffman encoding, with the most frequently occurring values assigned the shortest bit-codes. (Low byte values turned out to appear significantly more frequently than high byte values.) Bytes are then encoded in the stored vector with variable length bit-sequences, cf. [K73].

The results in Table I indicate that this encoding technique can achieve better compression rates than run-length encoding, at a somewhat lower run-time penalty.

2.3. Byte Masking

A much simpler predefined, static compression technique has been part of the distributed version of SPIN for several years. This method predefines a compression by identifying byte values in the state-vector that are known to contain constant values. Examples are padding bytes that are inserted by compilers when data structures are rounded to an integer multiple of a word-size, or the data-structures that are used to implement rendez-vous channels. (Rendez-vous channels can only contain data values during state transitions, but never before or after such a transition.) The constant fields are marked with a byte `Mask`. Before the state-vector is stored, only the non-marked bytes are copied, and the others are ignored.

The complete list of fields in the state vector that are masked is:

- the number of active processes and channels (`_nr_pr` and `_nr_qs`),
- process instantiation numbers (`_pid`'s); the process *type* identifier (`_t`) is not masked,
- unused elements of the state-vector, such as the fields used for cycle detection (when no cycle detection is used), fairness counters (when the fairness option is not selected),
- the channel contents for rendez-vous channels,
- all byte fields that are added by the compiler to secure proper alignment of variables and data structures on word boundaries.

To prevent false partial matches in state comparisons, the length of each state-vector is always prefixed to the state-vector, after compression. (Note that the length can change as new processes are instantiated.) Unless the maximum state-vector size was defined to be larger than 2^{16} bytes, this adds two bytes to the compressed state-vector (four for larger values). The masking operation is reversible, which guarantees that the compression is lossless. Note, for instance, that the size of a each process structure is determined by its type, and similarly, the value of the process instantiation numbers can be reconstructed from the order in which processes appear within the state vector.

When this method is applied to the same application that was used for the tests in [HGP92], we obtain a moderate reduction of the memory requirements for a slightly lower run-time penalty. Table I summarizes these findings.

TABLE I — PREDEFINED, STATIC COMPRESSION

Method	Run-Time	Memory-Use	Comments
0. No compression	100%	100.0%	`pan.c` compiled with `-DNOCOMP`
1. Run-Length	280%	91.5%	from Table II in [HGP92]
2. Static Huffman	205%	41.3%	from Table II in [HGP92]
3. Byte Masking	188%	83.6%	the default compression mode in SPIN

In [HGP92] we compared the compression ratios obtained by these methods against the reductions that could be obtained by the application of partial order reduction methods, and we concluded that partial order reduction was more effective as a reduction strategy. The reduction effects, however, are incremental. With a partial order reduction method in place [HP94], we would like to see how one can best reduce the memory requirements still further.

The results from Table I indicate that, compared to the default compression that is used in SPIN today, a considerably larger compression could be achievable for a not much greater run-time penalty. We consider several other alternatives for compression methods below.

3. Recursive Indexing

In SPIN version 2.7.4 (Sept. 1995) an experimental new compression mode was introduced that attempts to exploit a recursive indexing method for state-vector storage [H97, V97]. The method is based on the observation that the computational complexity of verification in asynchronous systems stems primarily from non-determinism of process executions. Each process and each data object can reach only a relatively small number of distinct values (or *local states*). The much larger number of reachable system states (*global states*) is obtained by the numerous ways in which these local states can be combined. Repeating a complete description of all local state-vector components in each global state is inherently wasteful.

3.1. SPIN's First COLLAPSE Mode

The COLLAPSE compression mode, as it was implemented in SPIN version 2.7.4, stores each local state-vector component, encoding the local state of a single process or channel, separately in the hash-table, and assigns it a unique index-number. The index-numbers for all components together form the compressed global state-vector.

This method can be applied recursively, by identifying smaller components of the process and channel states (e.g., local data objects within a process, or the separate message fields within a channel) and storing these with an extra level of indirection. The SPIN implementation uses only one level of indirection.

The list of state-vector components is defined as follows. Each component is stored separately in the hash-table, numbered, and represented by this number in the global state-vector.

- the first component encodes the current state of all global variables and all message channels combined, it also includes a length field for the state-vector as a whole.
- the remaining components encode the state of each active process separately, together with its local variables, but excluding locally declared channels.

In the original implementation of this method, the global variables and the message channels were stored under separate indices. This variation is still available in SPIN version 2.9.5 under compile-time option `-DSEPQS`. In most cases, however, this choice turns out to increase the memory requirements, and it was therefore not included in the measurements that follow.

A shortcoming of the first implementation of this method was that the user had to set an upper-bound on the largest index-number for the component states that was expected. The bound had to allow for an index to be stored in either one, two, three, or four bytes (the default being the most conservative estimate of four bytes, allowing for index numbers up to 2^{32}). If this method is applied to the application from Table I, using two bytes per index, we measure an increase of the run-time to 175%, combined with a reduction of the memory requirements to just 24.6% (see Table II below), which makes it superior to all methods listed in Table I. The required guess of the range of the component indices, however, is an annoyance that has so far prohibited this method from being promoted to the default compression mode in SPIN.

3.2. The Recursive Indexing Method (The Revised COLLAPSE Mode)

One way to avoid the requirement to specify an upper bound on the number of component indices is to store both the number of components and the number of bytes used per component index in the global state-vector. The numbers can now be allowed to change between state-vectors, because the precise method of compression (and decompression) for each state-vector can always be uniquely determined for each such vector. The number of bytes used per index is never less than one and never more than four. To store this information, just two bits of memory per component are needed. In the revised implementation of SPIN's COLLAPSE mode, this table of index widths is added to the global variables component. The width of the global variables component itself is stored as a separate byte in the global state-vector. This vector now contains:

- one *local* index (of between one and four bytes) for each active process that identifies the component that contains its state, including all its local variables, but excluding locally declared channels,
- one *global* index (between one and four bytes) for a single component that contains all global variables, all message channels, and the table of index widths for all *local* indices, and
- one byte that specifies the number of bytes used for the *global* index.

To reverse the encoding (to show that it is lossless), we can first use the last byte to determine the size of the *global* index. By retrieving the global component at this index, we can read the table of local indices, and retrieve and restore the local components one by one. We refer to this encoding method as *recursive indexing*.

There are several other details that have to be taken care of when this method is implemented. First of all, the length of each separately stored component has to be stored again with the component, to make sure no false partial matches can occur. Secondly, we can observe that if different types of components are (or can be) guaranteed to form disjoint sets, and the position of the corresponding indices in the compressed state-vector uniquely identifies the associated component-type, then these components can use separate index-

counts. This helps to keep the average value of these counts smaller, and can thus lower the number of bytes required per index in some cases. The current implementation of the recursive indexing method uses three different index-counters for three types of component indices that are always distinguishable: one counter for the global components, one for process components, and one for message channel components (when they are not merged with the global components, under compile-time option `-DSEPQS` as before).

TABLE II — BYTE MASKING AND RECURSIVE INDEXING COMPARED

Method	Run-Time	Memory-Use	Comments
3. Byte Masking	188%	83.6%	the default compression mode in SPIN
4. Original Collapse	175%	24.6%	using two bytes per index, predefined
5. Recursive Indexing	280%	18.3%	new collapse mode SPIN version 2.9.5

Table II compares the last two compression methods with the byte masking method from Table I. The recursive indexing method achieves the greatest reduction, also compared with all other methods from Table I, but it also incurs the greatest run-time penalty.

4. Two-Phase Compression

Recursive indexing can be pushed one step further still. Consider again the static Huffman compression method from Table I. In this case, the compression dictionary was precomputed, by measuring and averaging the byte-value frequencies for a number of verification runs. Would it be possible to use the actual byte frequencies for the specific verification run we are interested in to obtain still greater reductions?

If we accept run-time penalties that double or triple the cost of a verification run, as in Table II, the following technique also becomes a viable alternative.

- We always perform at least two verification runs. The first run is a *training run*, where statistics on the particulars of the reachable state space are collected. This training run can either be an approximate `BITSTATE` run, that samples a large fraction of the reachable state space, or it can be a fast exhaustive run without compression, that either terminates normally (and hence makes further verification attempts superfluous), or it terminates by running out of memory.
- At the end of the training run, statistics for the part of the reachable state space that was visited are used to generate code for a custom-made compression method that is used in subsequent runs. These new runs perform the actual verification in minimal memory.
- Because the statistics are incomplete (being based on just part of the reachable state space, visited in the training run), they may have to be adjusted during the second run. In many cases this can be done on-the-fly, without interrupting the run. In some cases, though, this may require an adjustment of the compression code, followed by an (automatic) recompile and restart of the run.

An experimental version of this method was implemented in SPIN (i.e., it is not in the distributed version just yet). The method uses the results of the training run to build a lookup table that stores for each byte position the number of distinct byte-values seen in that position of the uncompressed state-vector. This lookup table is used to compute the number of bits needed to store each individual byte (i.e., the log of the number of distinct values) and defines a mapping from each byte value encountered to an index value within this range.

If during the final verification run a byte-value is encountered that was not seen in the training run, it can be added to the table, provided that not all available values for the corresponding byte position were used. In the worst case, the number of bits allocated to the byte position will have to be incremented by one (doubling the number of available values that can appear in that position) and the code recompiled. The likelihood of this happening will depend on the accuracy of the statistics collected in the training run. In practice, even a very superficial training run turns out to collect statistics of good enough quality to make the occurrence of this event rare.

The different phases of the verification (i.e., two or more) are performed without manual intervention: at the end of the first phase, a new piece of C-code is generated on the fly, and the `pan.c` source is automatically recompiled and linked with this source. The resulting executable verifier is then (also automatically) restarted with the same run-time options that were used in the first phase. In the rare case that the lookup

table has to be adjusted during the final verification run, only the separate lookup-table code has to be recompiled and relinked (not the complete verifier). In that case, the final verification run is restarted from the beginning, as before.

Table III compares the performance of this method with the recursive indexing method from Table II, for the same application as used for Tables I and II. (Table III lists the run-time expense and memory requirements of the final verification run for the two-phase compression method.) It can be seen that the run-time requirements are increased significantly, especially if we also take the training runs into account. The memory requirements are reduced only slightly in this case.

TABLE III — RECURSIVE INDEXING AND TWO-PHASE COMPRESSION COMPARED

Method	Run-Time	Memory-Use	Comments
5. Recursive Indexing	280%	18.3%	new collapse mode SPIN version 2.9.5
6. Two-Phase Compression	388%	16.0%	final verification run only

To assess the relative performance of the different compression methods we have discussed, a single application is quite inadequate. We have therefore repeated the experiments for the last two techniques for seventeen different applications. The results of these tests are summarized in Table IV. All comparisons are against an exhaustive run without compression, (cf. the first row of Table I).

TABLE IV — RELATIVE PERFORMANCE OF COMPRESSION METHODS

Application	State-Vector	Nr of States	RI-Time	TP-Time	RI-Memory	TP-Memory
Cambridge Ring	56	1,252,660	**159.51%**	208.61%	35.34%	**28.28%**
Dining Philosophers(4)	160	81,504	448.76%	**339.95%**	32.94%	**25.71%**
Data Transfer	140	251,409	**186.67%**	387.65%	18.26%	**16.04%**
Hajek's protocol	52	116,087	**198.44%**	268.23%	42.84%	**30.89%**
Leader Election(5)	196	45,885	**255.17%**	560.34%	19.63%	19.63%
Future-Bus	136	134,444	**137.95%**	210.04%	**24.05%**	26.83%
Logical Link Control	112	19,407	**173.56%**	282.76%	50.00%	**48.35%**
File Transfer	144	439,895	**141.07%**	242.18%	17.50%	**14.60%**
Partial Order Test	52	682,120	162.31%	**122.93%**	77.92%	**20.76%**
Snooping Cache	112	91,920	**244.67%**	323.08%	32.97%	**22.29%**
Sorting	192	659,683	**415.03%**	470.57%	**12.74%**	14.71%
Sliding Window	48	402,997	**174.93%**	270.00%	39.84%	**33.15%**
Test 1	52	100,001	210.92%	**200.50%**	44.04%	**33.03%**
Test 2	72	100,001	**183.65%**	208.65%	44.45%	**34.27%**
Test 3	56	100,001	**177.32%**	186.06%	41.74%	**31.30%**
Phone Switch Model	60	3,918,290	**247.79%**	679.68%	39.04%	**18.21%**
Universal Receiver	120	19,339	**198.70%**	301.30%	53.46%	**44.83%**
Average	104	495,038	**218.62%**	309.561%	36.87%	**27.23%**

In Table IV, the columns marked `RI` give the results for the recursive indexing method (Table III, the row marked 5), and the columns marked `TP` give the results for the two-phase compression method (Table III, the row marked 6). Since only the relative performance of the various methods is significant, only those metrics are represented in the table. In all cases, we give the overall, total memory requirements for the verification run, including all memory used for extraneous data structures that is not affected by the specific compression method applied (such as the memory used for the depth-first search stack, for the hash-table etc.). Although the reductions for just the state storage, seen in isolation, are greater than is evident from these numbers, only the numbers for overall memory usage determine the relative merit of each method.

The amount of memory used for other data structures (i.e., other than the state vector contents proper) depends in part on the compression method that is chosen. It can include the memory required to number states, to add a length field, and to round state-vectors up to the nearest word-boundary. For the more aggressive compression methods, this extraneous data dominates the total memory requirements.

The best result in each row of Table IV is indicated in bold, and the average over all rows is shown in the bottom row. Overall, both the reduction and the run-time overhead of the two-phase method is greater than that of the recursive indexing method.

The best memory reduction is obtained with the recursive indexing method for the `Sorting` application: a reduction to 12.74% of the original memory requirements (the overhead consumes 4.9% of the memory requirements in this case). The two-phase method, however, achieves greater reductions than the recursive indexing method in all but two cases (`Future-Bus` and `Sorting`). The recursive indexing method, on the other hand, has lower run-time overhead in all but two cases (`Partial Order Test` and `Test 1`) or of course in all cases if we factor in the time needed for the first phase of the two-phase method.

In most cases, the reductions achieved by the two methods differ in no more than roughly 10%. In two cases the difference is larger (`Partial Order Test` and the `Phone Switch Model`), with the two-phase method performing considerably better than the recursive indexing method.

The run-time overhead is doubled or tripled when these compression techniques are used. In [V97] a direct application of BDD-based encodings was studied, which gave similar reductions in the memory requirements, but a run-time overhead of roughly an order of magnitude greater. In [G97] a more generic graph-based encoding was used, which gave similar run-time overhead, but greater memory reduction. (We briefly return to this in Section 5.)

Some of the results from Table IV are shown in graphical form in Figure 1. In this figure, the compression ratios are plotted against the state-vector sizes. Each bullet indicates a data-point from Table IV. For reference, we also connected each bullet with a vertical line to the point that indicates the memory overhead that is contributed by extraneous data in a run without compression of the state-vectors. This overhead is expressed as a percentage of total memory use as:

$$100\times\left\lceil\frac{M-S\times R}{M}\right\rceil\%$$

where M is the total amount of memory used, S is the state-vector size, and R is the number of reachable states. Different compression schemes can incur different overhead, so at best this is only an estimate of the best performance that could be expected.

The compression ratios appear to approach the lower-bound set by the extraneous overhead better as the state-vectors get larger. Overall, the two-phase compression method can also be seen to behave more predictably.

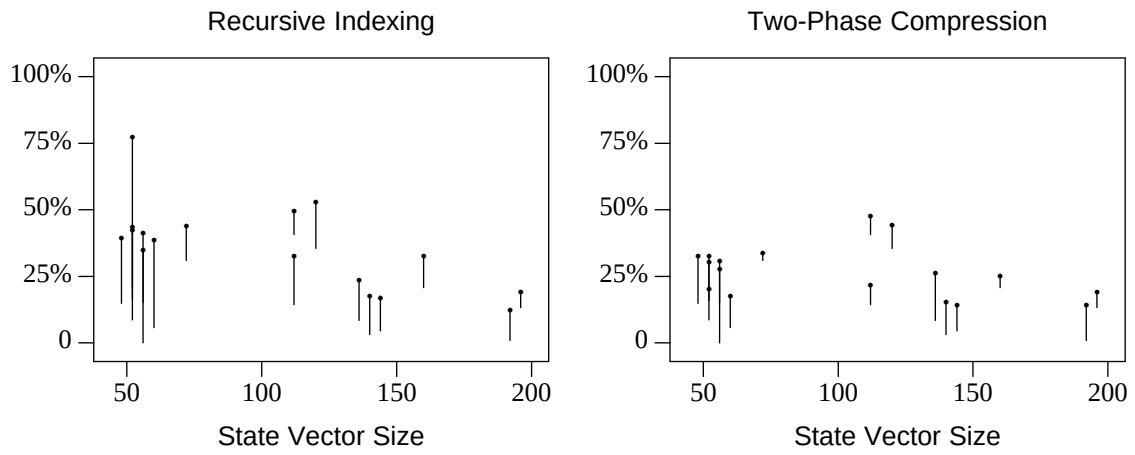

Figure 1. Memory Reduction in Percent and State Vector Size

5. Graph-Based Encodings

Greater reductions that what we have described so far can become possible with a compression method that is based on the following observations:

- Each state-vector is a finite sequence of either byte values or integer component indices. Such a sequence may be considered a finite word over a finite alphabet of integers.

- Instead of storing each state-vector separately, with or without compression, we can build the finite state automaton that accepts all finite words that correspond to state-vectors previously seen, and that is extended with each new word encountered. This finite state automaton can be kept in minimal form to reduce the memory requirements.

The state acceptor can be encoded as a finite, directed, acyclic graph, very similar to a generalized BDD. The use of such acyclic graph structures in the context of SPIN was investigated in [G97] and in [V97]. In both studies it was noted that the direct construction of a graph structure based on the uninterpreted bits in a state descriptor (as is done for BDD construction in hardware model checking) is ineffective when model checking asynchronous systems. A possible reason is that the state-vectors in SPIN contain descriptors for higher level data structures, encoding relatively small numbers of specific values. Unless such data structures are grouped and stored as units, the overhead of the graph structure (i.e., the pointers connecting the nodes) tends to dominate the storage requirements, and increase rather than decrease overall memory use.

The memory reduction obtained with the techniques explored here appear to be similar to those obtained with the graph based techniques in [V97], but smaller than those in [G97]. The greater reductions obtained in [G97] appear in large part to be du to a lower overhead, i.e., the hash-table structure can be completely avoided with the graph-based encodings. (At the time of writing, experiments with graph-based techniques for possible inclusion in SPIN are still being pursued.)

All measurements we have made with the techniques described in this paper indicate only a linear, or slightly better than linear, reduction of the memory requirements. Ideally, of course, one would like to see exponential size reductions. Are there any indications that this could be feasible? We briefly look at this point in the next section.

5.1. Compression Combined with Partial Order Reduction

The three graphs on the left-hand side of Figure 2 display the results of a measurement for the leader election protocols, for varying numbers of participating processes. The three graphs on the righ-hand side give the results for the classic dining philosophers problem.

The top graph in each column shows the number of reachable states for an exhaustive run and for a run where *partial order reduction* is applied [HP94]. In one case, the improvement of the partial order reduction method is linear, but in the other case the improvement is exponential. For the leader election protocol, the partial order reduction algorithm succeeds in reducing exponential growth to linear growth, which makes it possible to verify the properties of this application for virtually any number of participating processes. The remaining graphs show the effect of the recursive indexing technique *without* partial order reduction. The effect of the compression is slightly better than linear in both cases. We consider in the following subsection whether this is necessarily the case.

In [V97] it was observed that the effect of the memory reductions for a BDD-based encoding were diminished when SPIN's partial order reduction algorithm was enabled. No such effect was observed with the methods studied here. For example, when partial order reduction is enabled, the leader election protocol from Figure 2 can be verified with 100 participating processes in about 6 seconds of CPU time, generating 1504 reachable states. Each state in this example is encoded in a state-vector of 42 Kbytes, before compression. (This verification run is equivalent to an exhaustive run with in the order of 10^{14} reachable states of the same size.) The recursive indexing method reduces the state-vector from 41,712 bytes to an average of 200 bytes, or just 0.4% of the original size. The overall memory requirements, however, are reduced no further than to about 4% in this case, due to the memory used for extraneous data structures.

The results in Figure 2 show the effect of compression for the state space generated in exhaustive runs of the same protocol with smaller numbers of processes. For the run with six processes, for instance, the recursive indexing method reduces the state-vectors from 256 bytes to 24 bytes, or to about 10%. Therefore, in this case, when the state-vector size grows from 256 to 41,712, the compression ratio **improves** from 10% to 0.4%. This means that the effect of the compression on the memory requirements need not remain fixed, but can indeed increase with problem size. The trend is also detectable, though less prominent, for the relatively small range of state-vector sizes in the applications that produced the data for Table IV and Figure 1.

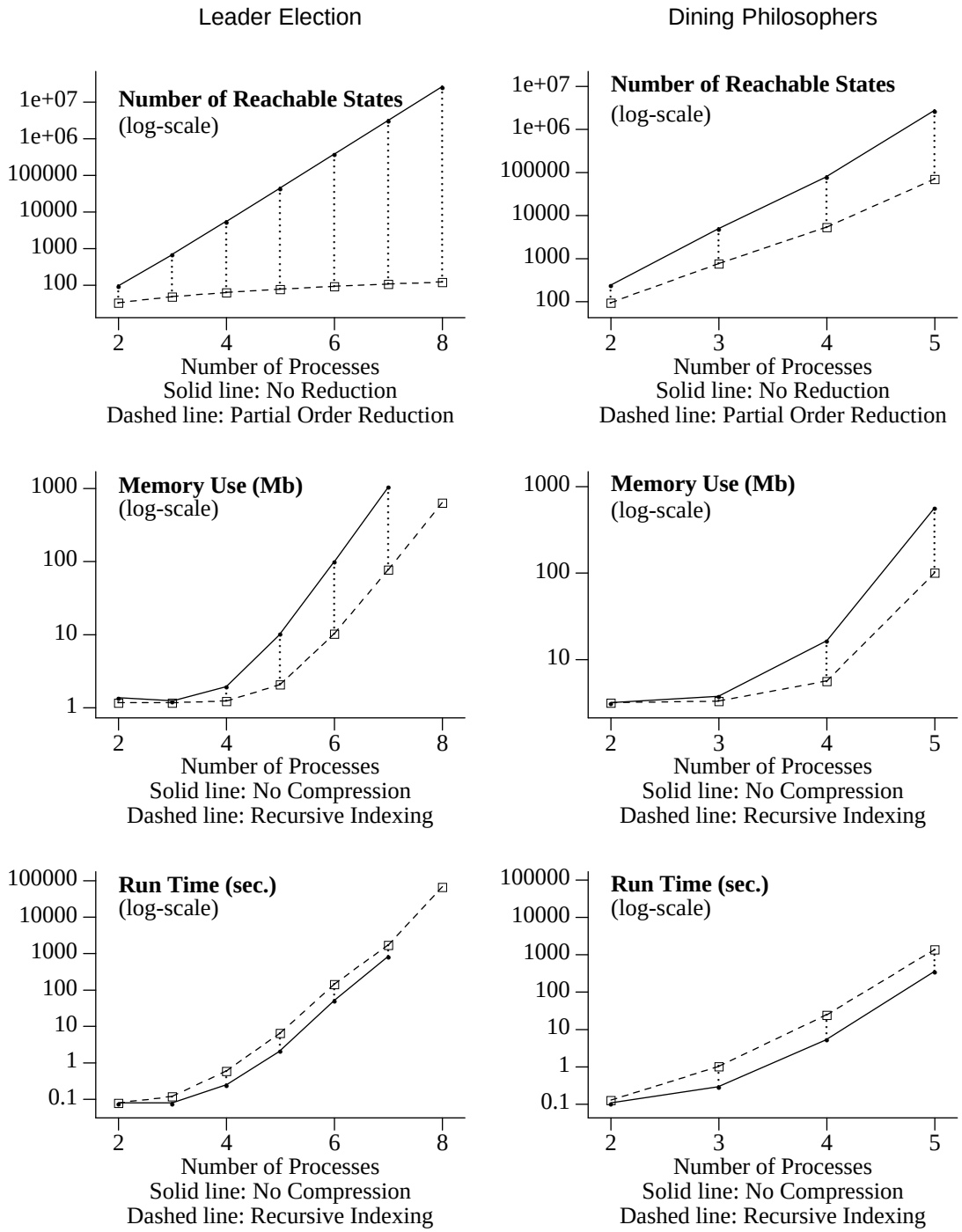

Figure 2. Recursive Index Compression and Problem Size (Exhaustive Search).

6. Conclusions

We have discussed three different methods to perform state-vector compression in systems such as SPIN, and briefly compared the most promising of these with a fourth method, based on graph encodings. Predefined, static compression methods were found to behave poorly in practice; combining a small reduction in memory use, with a relatively large run-time overhead. The recursive indexing method given in this paper, is a relatively simple and robust method that can give good reductions, in return for a doubling or tripling of

the run-time costs.

The recursive indexing method requires no major restructuring of the standard hash-table storage method. The state-vector is recursively split into smaller components, with each level in this recursion hierarchy stored in a conventional way. Frequently occurring components are stored just once, and can be referred to multiple times, this giving the potential for savings that increase with problem size.

A more effective compression method, that is based on a two-phase verification method, was also discussed. This method uses an initial *training run* to learn the appropriate statistics about the state space for the specific verification problem at hand, and then uses these statistics in the second run to optimize the compression strategy. In a trial implementation, the two-phase verification process could be completely automated, and hidden from the user. The overall run-time requirements of this type of search are predictably at least twice that of a single run. The savings are in almost all cases larger than those obtained with the recursive indexing method, though not dramatically so. The two-phase method may also be combined with the computation of an appropriate encoding for a *GETS* structure [G97], avoiding the need for manual intervention in their construction, and avoiding the need for hash-table structures entirely, thus opening the way for further substantial improvements.

The recursive indexing method is implemented in SPIN Version 2.9.5, and can be used to override the default *byte-masking* method by compiling the verifier sources in `pan.c` with the additional directive `-DCOLLAPSE`.

Acknowledgements

I am grateful to Willem Visser and Jean-Charles Grégoire for freely sharing their code and insights with me on their earlier work. Both have also provided sage feedback on this paper.

7. References

[G97] J.C. Grégoire, State space compression with GETSs. *Proc. 2nd SPIN Workshop*, Held August 15, 1996, Rutgers University, New Jersey. DIMACS Series No. 32. American Mathematical Society, 1997.

[HGP92] G.J. Holzmann, P. Godefroid, and D. Pirottin, Coverage preserving reduction strategies for reachability analysis. *Proc. 12th Int. Conf. on Protocol Specification, Testing, and Verification*, PSTV92, Orlando, Florida, June 1992.

[HP94] G.J. Holzmann, D. Peled, An improvement in formal verification, *Proc. Conf. on Formal Description Techniques*, FORTE94, Berne, Switzerland, 1994.

[HPY96] G.J. Holzmann, D. Peled, M. Yannakakis, On nested depth-first search, *Proc. 2nd SPIN Workshop*, Held August 15, 1996, Rutgers University, New Jersey. DIMACS Series No. 32. American Mathematical Society, 1997.

[H97] G.J. Holzmann, The model checker SPIN. IEEE Trans. on Software Engineering, to appear 1997.

[K73] D.E. Knuth, *The Art of Computer Programming*, Vol 1, Addison-Wesley, 1973.

[MWHC96] B.S. Majewski, N.C. Wormald, G. Havas, Z.J. Czech, A family of perfect hashing methods, *The Computer Journal*, Vol 39, No. 6, pp. 547-554.

[M93] K. McMillan, *Symbolic model checking*. Kluwer Academic Publishers, 1993.

[VW86] M.Y. Vardi, P. Wolper, An Automata-Theoretic Approach to Automatic Program Verification, *Proc. First Symposium on Logic in Computer Science*, Cambridge, June 1986, pp. 322-331.

[V97] W. Visser, Memory efficient storage in SPIN. *Proc. 2nd SPIN Workshop*, Held August 15, 1996, Rutgers University, New Jersey. DIMACS Series No. 32. American Mathematical Society, 1997.