Fast, All-Purpose State Storage

Peter C. Dillinger and Panagiotis (Pete) Manolios

College of Computer and Information Science, Northeastern University
360 Huntington Ave., Boston MA 02115, USA
{pcd,pete}@ccs.neu.edu

Abstract. Existing techniques for approximate storage of visited states in a model checker are too special-purpose and too DRAM-intensive. Bitstate hashing, based on Bloom filters, is good for exploring most of very large state spaces, and hash compaction is good for high-assurance verification of more tractable problems. We describe a scheme that is good at both, because it adapts at run time to the number of states visited. It does this within a fixed memory space and with remarkable speed and accuracy. In many cases, it is faster than existing techniques, because it only ever requires one random access to main memory per operation; existing techniques require several to have good accuracy. Adapting to accommodate more states happens in place using streaming access to memory; traditional rehashing would require extra space, random memory accesses, and hash computation. The structure can also incorporate search stack matching for partial-order reductions, saving the need for extra resources dedicated to an additional structure. Our scheme is well-suited for a future in which random accesses to memory are more of a limiting factor than the size of memory.

1 Introduction

An efficient explicit-state model checker such as SPIN can easily fill main memory with visited states in minutes if storing them exactly [1]. This is a hindrance to automatically proving properties of large asynchronous programs by explicit state enumeration. In most cases, a level of uncertainty in the completeness of the verification is acceptable, either because of other uncertainties in the process or because one is simply looking for errors. Over-approximating the set of visited states can allow orders of magnitude more states to be explored quickly using the same amount of memory.

Bitstate hashing [2], which uses *Bloom filters* [1, 3, 4], is the pinnacle of exploring as many states as possible when available memory per state is very small—say, less than 8 bits per state. The configuration that tends to cover the largest proportion of the state space in those conditions (setting three bits per state) covers even more when there is more memory per state, but it does not utilize the extra memory well. Using different configurations of the same or a different structure makes better use of the extra memory and comes much closer to full coverage of the state space—or achieves it. At around 36 bits per state,

hash compaction has a good probability of full coverage [5], while the standard bitstate configuration omits states even with 300 bits per state.

The difficulty in making better use of more memory is that special knowledge is needed for known schemes to offer a likely advantage. In particular, one needs to know approximately how many states will be visited in order to tune the data structures to be at their best. In previous work, we described how to use a first run with the standard bistate approach to inform how best to configure subsequent runs on the same or a related model [4]. This can help one to achieve a desired level of certainty more quickly. We also implemented and described automatic tool support for this methodology [6]. The shortcoming of this methodology is that the initial bitstate run can omit many more states than theoretically necessary, if there are tens of bits of memory per state.

Ideally, no guidance would be required for a model checker to come close to the best possible accuracy for available memory in all cases. Such a structure would be good at both demonstrating absence of errors in smaller state spaces and achieving high coverage of larger state spaces. If it were competitively fast, such a structure would be close to the best choice conceivable when the state space size is unknown.

This paper describes a scheme that is closer to this ideal than any known. The underlying structure is a compact hash table by John G. Cleary [7], which we make even more compact by eliminating redundancy in metadata. Our most important contribution, however, is a fast, in-place algorithm for increasing the number of cells by reducing the size of each cell. The same structure can similarly be converted in place to a Bloom filter similar to the standard bitstate configuration. These algorithms allow the structure to adapt to the number of states encountered at run time, well-utilizing the fixed memory in every case.

Do not mistake this scheme for an application of well-known, classical hash table technology. First of all, the Cleary structure represents sets using near minimum space [8], unlike any classical structure, such as that used by the original "hashcompact" scheme [9]. The essence of this compactness is that part of the data of each element is encoded in its position within the structure, and it does this without any pointers. Second, our algorithm for increasing the number of cells is superior to classical rehashing algorithms. Our adaptation algorithm requires no hash function computation, no random memory accesses, and $O(1)$ auxiliary memory.

Our scheme is also competitively fast, especially when multiple processor cores are contending for main memory. Because it relies on bidirectional linear probing, only one random access to main memory is needed per operation. Main memory size is decreasingly a limiting factor in verification, and our scheme is usually faster than the standard bitstate approach until memory is scarce enough to shrink cells for the first time. Even then, each adaptation operation to shrink the cells only adds less than two percent to the accumulated running time.

Execution time can also be trimmed by integrating into the structure the matching of states on the search stack or search queue, used for partial-order reductions [10, 11]. This entails dedicating a bit of each cell to indicating whether

that element is on the stack. In many cases, such integration reduces the memory required for such matching and improves its accuracy as well.

In Section 2 we overview Cleary's compact hash tables and describe a noticeable improvement. Section 3 describes our fast adaptation algorithms. Section 4 describes how to incorporate search stack/queue matching to support partial-order reduction. Section 5 tests the performance of our scheme. In Section 6, we conclude and describe avenues for future work.

2 Cleary Tables

John G. Cleary describes an exact representation for sets in which only part of the descriptor of an element and a constant amount of metadata needs to be stored in each cell [7]. The rest of the descriptor of each element is given by its preferred location in the structure; the metadata link elements in their actual location with their preferred location. We describe one version of Cleary's structure and describe how redundancy in the metadata enables one of three metadata bits to be eliminated.

A Cleary table is a single array of *cells*. For now, assume that the number of cells is 2^a, where a is the number of *address bits*. If each element to be added is b bits long, then the first a are used to determine the *home address* (preferred location) and the remaining $b-a$ are the *entry* value stored in the cell. With three metadata bits, each cell is therefore $b-a+3$ bits.

At this point, we notice some limitations to the Cleary structure. First, each element added has to be of the same fixed size, b bits. It might also be clear that operations could easily degrade to linear search unless the elements are uniformly distributed. This can be rectified by using a randomization function (1:1 hash), but in this paper, we will use the Cleary table exclusively to store hashes, which are already uniformly distributed. We are using an exact structure to implement an inexact one, a fine and well-understood approach [8, 12].

2.1 Representation

We can't expect each element added to be stored in the cell at its home address; this is where bi-directional linear probing and the metadata come in. Entries with the same home address will be placed in immediate succession in the array, forming *chains*. The change metadata bit marks that the entry in that cell is the beginning of a chain. The mapped bit at an address marks that there is a chain somewhere in the table with entries with that home address. The occupied bit simply indicates whether an entry is stored in that cell.

The nth change bit that is set to 1 begins the chain of entries whose home address is where the nth mapped bit that is set to 1 is located. To ensure that every occupied entry belongs to a chain with a home address, a Cleary table maintains the following invariant:

Invariant 1 *In a Cleary table, the number of* mapped *bits set is the same as the number of* change *bits set. Furthermore, the first* occupied *cell (if there is one) has its* change *bit set.*

The chains do not *have* to be near their homes for the representation to work, but the order of the chains corresponds to the order of the set mapped bits. Conceptually, the occupied and change bits relate to what is in the cell and the mapped bit relates to the home address whose preferred location is that cell. One could implement a variant in which the number of cells and home locations is different, but we shall keep them the same.

2.2 Random Access

For the structure to be fast, chains should be near their preferred location, but it is not always possible for each chain to overlap with its preferred location. Nevertheless, this next invariant makes fast access the likely case when a portion of cells are left unoccupied:

Invariant 2 *In a Cleary table, all cells from where an element is stored through its preferred location (based on its home address) must be occupied.*

This basically says that chains of elements must not be interrupted by empty cells, and that there must not be any empty cells between a chain and its preferred location.

Consequently, when we go to add an element, if its preferred location is free/unoccupied, we store it in that cell and set its change bit and mapped bit. We know by Invariant 2 that if the preferred location of an element is unoccupied, then no elements with that home address have been added. Consequently, the mapped bit is not set and there is no chain associated with that home address. (See the first two additions in Figure 1.)

If we're trying to add an element whose preferred location is already occupied, we must find the chain for that home address–or where it must go–in order to complete the operation. Recall that the element stored in the corresponding preferred location may or may not be in the chain for the corresponding home address.

To match up chains with home addresses—to match up change bits with mapped bits—we need a "synchronization point." Without Invariant 2 the only synchronization points were the beginning and end of the array of cells. With Invariant 2, however, unoccupied cells are synchronization points. Thus, to find the nearest synchronization point, we perform a bidirectional search for an unoccupied cell from the preferred location.

From an unoccupied, unmapped cell, we can backtrack, matching up set mapped bits with set change bits, until we reach the chain corresponding to the home address we are interested in—or the point where the new chain must be inserted to maintain the proper matching. If we are adding, we have already found a nearby empty cell and simply shift all entries (and their occupied and

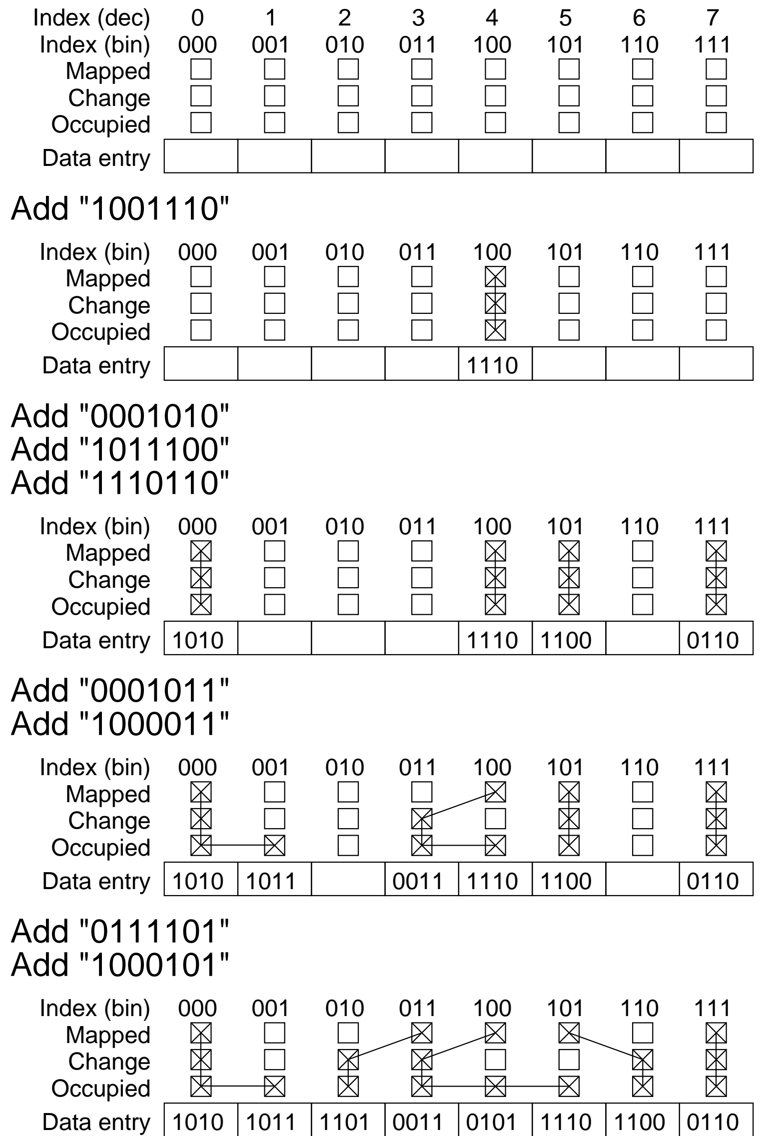

Fig. 1. This diagram depicts adding eight elements to a Cleary table with eight cells. In this example, the elements are seven bits long, the home addresses are three bits long, and the cell data entries are, therefore, four bits long. Each cell is shown with three metadata bits (mapped, change, and occupied). The lines connecting various metadata bits depict how the metadata bits put the entries into chains associated with home addresses.

change bits) toward and into the empty cell, opening up a space in the correct chain—or where the new chain must be added. The remaining details of adding are a trivial matter of keeping the change and mapped bits straight. See Figure 1 for examples, but Cleary's paper can be consulted for full algorithms [7].

Because each add or query operation requires finding an empty cell and the structure uses linear probing, the average search length can grow very large if the structure is allowed to grow too full. Others have analyzed this problem in detail [13], but in practice, more than 90% occupancy is likely to be too slow. We default to allowing no more than 85% occupancy. Note, however, that the searching involves linear access to memory, which is typically much less costly than random access. For large structures, random accesses are likely to miss processor cache, active DRAM pages, and even TLB cache.

2.3 Eliminating occupied bits

The last invariant was originally intended to speed up lookups, but we show how it can be used to eliminate the occupied bits:

Invariant 3 *In a Cleary table, all entries in a chain are put in low-to-high unsigned numerical order.*

This makes adding slightly more complicated, but probably does not affect much the time per visited list operation, in which all negative queries become adds.

With Invariant 3, the occupied bits are redundant because if we fill all unoccupied entries with all zeros and make sure their change bit is unset, then entries with all zeros are occupied iff their change bit is set. This is because entries with all zeros will always be first in their chain, so they will always have their change bit set.

Eliminating this bit allows another entry bit to be added in the same amount of memory, which will cut expected omissions in half. This optimization does seem to preclude the encoding of multisets by repeating entries, because only one zero entry per chain is allowed under the optimized encoding. It seems this is not a problem for visited lists, but will complicate the combination of adaptation and stack matching.

2.4 Analysis

As briefly mentioned, we will use this structure to store exactly inexact hashes of the visited states. This makes the probabilistic behavior relatively easy to analyze. If the hashes are b bits long with a bits used for the home address, the structure has 2^a cells of $b - a + 2$ bits each. Let n be the number of cells occupied, which is also the number of states recognized as new. The probability of the next new state appearing to have already been visited is $f = n/2^b$. The expected number of new states falsely considered visited until the next one correctly recognized as new is $f/(1 - f)$. (For example, when $f = 0.5$, we expect

$0.5/(1-0.5) = 1$ new state falsely considered visited per new state correctly recognized as new.) Thus, the *expected hash omissions* for a Cleary table after recognizing n states as new is

$$\hat{o}_{\mathsf{CT}}(n,b) = \sum_{i=0}^{n-1} \frac{n/2^b}{1-n/2^b} \approx n\frac{2^b}{n}\int_0^{n/2^b} \frac{f}{1-f}df = -n - 2^b \ln\left(1-\frac{n}{2^b}\right) \quad (1)$$

Note that floating-point arithmetic, which is not good at representing numbers very close to 1, is likely to give inaccurate results for the last formula. Here are simpler bounds, which are also approximations when $n \ll 2^b$:

$$\frac{n(n-1)}{2^{b+1}} = \frac{n-1}{2}\cdot\frac{n/2^b}{1} \;\le\; \hat{o}_{\mathsf{CT}}(n,b) \;\le\; \frac{n-1}{2}\cdot\frac{n/2^b}{1-n/2^b} = \frac{n(n-1)}{2^{b+1}-2n} \quad (2)$$

For example, consider storing states as $b = 58$-bit hashes in 2^{28} cells. There are then $a = 28$ address bits and $58-28+2 = 32$ bits per cell. That is correct: a Cleary table can store any 2^{28} 58-bit values using only 32 bits for each one. If we visit $n = 2 \times 10^8$ states, we expect 0.06939 hash omissions (all approximations agree, given precise enough arithmetic). Consequently, the probability of any omissions is less than 7%. The structure is approximately 75% full, which means the structure is using roughly 43 bits per visited state.

Non-powers of 2 The problem of creating a Cleary table of hash values with a number of cells (and home addresses) that is not a power of two can be reduced to the problem of generating hash values over a non-power-of-two range. Take how many cells (home addresses) are desired, multiply by 2^{b-a} (the number of possible entry values), and that should be the number of possible hash values. In that case, stripping off the address bits—the highest order bits of the hash value—results in addresses in the proper range.

3 Fast Adaptation

Here we describe how to adapt a Cleary table of hash values to accommodate more values in the same space, by forgetting parts of the stored hash values. The basis for the algorithms is a useful "closer-first" traversal of the table entries. In this paper, we use this in doubling the number of cells, by cutting the size of each in half, and in converting the table into certain Bloom filters. Both of these can be done in place making only local modifications throughout the structure (no random accesses).

3.1 Twice as many, Half the size

Consider the difference between a Cleary table with cells of size $2^j = b - a + 2$ bits and one with cells of half that size, $2^{j-1} = b' - a' + 2$ bits. If they are the same size overall, then the second has twice as many addresses—one more address bit: $a' = a + 1$. If the elements added to these are prefixes of the same

set of hash values, they will have similar structure. Each home address in the first corresponds to two home addresses in the second, so each mapped home address in the first will have one or both of the corresponding home addresses in the second mapped. In fact, the left-most (highest) bit of an entry in the first determines which of the two corresponding home addresses it has in the second. Thus, the entries in the second have one less bit on the left, which is now part of the home address, and $2^{j-1} - 1$ fewer on the right. Only the $2^{j-1} - 1$ bit on the right are truly missing in the second structure; that is half of the $2^j - 2$ bits per entry in the first structure.

For example, if $j = 5$ and $a = 20$, the first structure has $2^5 = 32$ bits per cell, $32 - 2 = 30$ bits per stored entry, and each element is $30 + 20 = 50$ bits. The second structure has 16 bits per cell, $16 - 2 = 14$ per stored entry, $20 + 1 = 21$ address bits, and $14+21 = 35$ bits per element. The second effectively represents $30/2 = 15$ fewer bits per element. Both structures are 32×2^{20} bits or 4 megabytes overall.

Converting from the first to the second requires careful attention to two things: (1) making sure the new mapped and change bits are updated properly, as each one set in the old structure could entail setting one or two in the new, and (2) making sure to shift elements toward their preferred location, displacing any empty cells in between. These correspond to preserving Invariant 1 and Invariant 2 respectively. Preserving Invariant 3 is just a matter of keeping elements in the same order. In fact, if a chain becomes two in the new structure, Invariant 3 guarantees that all elements in the new first chain are already before elements of the new second chain, because elements with the highest-order bit 0 come before those with highest-order bit 1!

A naive approach that converts the data entries in order from "left" to "right" or "right" to "left" fails. As we iterate through the chains, we need to update the new mapped bits according to the presence of new chains, and those mapped bits might be off to either side of the chains. The problem is that half of the new mapped bits are where old data entries are/were, and we cannot update mapped bits on the side where we have not processed. We could queue up those new mapped bits and update them when we get there, but the queue size is technically only bounded by the length of the whole structure.

A preferable solution should only process data entries whose new mapped bit lies either in the cell being processed or in cells that have already been processed. We can solve this problem with a traversal that processes entries in an order in which all entries between an entry and its preferred location are processed before that entry is processed.

3.2 Closer-first traversal

It turns out that virtually any in-place, fast adaptation we want to do on a Cleary table can be done by elaborating a traversal that processes the entries between any given entry and its preferred location before processing that entry. This traversal can be done in linear time and constant space. All accesses to the

table are either linear/streaming or expected to be cached from a recent linear access.

The following theorem forms the basis for our traversal algorithm:

Theorem 1 *In a Cleary table, the maximal sequences of adjacent, occupied cells can be divided uniquely into subsequences all with the following structure:*

- ***Zero or more*** *"right-leaning" entries each with its preferred location higher than its actual location, followed by*
- ***One*** *"pivot" entry at its preferred location, followed by*
- ***Zero or more*** *"left-leaning" entries each with its preferred location lower than its actual location.*

Proof Idea Proof is by induction on the number of pivots in a maximal sequence, using these lemmas:

- An entry in the first cell of the structure is not left-leaning, and an entry in the last cell is not right-leaning. (This would violate Invariant 1 or how `change` bits are implicitly connected to `mapped` bits.)
- An entry adjacent to an unoccupied cell cannot be leaning in the direction of the unoccupied cell. (This would violate Invariant 2.)
- A left-leaning entry cannot immediately follow a right-leaning entry. (Such entries would be in different chains and violate how `change` bits are implicitly connected to `mapped` bits.) □

This theorem solves the bootstrapping problem of where to start. We start from each pivot and work our way outward. To minimize the traversing necessary to process entries in an acceptable order, we adopt this order within each subsequence: process the pivot, process the right-leaning entries in reverse (right-to-left) order, and then process the left-leaning entries in (left-to-right) order.

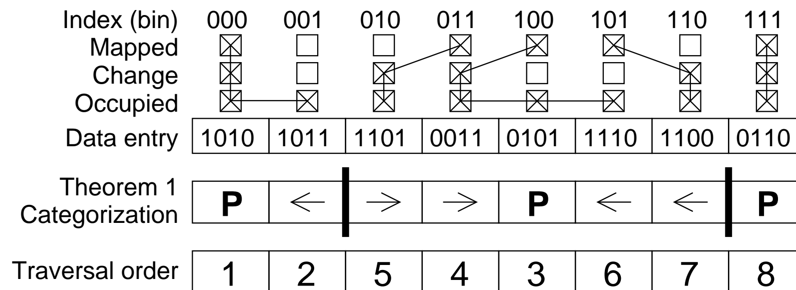

Fig. 2. This diagram takes the final Cleary table from Figure 1, shows how the entries are categorized according to Theorem 1, and lists the order of processing by the closer-first traversal.

A natural implementation of the overall traversal, which finds subsequences "on the fly," goes like this: (1) Remember starting location. (2) Scan to a pivot and remember its location. (3) Process the pivot and then all entries back through the starting location (reverse order). (4) Process entries after the pivot until one that is not left-leaning and go back to (1).

To determine the direction of each entry's home, home addresses for the current and saved locations are tracked and updated as they are updated. Proper tracking eliminates the need for searching for synchronization points. All accesses are adjacent to the previous except between steps (3) and (4). Since the pivot was processed recently, it will still be cached with high probability.

A corollary of Theorem 1 is that processing entries in this order guarantees that processing the (old or new) home location of an entry will never come after the processing of that entry.

3.3 Building on the traversal

The closer-first traversal allows us to compact cells to half their size and update mapped and change bits accordingly without overwriting unprocessed cells. (That handles Invariant 1.)

Storing entries near their preferred location (Invariant 2) can also be handled in the same elaborated traversal. We store the pivot entry in the new cell that is its preferred location. For processing the right-leaning entries, we keep track of the "next" new location, which is the one just before (left of) the new cell we most recently put an entry into. Each right-leaning entry will be placed at the minimum (left-most) of that "next" location and the entry's preferred location. We do the symmetric thing for left-leaning entries. This procedure guarantees Invariant 2 because it places each entry either at its preferred location, or adjacent to an entry (which is adjacent to an entry . . .) which is at that preferred location.

3.4 Our Design

Cell sizes should be powers of two to allow for repeated cutting in half. Our current design starts with 64 bits per cell. If there are, say, 2^{28} cells (2 GB), states are stored as $62 + 28 = 90$ bit hash values. The probability of any omissions is theoretically one in billions. Jenkins' hash functions are fast and behave according to expectation [14]. Thus, the only tangible benefit of allowing larger values is psychological, and it might require more hash computation.

Recall that the structure becomes unacceptably slow beyond 90% occupancy. Thus, when an occupancy threshold is reached (default 85%), we convert from 64 to 32, from 32 to 16, and from 16 to 8. We do not go from 8 to 4. Consider what a Cleary table with 4 bits per cell would be like. Two bits are metadata and two bits are left for the entry. Each cell contains only one of four possible entries. But each cell is four bits long. This means we could represent sets of values with the same number of bits using the same amount of memory just by using them as bit indexes into a bit vector, and that would allow us to add any

number of such values. That would be the same as a $k = 1$ Bloom filter. You could also think of it as a Cleary table with just mapped bits; entries are 0 bits, so no need for change bits. In other words, a Cleary table with 4 bits per cell is no more accurate than a $k = 1$ Bloom filter, cannot accommodate as many elements, and might be noticeably slower.

Thus, we choose to convert from an 8-bit-per-cell Cleary table into a Bloom filter. We actually convert into a special $k = 2$ Bloom filter, but let us first examine how to convert an 8-bit-per-cell Cleary table into a single-bit ($k = 1$) Bloom filter.

3.5 Adapting to Bloom filter

Adapting a Cleary table using 8 bits per cell into a single-bit ($k = 1$) Bloom filter is incredibly easy using the traversal. To turn an old entry into a Bloom filter index, we concatenate the byte address bits with the highest three data bits, from the six stored with each old entry. This means that setting the Bloom filter bit for each old entry will set one of the eight bits at that entry's preferred location. In other words, only bytes that had their mapped bits set will have bits set in the resulting Bloom filter. Using the same "closer-first" traversal guarantees that entries are not overwritten before being processed.

Unfortunately, single-bit Bloom filters omit states so rapidly that they often starve the search before they have a high proportion of their bits set. Holzmann finds that setting three bits per state ($k = 3$) is likely to strike the right balance between not saturating the Bloom filter and not prematurely starving the search. Unfortunately, we cannot convert a Cleary table using 8 bits per cell into any structure we want. First of all, it needs to have locality in the conversion process, so that we can do the conversion in place. Second, it can only use as many hash value bits as are available in the 8-bit-per-cell Cleary table.

We believe the best choice is a special $k = 2$ Bloom filter that has locality. It is well-known that forcing Bloom filter indices to be close to one another significantly harms accuracy, but we do not have much choice. The first index uses three of the six old entry bits to determine which bit to set. That leaves only three more bits to determine the second index, which can certainly depend on the first index. Running some simulations has indicated that it does not really matter how those three bits are used to determine the second index from the first; all that matters is that all three are used and that the second index is always different from the first.

We have decided that the easiest scheme to implement uses those three bits as a bit index into the next byte. Thus, the same address bits that determined the home addresses for the 8-bit Cleary table determine the byte for the first Bloom filter index. The first three entry bits determine the index within that byte, and the next three determine the index within the next byte.

Altering the conversion/adaptation algorithm to generate this structure is not too tricky. Recognizing that we cannot order our traversal to avoid the second indices overwriting unprocessed entries, we must keep track of the new byte value that should follow the right-most processed entry and not write it until the entry

has been processed. That is the basic idea, anyway. Our implementation caches up to three bytes yet to be written: one left over from the previous subsequence, one to come after the pivot, and one more to work with while iterating down the "leaning" sides.

We can analyze the expected accuracy of this scheme by building on results from previous work [4]. This is a $k = 2$ fingerprinting Bloom filter whose fingerprint size is 3 bits more than one index ($\log_2 s = 3 + \log_2 m$; $s = 8m$; s is the number of possible fingerprints and m is the number of bits of memory for the Bloom filter). Previous work tells us that a simple over-approximation of the expected hash omissions from a fingerprinting Bloom filter is the sum of the expected hash omissions due to fingerprinting, which we can compute using Equation 1 or 2 ($b = 3+\log_2 m$), and the expected hash omissions due to the underlying Bloom filter, which is roughly $\sum_{i=0}^{n-1}(1 - e^{-2i/m})^2 \approx n(1 - e^{-2n/m})^2/2$. Thus, our rough estimate is

$$\hat{o}_{\mathsf{BF}} \approx \frac{n(n-1)}{2(8m-n)} + \frac{n}{2}\left(1 - e^{-2n/m}\right)^2 \tag{3}$$

That formula and Equations 1 and 2 give the expected omissions assuming we had been using the given configuration since the beginning of the search. That is easily corrected by subtracting the expected hash omissions for getting to the starting conditions of each new configuration—had that configuration been used from the beginning. For each Cleary table configuration, we subtract the expected hash omissions for the starting number of occupied cells from the expected hash omissions for the ending number of occupied cells. Note that collapses can make the next starting number smaller than the previous ending number. We cannot easily be so exact with the Bloom filter, so we can just use the previous ending number as the starting number.

4 Search Stack Matching

Partial-order reductions play a central role in making contemporary explicit-state verifiers effective [15], by reducing the size of the state space that needs to be searched for errors. In many cases, the reduction is dramatic, such as making previously intractable problems tractable.

A typical implementation requires runtime support in the form of a "cycle proviso," which needs to know whether a state is on the DFS stack [10] or BFS queue [11] (depending on whether depth-first or breadth-first search is being used). We will refer to this as checking whether the state is *active*.

Combining with visited list A visited list based on cells, such as the Cleary table, can include a bit with each cell that indicates whether the state stored in that cell is active. This can be a compact option since no other random-access structure with this data is required. However, the relative size of the stack or queue can be small enough that the vast majority of active bits would be zero and a separate structure would be more compact.

Speed should favor the unified structure, because separate lookup is not required to check whether a state is active. Marking the state as active can also "piggy-back" on the initial look-up by saving the cell location where the new entry was just placed.

Accuracy is also a strong point of the unified structure. Specifically, stack/queue matching is 100% accurate whenever the visited list is accurate. Using a separate structure that is independently inaccurate could lead to imprecise reductions and error omission even if the visited list causes no omissions.

Complications Despite the fact that multiple states can map to the same value at the same address in a Cleary table (or other "hash compaction" scheme), there is not traditionally a need to account for multiple stack entries per stored value, because only one such state would ever be recognized as new and, therefore, only one can ever be active.

But our *adaptive* Cleary tables can have more than one state on the stack that each map to the same table entry. When shrinking cells, some pairs of entries will no longer be distinguishable and are collapsed into one. (One of each such pair would have been a hash omission if the new cell size had been used from the beginning!) If both states are on the stack, however, we prefer to be able to say that there is more than one state on the stack matching a certain entry. Cleary's table representation allows duplicate entries, so in these rare cases, perhaps we could duplicate the entry for each matching state on the stack. However, our optimization that allowed only two metadata bits per state assumed that an entry of all zeros would be at the beginning of a chain, and if we allow more than one entry of all zeros in a chain, this is no longer the case. However, the only case in which we want to have duplicate entries is when each of those entries needs to have its active bit set. As long as the active bit is set, therefore, it would be possible to distinguish the entry from an unoccupied cell. When the state gets removed from the stack/queue, however, we would need to do something other than clearing the active bit and turning the duplicate all-zeros entry into an unoccupied cell (which could violate Invariant 2). Since deletion from a Cleary table is possible, we just delete the duplicate entry when its active bit is cleared. Our implementation deletes all such duplicates, not just all-zero entries, to (1) make room for more additions and (2) maintain the invariant that no state that is active has a matching entry in the Cleary table without its active bit set.

A final complication comes when the structure becomes a Bloom filter, which is not based on cells. For a single-bit Bloom filter, we could have an active bit for each bit, but that would be a waste of space considering what a small proportion of visited states are typically on the stack. There is also the problem that a Cleary table with 8 bits per cell and an active bit in each cell only has five entry data bits per cell. Ideally, we want a two-bit Bloom filter that uses all five bits and takes up more space than the accompanying active information. Here's a design: use two bits per byte as a counter of the number of active states whose home is/was this byte. As in a counting Bloom filter [16], the counter can overflow and underflow, introducing the possibility of false negatives—in addition to the false positives due to sharing of counters. In the context of a search that is already

rather lossy, these are not big issues. Six bits of each byte remain for the Bloom filter of visited states. If we set two bits per state, one in the home and one in the following byte, that is 36 possibilities for states whose home is this byte. The 5 bits of data left allow us to cover 32 of those 36 possibilities. Mapping those 5-bit values to pairs of indices 0..5 is efficient with a small look-up table. This makes it easy to spread indices somewhat evenly over that range, but all six bit indices cannot have the exact same weight/likelihood.

5 Validation

We have implemented the schemes described in a modified version of Spin 5.1.7 and use that for all of the experimental results. It can be downloaded from [17]. Our implementation also outputs the expected hash omission numbers based on formulas given. Timing results were taken on a 64-bit Mac Pro with two 2.8 GHz quad-core Intel Xeon processors and 8GB of main memory.

5.1 Accuracy

In this section we demonstrate the accuracy advantages of our adaptive Cleary+ Bloom structure as compared to the standard $k = 3$ bitstate approach and validate the predictive value of our formulas.

Setup The main accuracy test of Figure 3 has an artificial aspect to it, and we explain that here. A typical protocol is subject to what we call the *transitive omission problem* [4]. States omitted from a lossy search can be put in two categories: *hash omissions*, those states that were falsely considered not new by the visited list, and *transitive omissions*, those states that were never reached because other omissions made them unreachable. Clearly, if there are zero hash omissions, there are zero transitive omissions. But when there are hash omissions, we do not reliably know how many transitive omissions resulted. In well-behaved protocols, there tends to be a linear relationship, such as two transitive omissions per hash omission.

Despite the transitive omission problem—or perhaps because of it—minimizing expected hash omissions is key to maximizing the accuracy of a search. This approach also optimizes two other metrics: the probability of any omissions and expected coverage. Note that when much smaller than 1, the expected hash omissions approximate the probability of any omissions. However, the probability of any omissions is not good at comparing searches that are expected to be lossy, and coverage is hard to predict in absolute terms in the presence of transitive omissions. Thus, we focus on minimizing hash omissions.

To measure hash omissions and compare those results against our mathematical predictions, we generated a synthetic example that is highly connected and, therefore, should have almost no transitive omissions. The model consists of a number starting at zero, to which we non-deterministically add 1 through 10 until a maximum is reached. Choosing that maximum allows us to manipulate the size of the state space, as we have done to get the results in Figure 3.

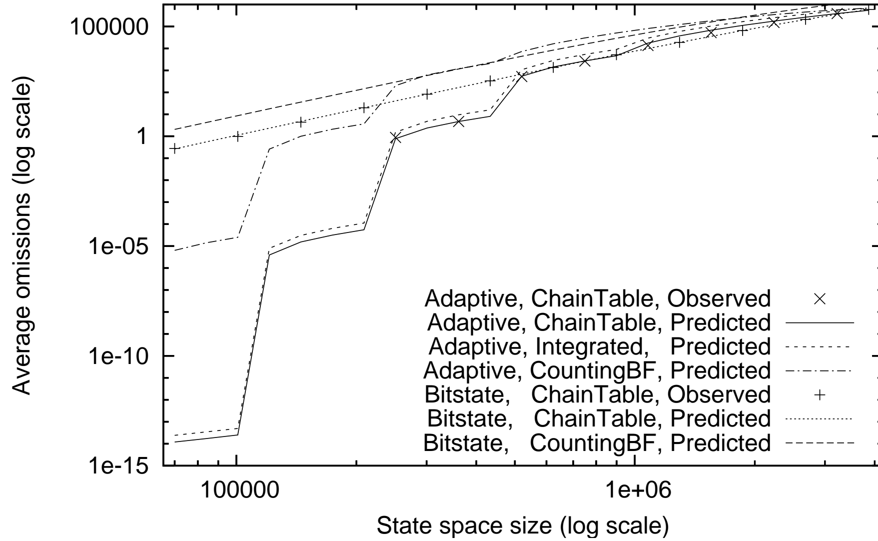

Fig. 3. This graph compares the predicted and observed accuracy of searches using our adaptive scheme and the standard $k = 3$ bitstate scheme, using different DFS stack-matching schemes. The model is described in Section 5.1; it exhibits virtually no transitive omissions and allows the state space size to be manipulated. About 1MB total is available for visited and active state matching (see Section 5.1). Observation data points represent the average of 10-50 trials. Not enough trials were performed to observe any omissions from searches expecting full coverage with high probability. To avoid clutter, observations for many configurations are omitted but are analogously close to prediction. Predicted values are based on equations given in this paper.

Adaptive vs. Bitstate A quick examination of Figure 3 confirms that when memory is not highly constrained (left side), our visited list scheme ("Adaptive") is more accurate than (below) the standard bitstate approach ("Bitstate"). (For now, only consider the results using a chaining hash table for stack matching, "ChainTable.") For example, at around 200 000 states, our scheme has less than a 1 in 10 000 chance of any omissions while the bitstate scheme expects to have omitted about 10 states. When memory is highly constrained (right side), the two yield similar accuracy.

If we look at the "Adaptive, ChainTable" accuracies in more detail, we can see where the adaptations occur. When the expected omissions are near 10^{-14}, it is still using 64 bits per cell. When the expected omissions jump to 10^{-5}, it has changed over to 32 bits per cell. Near one omission, it is using 16 bits per cell. When it moves to 8 bits per cell, its accuracy is similar to the standard $k = 3$ bitstate approach. The structure converts to the special $k = 2$ Bloom filter around 10^6 states. The mathematical prediction overestimates a little at first because of the roughness of the approximation for fingerprinting Bloom filters, but it gets closer later.

No observations show up for the 64 and 32 bits per cell cases because no omissions were encountered in the tens of trials run for each state space size. At least a million trials would have been required to get good results for 32 bits per cell.

Figure 3 generalizes very easily and simply. These results are for 1 megabyte. To get the results for c megabytes, simply multiply the X- and Y-axis values by c. It is that simple. In other words, the proportion of states that are expected to be hash omissions under these schemes depends only on the ratio between states and memory, not on their magnitude. Unlike many classical structures, these schemes scale perfectly.

Also, the Jenkins hash functions [14] used by Spin are good enough that, for all non-cryptographic purposes, the relationships among reachable state descriptors in a model are irrelevant. The Jenkins hashes are effectively random.

Stack matching The "ChainTable" results of Figure 3 assume that a chaining hash table is used to match active states and that its memory size is negligible compared to the memory size of the visited list (see memory usage in Figure 4). This is the case for problems whose maximum depth is orders of magnitude smaller than the state space size. Because this approach is faster and usually more compact than what is currently available in Spin(version 5.1.7)—but an application of classical techniques—we consider it state-of-the-art for this case. (More information is on the Web [17].)

The "CountingBF" results assume that a counting Bloom filter [16] that occupies half of available memory is used. This is the allocation usually used by Spin's CNTRSTACK method, and we consider it state-of-the-art for unbounded active lists that are dynamically swapped out to disk, as in Spin's SC ("stack cycling") feature. Note that a counting Bloom filter cannot overflow a fixed memory space as a chaining hash table can.

The important difference for Figure 3 is that only half as much memory is available to the visited list for the "CountingBF" results as for the "ChainTable" results—to accommodate the large counting Bloom filter. Thus, the counting Bloom filter approach is clearly detrimental to accuracy if the search stack is small enough to keep in a small area of memory (Figure 3). The ($k = 2$) counting Bloom filter is also relatively slow, because it always requires two random lookups into a large memory space; thus it is DRAM-intensive (see Figure 4). But if the search stack is large enough to warrant swapping out, the counting Bloom filter is likely to be a better choice (not graphed, but see memory usage in Figure 5).

Using our adaptive Cleary+Bloom structure allows a third option: integrating active state matching with visited state matching ("Integrated"), which is always preferable to the counting Bloom filter. Making room for the counting Bloom filter as a separate structure requires cutting the bits per entry by about half. Making room for one `active` bit in each cell only takes away one entry bit. The result is a doubling in the expected omissions, which is tiny compared to the impact of cutting cell sizes in half ("Adaptive, Integrated" vs. "Adaptive, CountingBF" in Figure 3).

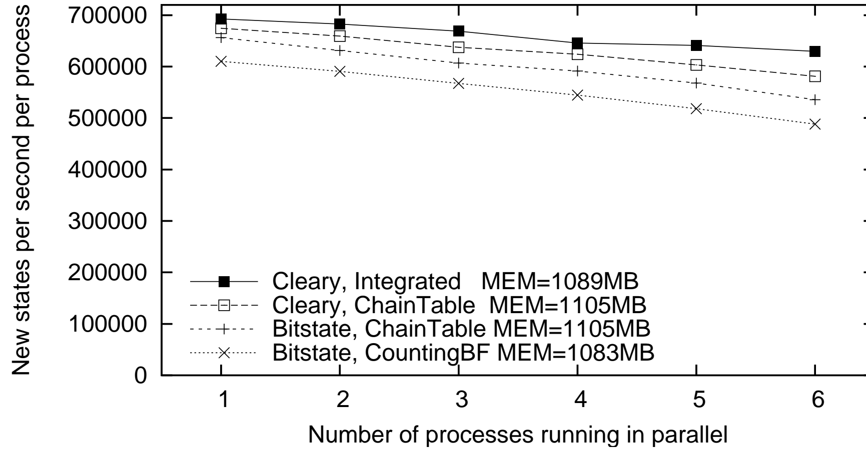

Fig. 4. This plots the verification speed, in average time per state per process, of various visited and stack matching methods for one to six instances running in parallel on a multicore system. In this test, Cleary tables never became full enough to trigger adaptation algorithms. The model is PFTP with LOSS=1,DUPS=1,QSZ=5. State-vector is 168 bytes. Visited state storage is 1024MB, except for CountingBF, which is given 512MB each for the visited set and the counting Bloom filter stack. Depth limit of 1.5 million was beyond the maximum and required a relatively small amount of additional storage. 67.8M states results in about 127 bits per state.

5.2 Speed

Figure 4 confirms that the Cleary table is very fast when the structure is big and never gets beyond about half full. Plugging in a 64-bit Cleary table in place of $k = 3$ bitstate increases speed by about 2.8% when running by itself. Using the Cleary table also for search stack matching increases that to 5.5%, unless using the counting Bloom filter with standard bitstate, which is 13.5% slower than the integrated Cleary structure.

The Cleary table with integrated matching of active states is the least affected by running in parallel with other model checker instances. Running six instances simultaneously slows each by about 9%, but running six instances of Spin's bitstate implementation slows them by about 20%. This can easily be explained by the fact that the Cleary table with integrated stack matching only needs one random access to main memory to check/add a new state and/or check/add it to the search stack. The $k = 3$ bitstate method with the counting Bloom filter stack requires five random accesses.

Figure 5 shows how the user can optimize more for speed or more for accuracy by setting the maximum occupancy before adaptation. The omissions (not graphed) decrease with higher maximum occupancy, most dramatically between 60% and 65% (in this case) because 60% and lower ended with a smaller cell size. The omissions for 60% were about twenty times higher than for 65%, a

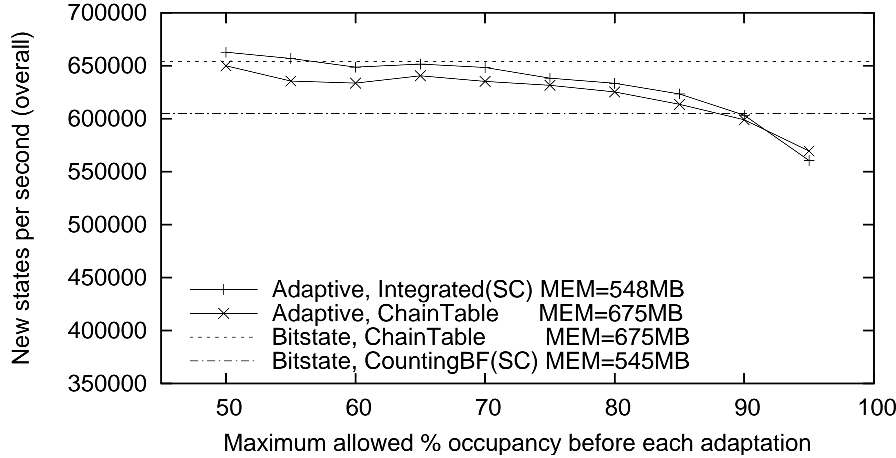

Fig. 5. This plots the verification speed, in average time per state, of our adaptive Cleary tables, allowed to fill to various occupancies before each doubling of the number of cells. The straight lines show $k = 3$ bitstate performance with the specified stack matching scheme. Visited set is given 512MB, except for CountingBF(SC), which splits that equally between visited and active state matching. "Stack cycling" (SC) was used when supported to reduce memory requirements. Otherwise, depth limit of 3 million was needed to avoid truncation. The model is PFTP with LOSS=1,DUPS=1,QSZ=6. State-vector is 192 bytes. 170M states results in about 25 bits per state. All Cleary tables converted from 64 to 32 and 32 to 16 bits per cell. Those limiting to less than 65% occupancy also converted from 16 to 8 and had at least an order of magnitude more omissions. These results are for a single process running alone; results with four processes are slightly steeper and also cross the bottom line at around 90%.

much larger difference than between 50% and 60% or 70% and 90%. Adaption itself does not cause omissions, but after adaption, the significantly higher rate of omission causes the number of omissions to jump, as in Figure 3.

Typically, lower maximum occupancy means faster, but 60% was actually slower than 65% because the 60% doubled the number of cells right before it finished. It should be possible to avoid such scenarios with heuristics that predict how close to completion the process is and allowing higher occupancy if near completion. Nevertheless, even after doubling its number of cells several times, our adaptive storage scheme is faster than standard bitstate in some cases.

Another experiment (not graphed) exhibits how little time is needed for adaptation. We ran a 370 million state instance of PFTP(1,1,7) using 320 megabyte instances of our adaptive structure, causing adaptation all the way down to a $k = 2$ Bloom filter. Adaptation operations required 1.3-2.0% of the running time so far, with the total time spent on adaptation never exceeding 3.3% of the running time so far. This includes maximum allowed occupancies of 75% and 90%, with and without integrated active state matching, and 1 to 4 processes running simultaneously. Despite all the adaptations, "Adaptive, Integrated(SC)" was 7-

18% faster and explored more states than "Bitstate, CountingBF(SC)" on the same problem given the same amount of memory. (Non-power of 2 memory size and significant time spent as a localized $k = 2$ Bloom filter conferred advantages to our adaptive structure not seen in previous results.)

In other experiments not shown, we have noticed that the Cleary structure gains speed relative to competing approaches as memory dedicated to the structure grows very large. We suspect that this relates to higher latency per main memory access because of more TLB cache misses in accessing huge structures. If one compares the approaches using relatively small amounts of memory, findings are likely to be skewed against the Cleary table.

The biggest sensitivity to the particular model used is in how long it takes to compute successors and their hashes, which is closely tied to the state vector size. More computation there will tend to hide any differences in time required by different state storage techniques. Less will tend to inflate differences. There is little way for a different model to result in different speed rankings. The PFTP model used here for timings has a state vector size (about 200 bytes) that is on the low end of what might be encountered in large problems, such as those listed in Tables V and VI of [18].

6 Conclusion and Future Work

We have described a novel scheme for state storage that we believe has a future in explicit-state model checkers such as Spin. It has the flexibility to provide high-assurance verification when memory is not highly constrained and good coverage when memory is highly constrained. In that sense, it is an "all-purpose" structure that requires no tuning to make good use of available memory.

In many cases, our scheme is noticeably faster than Spin's standard bitstate scheme. We believe this is due to its favorable access pattern to main memory: only one random look-up per operation. For example, when multiple processor cores are contending for main memory, our scheme is consistently faster. When supporting unbounded search stack/queue sizes, our scheme is consistently faster. Otherwise, the standard bitstate scheme is only a little faster once the number of states reaches about 1/100th the number of memory bits. At that point, bitstate has already omitted states while our scheme can visit twice that many with no omissions. Cleary's compact hash tables and our fast, in-place adaptation algorithm offer speed, accuracy, compactness, and dynamic flexibility that previous schemes fall well short of in at least one category.

We plan to extend this technique further. It should be possible to start a search storing full states and then adapt quickly in place to storing just hash values. This would provide the psychological benefit of exact storage for as long as possible. It should also be possible to make an intermediate adaptation step in between splitting cells in half. This is much trickier, but would make even better use of available memory. In fact, we hope to demonstrate how the scheme is "never far from optimal."

References

1. Holzmann, G.J.: The Spin Model Checker: Primer and Reference Manual. Addison-Wesley, Boston, Massachusetts (2003)
2. Holzmann, G.J.: An analysis of bitstate hashing. In: Proc. 15th Int. Conf on Protocol Specification, Testing, and Verification, INWG/IFIP, Warsaw, Poland, Chapman & Hall (1995) 301–314
3. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM **13**(7) (July 1970) 422–426
4. Dillinger, P.C., Manolios, P.: Bloom filters in probabilistic verification. In: Formal Methods in Computer-Aided Design (FMCAD) 2004. Volume 3312 of LNCS., Springer-Verlag (2004)
5. Stern, U., Dill, D.L.: A new scheme for memory-efficient probabilistic verification. In: IFIP TC6/WG6.1 Joint Int'l Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification. (1996) 333–348
6. Dillinger, P.C., Manolios, P.: Enhanced probabilistic verification with 3Spin and 3Murphi. In: 12th SPIN Workshop on Model Checking Software. Volume 3639 of LNCS., Springer-Verlag (August 2005)
7. Cleary, J.G.: Compact hash tables using bidirectional linear probing. IEEE Trans. Computers **33**(9) (1984) 828–834
8. Pagh, A., Pagh, R., Rao, S.S.: An optimal bloom filter replacement. In: Proceedings of the 16th ACM-SIAM Symposium on Discrete Algorithms (SODA), SIAM (2005) 823–829
9. Wolper, P., Leroy, D.: Reliable hashing without collision detection. In: 5th International Conference on Computer Aided Verification. (1993) 59–70
10. Holzmann, G.J., Peled, D.: Partial order reduction of the state space. In: First SPIN Workshop, Montrèal, Quebec (1995)
11. Bosnacki, D., Holzmann, G.J.: Improving spin's partial-order reduction for breadth-first search. In: 12th SPIN Workshop on Model Checking Software. Volume 3639 of LNCS., Springer (2005) 91–105
12. Carter, L., Floyd, R., Gill, J., Markowsky, G., Wegman, M.: Exact and approximate membership testers. In: Proceedings of the 10th ACM Symposium on Theory of Computing (STOC), ACM (1978) 59–65
13. Pagh, A., Pagh, R., Ruzic, M.: Linear probing with constant independence. In: Proceedings of the 39th ACM Symposium on Theory of Computing (STOC), New York, NY, USA, ACM (2007) 318–327
14. Jenkins, B.: http://burtleburtle.net/bob/hash/index.html (2007)
15. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)
16. Fan, L., Cao, P., Almeida, J., Broder, A.Z.: Summary cache: a scalable wide-area Web cache sharing protocol. IEEE/ACM Transactions on Networking **8**(3) (2000) 281–293
17. Dillinger, P.C., Manolios, P.: 3Spin home page http://3spin.peterd.org/.
18. Holzmann, G.J., Bosnacki, D.: The design of a multicore extension of the spin model checker. IEEE Trans. Softw. Eng. **33**(10) (2007) 659–674