A Nearly Memory-optimal Data Structure for Sets and Mappings

Jaco Geldenhuys and Antti Valmari

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

Abstract. A novel, very memory-efficient hash table structure for representing a set of bit vectors — such as the set of reachable states of a system — is presented. Let the length of the bit vectors be w. There is an information-theoretic lower bound on the average memory consumption of any data structure that is capable of representing a set of at most n such bit vectors. We prove that, except in extreme cases, this bound is within 10% of $nw - n \log_2 n + n$. In many cases this is much smaller than what typical implementations of hash tables, binary trees, etc., require, because they already use nw bits for representing just the payload. We give both theoretical and experimental evidence that our data structure can attain an estimated performance of $1.07nw - 1.05n \log_2 n + 6.12n$, which is close to the lower bound and much better than nw for many useful values of n and w. We show how the data structure can be extended to mappings while retaining its good performance. Furthermore, our data structure is not unduly slow.

1 Introduction

The greatest obstacle faced by automated verification techniques based on state exploration is the *state explosion problem*: the number of states grows exponentially in the number of components. Unfortunately, explicit state enumeration approaches require that the set of reachable states is stored.

In this paper we consider the case where the number of states is much smaller than the size of the universe from which the states are drawn, as is typical in state exploration. We address the question *how close can we come to the absolute lower bound on the memory needed in this kind of situation, without an unbearable increase in the running time?*

The literature contains many approaches that attempt to strike a balance between optimal use of available memory and reasonable runtime costs. Examples include state compaction [3] and byte masking, run-length encoding, and Huffman encoding [7], closed (rehashing) hash tables, open (chaining) hash tables [4], bit-state hashing [6], hash compaction [14, 17], recursive indexing [8, 16], minimized automata-based techniques [9], sharing trees (or GETSs) [5], queue-content decision diagrams (QDDs) [1], and difference compression [12].

Many of these methods rely on a certain degree of regularity in the structure of the reachable states. For example, recursive indexing relies on the fact that various subparts of the state take on only a few distinct values, resulting in a distribution with “clusters”.

Some of the data structures store the entire state vectors. This means that their memory consumption is subsumed by that of *linear storage*, a data structure that stores states consecutively as a string of bits. It is, however, possible to significantly reduce memory consumption below that of linear storage.

We describe a novel data structure that minimizes the memory overhead associated with the management of its data, and makes no assumptions about its distribution. It is therefore not application-specific. The design of the data structure emphasizes memory efficiency over time efficiency. Our approach is a variant of closed hashing with linked lists for collision handling [10, Section 6.4]. To minimize memory usage, we (1) use a hashing function that makes it unnecessary to store whole elements (see [10, Exercise 6.4.13]), and (2) allocate memory in two kinds of blocks that can hold several entries, thus eliminating the need for most pointers.

The data structure is perhaps not as fast as many of its competitors, but not woefully slow either. It can also save time by allowing an analysis that would otherwise require the use of slow disk memory, to complete in fast central memory. Saving memory can mean saving time if it reduces the amount of swapping performed by the operating system. The primary objective of the design then, is to make optimal use of available memory to store as many states as possible; making the operations as fast as possible is only a secondary objective.

In Section 2 we discuss the information-theoretic lower bound on the memory required for representing a set of reachable states, and introduce approximations that we use later on. Section 3 describes the data structure and its operations, Section 4 relates a series of simulations we performed to investigate the parameters for optimal performance, and Section 5 analyses the memory consumption of the data structure theoretically. A test implementation of the data structure inside an existing tool for parallel composition is described in Section 6 and measurements obtained from actual models are presented. Lastly, conclusions are presented in Section 7.

2 Information-theoretic Bounds

It is possible to derive an absolute information-theoretic lower bound for the amount of memory needed for representing a small subset of a large set U (called the universe). The number of subsets of size n of U is $\binom{|U|}{n}$. A data structure that can represent each of these subsets must use at least

$$M_{low} = \log_2 \left(\frac{|U|!}{n!(|U|-n)!} \right)$$

bits on the average. Some sets of size n may have shorter representations, but, if that is the case, then some others need more than M_{low} bits.

In many applications, such as explicit state space construction, the set grows to its final size in several steps. It is therefore also interesting to consider a data structure that is able to represent all subsets that have at most n elements. Then the number of bits it uses on the average is at least

$$M_{opt} = \log_2 \left(\sum_{k=0}^{n} \frac{|U|!}{k!(|U|-k)!} \right) .$$

However, when $n \ll |U|$, the value of M_{opt} differs very little from M_{low}. So, in practice, M_{low} is a valid limit also in this case.

While these formulations allow us to compute lower bounds numerically, they are cumbersome to manipulate. Therefore, we derive a simple but reasonably precise estimate that will allow us greater flexibility. Let w be the number of bits in the representation of an individual element of the set U. There are therefore $|U| = 2^w$ possible elements in the universe.

Let $w' = w - \log_2 n$. We shall use the following estimate for the information-theoretic lower bound:

$$M_{est} = nw' + n.$$

Before discussing the accuracy of this estimate, let us give it an intuitive meaning. Storing the n elements as such in an array would take n locations of w bits each, totalling nw bits. We call this technique *linear storage*. If the elements are explicitly stored in a hash table, binary tree or other similar structure, then nw bits are needed for them, and additional bits are needed for pointers, etc. We see that nw is a lower bound for any data structure that stores each element of the set explicitly. However, when n is large, $nw' + n \ll nw$. This means that it should be possible to represent large sets much more densely than linear storage does.

We claim the following about the accuracy of our estimate, namely that the absolute lower bound is within 10% of the estimate, when $w' \geq 5$ and $n \geq 7$.

Theorem 1. *$0.9M_{est} \leq M_{low} \leq M_{opt} \leq 1.1M_{est}$ when $w' \geq 5$ and $n \geq 7$.*

Proof. First we show that $M_{low} \geq 0.9M_{est}$. Making use of the fact that

$$\begin{aligned}
\log_2\left[(2^w)!/(2^w-n)!\right] &= \log_2\left[2^w(2^w-1)\cdots(2^w-n+1)\right] \\
&\geq \log_2(2^w-n)^n \\
&= \log_2(n(2^{w'}-1))^n \\
&= n\log_2 n + n\log_2(2^{w'}-1) \\
&= n\log_2 n + nw' - \varepsilon n
\end{aligned}$$

where $\varepsilon = \log_2(2^{w'}/(2^{w'}-1)) < 0.1$ for all $w' \geq 4$, and the fact that $\log_2 n! \leq n(\log_2 n - 0.5)$ when $n \geq 2$, we find that

$$\begin{aligned}
M_{low} &= \log_2\left[(2^w)!/\left(n!(2^w-n)!\right)\right] \\
&\geq n\log_2 n + nw' - \varepsilon n - \log_2 n! \\
&\geq n\log_2 n + nw' - \varepsilon n - n\log_2 n + 0.5n \\
&= nw' + \kappa n \ .
\end{aligned}$$

Since $\varepsilon < 0.1$ when $w' \geq 4$, we know that $\kappa = 0.5 - \varepsilon \geq 0.4$. Now, if $w' \geq 5$, $0.9M_{est} = 0.9nw' + 0.9n = nw' - 0.1nw' + 0.9n \leq nw' - 0.5n + 0.9n \leq M_{low}$.

The second inequality follows directly from the definitions of M_{low} and M_{opt}. Due to lack of space, we will not go into the details of proving the last inequality here, except to mention that it relies on the fact that $\log_2 n! \geq n(\log_2 n - \log_2 e)$ (based on Stirling's approximation), and on the proximity of M_{opt} and M_{low}. □

3 The Data Structure

We now describe a data structure, which we call a very tight hash table. Often state space exploration techniques rely on information about only the presence or absence of states, and then the data structure can be used to implement a set. Sometimes, however, tools also need to store associated data for each state; we discuss how very tight hash tables can be used to implement such mappings.

The data structure is called "very tight" because the primary objective is to get the most out of the available memory and come as close to the theoretical lower bound as possible. This goal is approached in two ways: first, only a part of each state is stored, and second, the design minimizes the overhead needed for the organization of the data.

The data structure was designed for the case where the elements are drawn from the uniform distribution. This assumption does *not* result in hash lists being roughly the same length; rather, they are distributed according to the binomial distribution. Because this assumption does not automatically hold in practice, a function may have to be applied to the elements to "randomize" them. The purpose of the function is to have as many bits of the element as possible affect the bits that are eventually used as the index. Many such functions are available; one example is described in Section 6.1. In the rest of this section we assume that the assumption *does* hold.

3.1 The Implementation of Sets

A data structure D for representing a subset of U must support at least the following dictionary operations on the elements $s \in U$:

- $D.Add(s)$: Adds s to the subset.
- $D.Find(s)$: Determines whether s is in the subset.
- $D.Delete(s)$: Removes s from the subset.

Let each element $s \in U$ be represented in w bits. The basis of our proposal is to break an element $s = s_1 s_2 \cdots s_w$ into two parts: the *index* $s_1 \cdots s_i$ and the *entry* $s_{i+1} \cdots s_w$. The index contains $i = \mathsf{bii}$ (bits in index) bits and the entry contains $w - i = \mathsf{bie}$ (bits in entry) bits.

The available memory is partitioned into two parts. The greater part, called the *basic region* is subdivided into 2^{bii} *basic blocks*. An element's index determines the basic block where the element's entry is stored, if possible. Each basic block contains sib (slots in basic block) slots where up to sib entries can be stored.

A basic block may of course fill up, and this is where the second part of the memory, the *overflow region* comes into the picture. It, too, is subdivided into blocks: there are nob (number of overflow blocks) blocks, containing sio (slots in overflow block) slots. The overflow blocks are allocated as needed to accommodate those entries that cannot fit into their basic blocks. When an overflow block fills up, another overflow block is allocated to store the extra entries. A basic block together with its overflow blocks constitute a *hash list*.

The very tight hash table is fully specified by the values of the five parameters: bii, bie, sib, nob, and sio. This organization of memory (depicted in Fig. 1) was selected to expend as few bits as possible on the organization of the data structure. Given the parameters, a very tight hash table has altogether

$$\begin{array}{rlr} \mathit{capacity} = & 2^{\mathsf{bii}} \cdot \mathsf{sib} & \text{basic block slots} \\ & + \mathsf{nob} \cdot \mathsf{sio} & \text{overflow block slots} \end{array}$$

slots. They are numbered from 0 to $\mathit{capacity} - 1$. The practical capacity of the data structure is less than *capacity*, because some blocks may be partially empty when the overflow blocks are exhausted. According to our experiments in Section 4, the practical capacity varies from 80 to 98% of the theoretical capacity.

The parameters have a critical effect on the memory consumption. Each basic block contains $\mathsf{sib} \cdot \mathsf{bie}$ bits. In addition, a counter of $\lceil \log_2(2+\mathsf{sib}) \rceil$ bits is needed to indicate the number of entries in the block along with two special conditions: that the block is empty and that it overflows. Overflow blocks contain $\mathsf{sio} \cdot \mathsf{bie}$ bits and a counter of $\lceil \log_2(1+\mathsf{sio}) \rceil$ bits. It is unnecessary to indicate that an overflow block is empty; it is either unallocated or contains at least one entry.

Overflow blocks further contain a link of $\lceil \log_2 \mathsf{nob} \rceil$ bits that indicates the number of the overflow block into which it overflows (if it does). Basic blocks do not have such a link since those bits would be wasted in blocks that do not overflow. Instead, an overflowing basic block stores its overflow link in the first bits of the block. The entry bits overwritten by the link are moved to the as yet unused link bits of the overflow block. When it, in turn, overflows, the bits are moved to link bits of the next overflow block, and so on. Consequently, the last link field in a hash list always stores entry bits that correspond to the first slot(s) of the basic block. Unused overflow blocks are linked together in a free list using the link bits as a pointer to the next block in the list. When an overflow occurs, the first block of the free list is allocated and removed from the list.

In total, the data structure uses

$$\begin{array}{lr} 2^{\mathsf{bii}} \cdot (& \\ \quad \mathsf{sib} \cdot \mathsf{bie} & \text{basic block entries} \\ \quad + \lceil \log_2(2+\mathsf{sib}) \rceil) & \text{basic block counter} \\ + \mathsf{nob} \cdot (& \\ \quad \mathsf{sio} \cdot \mathsf{bie} & \text{overflow block entries} \\ \quad + \lceil \log_2(1+\mathsf{sio}) \rceil & \text{overflow block counter} \\ \quad + \lceil \log_2 \mathsf{nob} \rceil) & \text{overflow block link} \\ + \lceil \log_2(1+\mathsf{nob}) \rceil & \text{free list start link} \end{array}$$

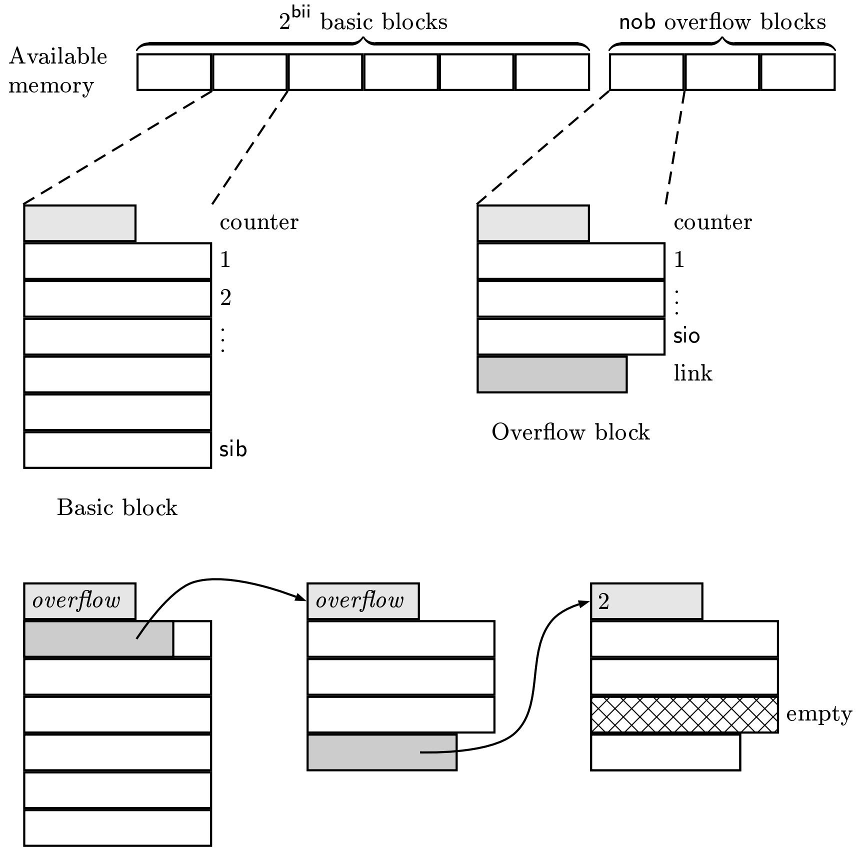

Fig. 1. Organization of very tight hash tables. The three blocks at the bottom show a hash list that overflows into two overflow blocks. The initial bits of the basic block entries are moved to the link field of the last block of the list to make room for a link.

bits of memory. We will address the selection of parameters in Section 4.

The workings of the *Find*, *Add*, and *Delete* operations are very similar: given an element, a linear search is performed through the hash list until the entry part of the element is located, or the end of the list is reached. *Find* simply reports the outcome of the search, while *Add* adds the entry if it wasn't located (possibly causing an overflow). When *Delete* finds an entry it moves the last entry of the hash list into the position of the deleted entry and decreases the size of the last block. If this block is an overflow block that becomes empty, it is deallocated and returned to the free list, and the appropriate adjustments are made to the new last block of the hash list.

All three operations have an average case asymptotic time complexity of $\Theta(n/2^{\mathsf{bii}})$, where n is the number of elements stored in the data structure. As with linked lists, *Find* can be made faster, at the cost of making *Add* and *Delete* slower, by maintaining the hash lists in order.

3.2 The Implementation of Mappings

For mappings, there is a second set A of associated data items. In this paper we treat the elements of A as opaque. The operations a data structure E for a mapping needs to support are very similar to those for a set:

- $E.Add(s, a)$: Adds the association (s, a) to the mapping.
- $E.Find(s)$: Returns the element a associated with s, or NIL if there is none.
- $E.Delete(s)$: Removes s from the mapping (i.e., maps it to NIL).

An obvious possibility for implementing a mapping is to store the associated data along with each entry either in the same array, or in a separate array in the place determined by the slot number of the entry. Because the practical capacity is less than the theoretical, a part of the array will be wasted. If the associated data items are large, it is of course possible to allocate them dynamically and store only pointers to them in the array, to reduce the wasted memory.

Alternatively, one can delegate the storage of the associated data to the user of the data structure, by returning the slot number after each *Add* and *Find* operation. A complication arises, however. If an element is deleted in the way described earlier, the last entry in the hash list is moved into the now empty position. The slot number of the last element therefore changes.

This problem can be circumvented by marking deleted entries with a special entry value, e.g., all zeros or all one's, instead of moving the last entry. This means that the special value is not usable as a regular entry. To compensate for this, it suffices to add just one extra bit to each basic block. This bit will indicate whether the special value is present in the hash list as a regular entry, even though it is never stored as such. Also, the special value must be given a special slot number that corresponds to no slot. Only 2^{bii} extra bits are needed to implement this scheme. An overflow block can be deallocated, when all its entries have been deleted. Because of this possibility, the performance penalty caused by deleted entries is less than with open addressing hash tables.

4 Simulations

To gain an understanding of how the choice of parameters influences the performance of the very tight hash tables, we ran two sets of simulations. In the first, random data is generated and the behaviour of our data structure is simulated, while the second simulation analyses the behaviour of the data structure given a theoretical distribution of the hash list lengths.

4.1 Simulation with Random Data

The first set of simulations consists of about ten thousand simulation experiments. In each experiment, random data items between 0 and $2^{\mathsf{bii}} - 1$ were generated. Counters were maintained to keep track of the length of each simulated hash list. Whenever a counter reached a value that corresponds to overflow,

Table 1. Simulations (with random data) for varying element and memory sizes

w	Memory	bii	sio	sib	Theoretical capacity	Practical capacity	Performance	Basic/overflow boundary
24	10	7	3	24	4562	4327	143%	78%
	100	12	3	17	59823	54106	159%	78%
	500	14	3	16	375246	332885	181%	79%
	2000	17	3	13	2090544	1715022	217%	80%
32	10	6	3	32	3027	2919	128%	81%
	100	10	3	23	35219	33290	134%	79%
	500	13	3	28	196239	185013	139%	81%
	2000	15	3	33	874837	822803	146%	83%
48	10	5	2	40	1841	1799	116%	84%
	100	8	3	52	19730	19320	119%	81%
	500	11	3	34	105862	102220	121%	80%
	2000	13	3	37	447053	431128	123%	82%
60	10	5	2	49	1426	1401	113%	81%
	100	8	3	53	15068	14801	115%	79%
	500	10	3	51	78903	77223	116%	81%
	2000	12	3	54	328167	321073	117%	82%

an extra counter that kept track of the number of simulated free overflow blocks was decremented. When this counter underflowed (reached -1), the simulation was terminated and the number of random data items was returned.

Each simulation experiment consisted of three simulation runs, of which the results of the best and worst were discarded. Simulation experiments were made with a wide range of combinations of values for bii, sio, and sib. The value of bie was equal to $w-\mathsf{bii}$ by definition, and nob was made as large as the given memory allowed. For each data width w and amount of memory, the average parameter values and performance results of the best five parameter combinations were reported. A representative sample of the results is given in Table 1. Memory is measured in 1000 bytes, and performance is M/M_{est}, where M is the amount of memory used.

Some interesting observations were made from the simulation results. Firstly, 3 proved to be an almost universally good value for sio. Very small values of sio lead to many links and large values result in many unused slots, so a "smallish" optimal value was expected. Another apparently universal observation is that about 80% of the available memory should be devoted to slots in basic blocks. The last column shows measured values for this percentage, that is, $2^{\mathsf{bii}} \cdot \mathsf{bie} \cdot \mathsf{sib}/\textit{total memory}$. When reading the column, one should note that when sib grows by one, the figure changes by several percentage units.

These observations can be used as guidelines when picking parameter values for a practical implementation. Of course, one should remember that they are not necessarily valid outside the data width and memory amount range of the simulations. Furthermore, the simulation ignores the fact that in reality a long list has a slightly smaller probability of growing than a short list, because only $|U|/2^{\mathsf{bii}}$ items can fall into each list. This effect is small when 2^{bie} is large compared to the list length.

4.2 Simulation with Ideal Distribution

To verify the results above, we ran a second set of simulations. Instead of generating random data, hash list lengths were calculated using the Poisson distribution, which is the limit of the binomial distribution when the number of elements grows large relative to the number of lists [11, 13]. If n elements are thrown into n/h lists at random with equal probability, the average list length becomes h, and the expected proportion of lists of length k is

$$\frac{n!}{k!(n-k)!}\left(\frac{h}{n}\right)^k\left(1-\frac{h}{n}\right)^{n-k} \approx_{(\text{when } n \gg k)} \frac{h^k}{k!}e^{-h}.$$

Given the number of elements and the data width w, the simulation calculated those values of bii, sio, and sib that result in the best performance.

Table 2 shows the outcome of the second set of simulations. Specifically, each row of the table describes a simulation experiment with the same data width and number of states (i.e., practical capacity) as that of its counterpart in Table 1. The third, fourth and fifth columns contain the calculated values of bii, sio, and sib that yield the best performance. Column six contains the average list length $h = n/2^{\mathsf{bii}}$. The memory (in 1000 byte units) needed by a very tight hash table with these parameters is given in column seven, and the last column gives the corresponding performance.

Once again, several interesting observations can be made. The values of bii and sio correspond closely to those of the first experiments, but the same is not true for sib. We hypothesize that the precise value of sib is not crucial, and that average list length can be used as a heuristic. In all cases the memory requirements are less than the available memory of the first experiment, and, most importantly, the performance figures match those of Table 1 to within three percentage points.

Not only does this second set of simulations confirm the results of the first set, but it also provides us with a convenient tool to derive parameters when the number of states is known, as is the case in the next section. Moreover, it is much faster to generate the hash list length distribution (in the order of 2^{bii} operations), than to produce the random data. This allows us not only to quickly find "good" parameters for the data structure, but also to estimate the expected performance for larger numbers of states.

5 An Approximation of Memory Consumption

We are now able to derive an approximative formula for the memory consumption for very tight hashing, subject to the following assumptions: We fix

Table 2. Simulations (with Poisson distribution) for varying element and memory sizes

w	States	bii	sio	sib	h	Memory	Performance
24	4327	8	2	15	16.9	10	142%
	54106	12	3	12	13.2	100	158%
	332885	15	3	10	10.2	497	179%
	1715022	18	3	7	6.54	1973	214%
32	2919	7	2	20	22.8	10	127%
	33290	10	3	28	32.5	100	133%
	185013	13	3	20	22.6	498	139%
	822803	15	3	23	25.1	1986	145%
48	1799	6	2	22	28.1	10	116%
	19320	9	2	31	37.7	100	119%
	102220	11	3	43	49.9	499	121%
	431128	13	3	46	52.6	1994	122%
60	1401	5	2	37	43.8	10	112%
	14801	9	2	23	28.9	100	114%
	77223	10	3	65	75.4	499	115%
	321073	12	3	68	78.4	1997	116%

$\mathsf{sio} = 3$ as suggested by the simulations, and we assign a value to bii such that the average list length $h = n/2^{\mathsf{bii}}$. Once bii is fixed, we set the value of sib to $\lfloor h \rfloor = h - \varepsilon$, where $0 \leq \varepsilon < 1$. Since $\mathsf{bii} = \log_2 n/h$, we know that $\mathsf{bie} = w - \mathsf{bii} = w - \log_2 n + \log_2 h = w' + \log_2 h$.

Let p_h be the proportion of lists that are shorter than or equal to the average length (i.e., lists of length $\leq h$), and let q_h be the proportion of elements in such lists.

Theorem 2. *Setting* $\mathsf{sio} = 3$*, the memory consumption of the very tight hashing data structure for an average set is*

$$M_{vth} < c_1 n w' + c_2 n \log_2 n + c_3 n,$$

where $r_h = 1 - q_h - (\lfloor h \rfloor - 2)(1 - p_h)/h$*, and*

$$\begin{aligned} c_1 &= \lfloor h \rfloor / h + r_h, \\ c_2 &= r_h/3, \text{ and} \\ c_3 &= c_1 \log_2 h + 1/h \lceil \log_2(\lfloor h \rfloor + 2) \rceil + c_2(3 + \log_2 c_2). \end{aligned}$$

Proof. Consider the graph in Figure 2. It shows the n/h lists of a hypothetical distribution arranged according to their lengths. Area B contains all those elements that are in underflow basic blocks, while area A represents the wasted slots of basic blocks. Area C contains those elements that fill their basic blocks and spill into overflow blocks represented by area D. By simple arithmetic we know that $C = \lfloor h \rfloor (1 - p_h) n/h$, and that $B = q_h n$, and, since $B + C + D = n$, that $D = n - (B + C) = n - n(q_h + \lfloor h \rfloor (1 - p_h)/h) = n(1 - q_h - \lfloor h \rfloor (1 - p_h)/h)$.

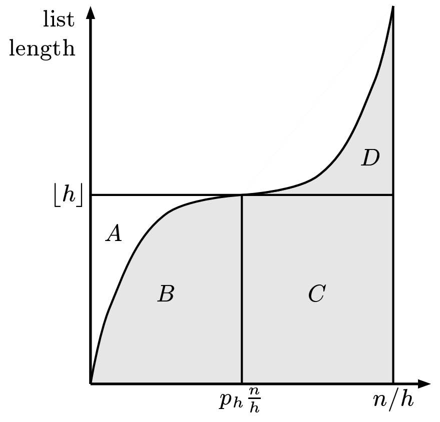

Fig. 2. The n/h lists of a typical distribution arranged according to length

The number of slots in overflow blocks is the number of overflow elements plus the number of wasted slots. Since $\mathsf{sio} = 3$, each overflow list contains at most 2 wasted slots. Let v denote the number of overflow slots. Then

$$\begin{aligned} v &= D + \text{wasted slots} \\ &\le n(1 - q_h - \lfloor h \rfloor (1-p_h)/h) + 2 \cdot (\text{number of overflow lists}) \\ &= n(1 - q_h - \lfloor h \rfloor (1-p_h)/h) + 2(1-p_h)n/h \\ &= nr_h \end{aligned}$$

The total memory consumption is therefore

$$\begin{aligned} M_{vth} = &\quad (\text{slots in basic blocks}) \cdot (\mathsf{bie}) \\ &+ (\text{number of basic blocks}) \cdot (\text{basic block counter size}) \\ &+ (\text{slots in overflow blocks}) \cdot (\mathsf{bie}) \\ &+ (\mathsf{nob}) \cdot (\text{2-bit counter size}) \\ &+ (\mathsf{nob}) \cdot (\lceil \log_2 \mathsf{nob} \rceil) \\ = &\quad (\lfloor h \rfloor n/h) \cdot (w' + \log_2 h) + (n/h) \cdot \lceil \log_2(\lfloor h \rfloor + 2) \rceil \\ &+ (v) \cdot (w' + \log_2 h) + (v/3) \cdot 2 + (v/3) \cdot \lceil \log_2(v/3) \rceil \\ < &\quad c_1 n w' + c_2 n \log_2 n + c_3 n \end{aligned}$$

□

While this theorem talks about the memory consumption of an average set, and not the average memory consumption of all sets, it still provides a good indication of the space needed. Table 3 lists the values of the constants for selected values of h. If the value of h is too small and the lists are too short, the performance of the data structure is poor, partly due to the ratio of data to overhead. On the other hand, if the lists are too long, too much time will be taken by linearly searching through them.

Table 3. M_{vth} upper bounds for various values of h

h	Upper bound
10	$1.21nw' + 0.07n\log_2 n + 4.31n$
20	$1.13nw' + 0.04n\log_2 n + 5.05n$
50	$1.07nw' + 0.02n\log_2 n + 6.12n$
100	$1.05nw' + 0.02n\log_2 n + 6.99n$
150	$1.04nw' + 0.01n\log_2 n + 7.52n$
200	$1.03nw' + 0.01n\log_2 n + 7.90n$

As in the last simulation, the Poisson distribution was used to compute the values of p_h and q_h:

$$p_h = \sum_{k=0}^{\lfloor h \rfloor} \frac{h^k}{k!} e^{-h} \qquad \text{and} \qquad q_h = \frac{1}{h} \sum_{k=0}^{\lfloor h \rfloor} k \frac{h^k}{k!} e^{-h}.$$

We have also computed p_h and q_h using the precise distribution for a wide range of n and w'. The values obtained were usually close to those in Table 3, and in almost all cases the differences were in such a direction that the precise distribution predicts a smaller memory consumption than Table 3.

The memory used for storing the payload is nw'. Thus, $c_1 - 1$ is the ratio of wasted slots to used slots. We see that the proportion of wasted slots becomes small as h grows. Unless n is unrealistically large (like 2^{60}), the $c_2 n \log_2 n$ term is insignificant.

6 Measurements with a Test Implementation

To demonstrate the feasibility of the data structure, we have incorporated it into an existing toolset, TVT [15] for the verification of concurrent systems. One of its components, `tvt.parallel`, computes the set of reachable states of a system. We have modified this program to make use of very tight hashing for storing the set of reachable states.

6.1 Implementation Details

The modified `tvt.parallel` tool uses very tight hashing to represent a mapping from states to auxiliary information about states. Of the previously discussed operations, only *Add* is required. In the implementation, *Add* returns a flag to indicate whether the state was really added or already present in the set. Since *Delete* is not used, the implementation need not maintain a free list of overflow blocks and it returns slot numbers with which the tool manages associated data. Overflow blocks are simply allocated from the first to the last.

The tool imposes a new requirement: it needs a *Retrieve* operation that, given the slot number, returns the original state. This is not a problem when the slot number is in the basic region, since a simple calculation yields the index part. However, when the slot number is in the overflow region there is no way to calculate the index short of scanning through all hash lists. For this reason, the index for each overflow block is stored in an extra field inside the block.

The implementation aligns the entries, counters and links on bit boundaries, thereby saving memory that might otherwise be lost to alignment.

States are randomized in the following way [2, p. 234]: Given the index and entry parts (s_I, s_E) of a state, a replacement index is computed as

$$s'_I = s_I \oplus ((a \cdot s_E + b) \bmod p),$$

where $\oplus$ represents the exclusive or operation, p is the smallest prime larger than 2^{bii}, and both a and b are chosen from the the range $\{1, 2, \ldots, p-1\}$.

6.2 The Results

The results of running the `tvt.parallel` program on four different models are shown in Table 4. The first column specifies a model parameter, such as the number of components. The second and third columns give the number of bits in the state and the number of states for each system. The fourth, fifth and sixth columns give the memory (in units of 1000 bytes and rounded to the nearest unit) and time (in seconds) used by the standard version of the tool, and its performance relative to the lower bound estimate (M/M_{est}). In the next few columns are the memory consumption when using very tight hash tables with and without the extra memory needed for the *Retrieve* operation, followed by the time and performance figures. The last column gives the value of nw/M_{est}.

The standard version of the tool stores states using an open hash table, in which each node in a hash list stores a state and a pointer to the next node. In addition to the set itself, a tool for computing the set of reachable states needs memory for other things, most notably for keeping track of incomplete work. The amount of memory needed depends on many factors, including the nature of additional services provided by the reachable state set data structure, like the *Retrieve* mentioned earlier. To keep the comparison to the theoretical lower bound straightforward, only the memory needed for representing the set is taken into account in the performance figures.

The parameters for the measurements were selected systematically according to the results of the simulations: sio was set to 3, bii was chosen so that the average list length h falls between 32 and 64, and sib was set to $\lfloor h \rfloor$. Even better results were obtained by handtuning the parameters and the randomization function, but they are not presented here since they would not be feasible in a practical setting.

Very tight hashing clearly outperforms the original data structure with respect to memory use. More importantly, the new data structure also outperforms the linear storage for all the models. As far as time consumption is concerned, very tight hashing fares slightly worse, but the difference does not appear to be significant.

Table 4. Memory and time consumption experiments

				Original			Very Tight Hashing				Linear
	Param.	Bits	States	Mem.	Time	Perf.	Mem.R.	Mem.	Time	Perf.	Perf.
Dining philosophers	8	24	6560	72	0.42	714%	15	15	0.53	148%	194%
	9	27	19682	156	1.38	461%	49	48	1.74	142%	196%
	10	30	59048	409	4.66	365%	158	156	5.96	139%	198%
	11	33	177146	1956	16.25	533%	521	514	19.81	140%	199%
Pipeline sort	7	36	2234	55	0.12	759%	9	9	0.15	126%	139%
	8	50	6469	98	0.41	314%	38	37	0.56	120%	130%
	9	56	18545	223	1.43	224%	120	119	1.83	120%	130%
	10	62	52746	809	5.17	259%	377	375	6.61	120%	131%
Sliding window protocol	11	27	3654	54	5.22	735%	11	10	5.38	140%	167%
	12	28	4622	61	13.33	622%	13	13	13.36	133%	166%
	13	29	5748	67	32.97	534%	17	17	33.35	132%	165%
	14	30	7044	75	82.51	468%	22	22	83.23	136%	164%
Rubik's cube model	3	15	5670	67	0.32	2671%	6	6	0.47	244%	424%
	4	20	68040	512	4.98	1216%	100	95	6.45	225%	404%
	5	25	612360	4865	55.24	938%	1216	1137	67.13	219%	368%
	6	30	3674160	31946	398.79	756%	8254	7893	467.65	186%	326%

7 Conclusions

In this paper we have described very tight hashing — a variant of open hashing for representing sets and mappings. Its primary objective is to make optimal use of available memory and it does so in two ways: first, only the entry part of each element is stored, while the index part is discarded without compromising the integrity of the elements stored. Second, overhead costs are kept low by allocating blocks of slots, instead of individual slots, at a time.

It is instructive to compare the estimate of the information-theoretic absolute lower bound, the average set upper bound for very tight hashing (when $h = 50$), and the memory consumption of linear storage:

$$\begin{array}{ll}
\text{Lower bound} \approx & nw' + n \\
\text{Very tight hashing} & 1.07nw' + 0.02n\log_2 n + 6.12n \\
\text{Linear storage} & nw' + n\log_2 n
\end{array}$$

We can see from the formulae that very tight hashing performs significantly better than linear storage for a wide range of n and w'. For instance, very tight hashing consumes only half of the memory consumed by linear storage when $h = 50$, $w' = 5$ and $n \geq 5 \cdot 10^5$. On the other hand, the theoretical limit does not allow any data structure to consume much less than half of the memory consumed by very tight hashing while $h = 50$, $n \leq 10^7$ and $w' \geq 5$.

As the simulations, experimental results and the above calculations show, very tight hashing comes closer to the theoretical limit than linear storage which represents a zero-overhead representation. Moreover, the runtime costs of very tight hash tables in our test implementation are not much inferior to those of the open hash tables used by the original version of our tool.

Acknowledgments. Thanks to Antti Puhakka for comments, and Juha Nieminen for help with the implementation. The work of J. Geldenhuys was funded by the Academy of Finland and by the TISE graduate school.

References

1. B. Boigelot, P. Godefroid, B. Willems & P. Wolper. The power of QDDs. In *Proc. 4th Static Analysis Symposium*, LNCS #1302, pp. 172–186. Springer-Verlag. 1997.
2. T. M. Cormen, C. E. Leiserson, R. L. Rivest & C. Stein. *Introduction to Algorithms, 2nd edition.* The MIT Press. 2001.
3. J. Geldenhuys & P. J. A. de Villiers. Runtime efficient state compaction in SPIN. In *Proc. 5th* SPIN *Workshop*, LNCS #1680, pp. 12–21. Springer-Verlag. 1999.
4. P. Godefroid, G. J. Holzmann & D. Pirottin. State space caching revisited. In *CAV'92: Proc. of the 4th Intl. Conf. on Computer-Aided Verification*, LNCS #663, pp. 175–186. Springer-Verlag. 1992.
5. J. C. Grégoire. State space compression with GETSs. In *Proc. 2nd* SPIN *Workshop*, Held Aug 1996, DIMACS Series No. 32, pp. 90–108. AMS. 1997.
6. G. J. Holzmann. On limits and possibilities of automated protocol analysis. In *PSTV'87: Proc. 7th Intl. Conf. on Protocol Specification, Testing, and Verification*, pp. 339–344. Elsevier. 1987.
7. G. J. Holzmann, P. Godefroid & D. Pirottin. Coverage preserving reduction strategies for reachability analysis. In *PSTV'92: Proc. 12th Intl. Conf. on Protocol Specification, Testing, and Verification*, pp. 349–363. Elsevier. 1992.
8. G. J. Holzmann. State compression in SPIN: recursive indexing and compression training runs. In *Proc. 3rd* SPIN *Workshop.* 1997.
9. G. J. Holzmann & A. Puri. A minimized automaton representation of reachable states. *Software Tools for Technology Transfer*, 2(3), pp. 270–278. Springer-Verlag. 1999.
10. D. E. Knuth. *The Art of Computer Programming, Vol. 3: Searching and Sorting.* Addison-Wesley. 1973.
11. A. Papoulis. Poisson Process and Shot Noise. Ch. 16 in *Probability, Random Variables, and Stochastic Processes*, 2nd ed., pp. 554–576. McGraw-Hill. 1984.
12. B. Parreaux. Difference compression in SPIN. In *Proc. 4th* SPIN *Workshop.* 1998.
13. P. E. Pfeiffer & D. A. Schum. *Introduction to Applied Probability.* Academic Press. 1973.
14. U. Stern & D. L. Dill. Improved probabilistic verification by hash compaction. In *CHARME'95: Advanced Research Working Conf. Correct Hardware Design and Verification Methods*, LNCS #987, pp. 206–224. Springer-Verlag. 1995.
15. http://www.cs.tut.fi/ohj/VARG/TVT/
16. W. C. Visser. Memory efficient storage in SPIN. In *Proc. 2nd* SPIN *Workshop*, Held Aug 1996, DIMACS Series No. 32, pp. 21–35. AMS. 1997.
17. P. Wolper & D. Leroy. Reliable hashing without collision detection. In *CAV'93: Proc. of the 5th Intl. Conf. on Computer-Aided Verification*, LNCS #697, pp. 59–70. Springer-Verlag. 1993.