An Analysis of Bitstate Hashing [1)]

GERARD J. HOLZMANN
Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974, U.S.A.

Abstract. The bitstate hashing, or supertrace, technique was introduced in 1987 as a method to increase the quality of verification by reachability analyses for applications that defeat analysis by traditional means because of their size. Since then, the technique has been included in many research verification tools, and was adopted in tools that are marketed commercially. It is therefore important that we understand well how and why the method works, what its limitations are, and how it compares with alternative methods over a broad range of problem sizes.

The original motivation for the bitstate hashing technique was based on empirical evidence of its effectiveness. In this paper we provide an analytical argument. We compare the technique with two alternatives that have been proposed in the recent literature. We also describe a sequential bitstate hashing technique that can be of value when confronted with very large problem sizes.

Keywords: Verification, Concurrency, State explosion, Model checking.

1. Introduction

The bitstate hashing technique, introduced in [H87] and elaborated in [H88] and [H91], can be shown to perform relatively high coverage verifications within a memory arena that may be orders of magnitude smaller than required for exhaustive verifications. The method has made it possible to apply formal verification techniques to problems that would normally have remained beyond the scope of automated tools, e.g. [C91], [C94], [L94], [H94a], [H94b].

This paper provides an analytical argument that explains the performance of the algorithm. The analysis is then extended to compare the method with alternative techniques, hash-compact, and multihash [WL93], [SD95], [SD96]. We show that a variation of the multihash technique, sequential bitstate hashing, that can outperform the other algorithms when applied to very large problem sizes.

Problem Definition

In the following, let N be the total number of reachable states of the concurrent system, i.e., the number of distinct nodes in the reachability graph that describes its behavior. Let S be the number of *bits* required to store a single state from this set without loss of information (i.e., this could be in compressed form), and, finally, let M be the total number of *bits* in memory that is available for performing the reachability analysis.

Knowns and Unknowns

In typical applications, the maximum value for parameter M is hardware dependent and cannot be modified significantly. The parameters N and S are application dependent and can, to a certain extent, be manipulated by the user. In the remainder of this paper we assume that they each have their lowest meaningful value, and are for all practical purposes constants that, together with parameter M, define the verification problem to be solved.

---

1. An early version of this paper appeared in the Proceedings of the 15th Symposium on Protocol Specification, Testing and Verification, pp. 301-314, published by Chapman & Hall, London, 1995.

It is an inherent part of the problem that the value of parameter N cannot be predicted without performing the reachability analysis itself. For given M, we can also say that the ratio M/N should be considered unknown. In the worst case it may even be smaller than one. Such problems used to be considered beyond the reach of automated verifiers. The premise of the bitstate hashing methods is, however, that when a verification problem is too large to solve completely, even an approximate answer can be of great value to a designer. Our algorithm, therefore, should not presuppose a cutoff ratio for M/N below which it stops working. The quality of a general bitstate hashing method is determined by the range of values for M/N for which it can be used.

> An upper-bound on N is trivially 2^S, but this is of little help since in practice N is always many orders of magnitude smaller. A value for S of 1024 bits, for example, would set the upper-bound at 2^{1024} distinct states.

Note that if a value for N could be predicted accurately, the problem we are considering would be simpler. If $S \leq M/N$ we can perform an exhaustive search and need not apply bitstate hashing.

> The value for M is rarely larger than 10^{10} bits (1 Gbyte of RAM). For non-trivial problems, the value for S is typically in the order of 10^3 bits. This means that for these two parameters we can solve the verification problem exhaustively for values of N up to 10^7 states.

When N is larger, or in general when $S > M/N$, this strategy fails. It is usually not a realistic alternative to built the reachability graph on slower secondary storage (disk), where M can be one to two orders of magnitude larger. The following subsection briefly explains the reasons.

On Using Disk Storage

If $N \cdot S$ bits of disk-space are available, we could attempt to construct the reachability graph on disk. Unavoidably, for every new state that is generated, sooner or later we must check in the existing graph whether that state is already present or must be added. There is no statistically significant relation that can be deduced from the order in which states are generated that could be exploited in a disk caching strategy. As far as disk access is concerned, each new state could be in any randomly chosen portion of the existing graph, and will require at least one true disk read-access to verify the presence or absence of the new state at that location. If each disk-access requires b seconds, it will cost in the order of $N \cdot b$ seconds to construct the reachability graph. Typical values for b are in the order of 10^{-2} sec. This means that exploring 10^7 reachable states with a disk-based graph woul consume at least 24 hours of run-time. To store 10^9 states would take not only a very large disk, but also months of dedicated computation. For all practical purposes, therefore, this approach would not provide a viable alternative.

Maximizing Coverage

If also the use of secondary storage is impossible or impractical, we can attempt to use an approximate strategy. If, for the moment, we still assume that we know a value for N, we can build the complete reachability graph in main memory by exploiting the fact that up to M/N bits can be used per state, of course, with a minimum of 1 bit. We can then maximize the coverage by fine-tuning a compression method for the specific N we know about, to convert the S bits of input into M/N bits of output with minimal loss of information, and optimal coverage.

The Goal of Bitstate Hashing

In the problem we consider here no reliable prior knowledge of the ratio M/N is assumed. It should be carefully noted that in the selected domain of application, where $S > M/N$, no algorithm can *guarantee* complete coverage of the reachability graph. Since we can only

store less information than the minimum that is required to encode a state (S), loss of information is inevitable. The goal of the bitstate hashing technique is to minimize the loss and maximize the coverage, for as large a range of relative values for M and N as possible.

Occam's Razor

For an analysis of the expected effect of the bitstate hashing techniques we have to make assumptions about the typical structure of a reachability graph. Those assumptions are inherently problem dependent (see *Approximation Error* below). This unavoidably limits the amount of precision that is meaningful in an analytical argument. Our intent in Section 2 is, therefore, to include no more detail then is necessary to explain the empirical evidence that is available (cf. Section 4).

2. Analysis

For this analysis it is convenient to stick to powers of two. We will therefore use the symbols n, m, and s, with the following meaning.

$$N = 2^n, \quad M = 2^m, \quad S = 2^s$$

For every state of S bits, a hash value of m bits is computed, which is associated with a unique bit position within a large bit array of size 2^m. If we convert N distinct states into their corresponding hash values, what is the average probability of a hash collision within that set? The procedure we will consider is the following. The initial value of all 2^m bits is zero. For every new hash value generated we inspect the current value of the bit that corresponds to the hash value, and if it is zero we set it to one. If the bit is already set, we count this as a hash collision. We consider two cases separately: $N < M$ and $N \geq M$.

Single-Bit Hashing

When $N < M$, the first state for which we enter a value into the bit array has zero probability of collision, and the last state has a maximum probability of collision of $(N-1)/M$. The average probability over all N states is therefore

$$P_1 \leq \frac{1}{N \cdot M} \sum_{i=0}^{N-1} i = \frac{N-1}{2M} \leq 2^{n-m-1}. \qquad [1]$$

The expected coverage of the search is $(1-P_1)\cdot 100$ %, so we would like P_1 to be as small as possible.

When $N \geq M$, i.e., the number of states is larger than the number of available bits, the minimum number of hash collisions will be $(N-M)$ and hence the smallest possible value for the average probability of collision becomes:

$$P_1 \geq 1 - \frac{M}{N} = 1 - 2^{m-n}. \qquad [2]$$

This simple approximation will be more accurate as $N >> M$ or $n >> m$, which is the domain targeted by the bitstate hashing technique.

Multi-Bit Hashing

The performance of the single bitstate hashing technique described above can be improved by using multiple hash functions [H91]. The verification tool SPIN, for instance, by default uses two hash functions, and stores two bits in the bit array for every state. A hash collision now requires a collision on both bits.

Note carefully that this does not mean that each state is hashed down from S to 2 bits of information. Each state is hashed down to two bit *positions* in the bit-array M, thus encoding up to $2log(M)$ bits of information. Because of the bit-array, only 2 bits of

memory are required to store this information.

To estimate the expected probability of collision for h simultaneous hash functions, we again distinguish two cases: $h \cdot N < M$ and $h \cdot N \geq M$.

When $h \cdot N < M$, the probability of collision with h independent hash functions is maximally:

$$P_h \leq \frac{1}{N} \sum_{i=0}^{N-1} \left(\frac{h \cdot i}{M}\right)^h \leq \frac{(2\ h)^h}{(h+1)} (P_1)^h \quad [3]$$

which for $h = 2$ reduces to:

$$P_2 \leq \frac{16}{3}\ 2^{2(n-m-1)} \quad [4]$$

The number of states N that can be accommodated in memory is minimally M/h, assuming no overlap in any of the h-tuples that are generated for the N states, and it is maximally $M - h + 1$, if all tuples overlap in a maximum of $h - 1$ places (i.e., the first state occupies h unique bit positions, and each subsequent state then claims just one extra bit). When $h \cdot N \geq M$, the average probability of collisions can therefore be no worse than:

$$P_h \leq 1 - \frac{M}{h \cdot N} = 1 - \frac{1}{h} 2^{m-n}. \quad [5]$$

and it can be no better than given by equation [2]. With increasing N (or shrinking M) the overlap between h-tuples must increase, and with that the value for P_h can be expected to move away from [3] and [5] towards the bounds predicted by [1] and [2].

The last part of the analysis we need is an estimate of the probability of lost states with a traditional exhaustive search. This probability is trivially zero when $N \leq M/S$, and for larger values of N it equals:

$$P_{trad} = 1 - \frac{M}{N \cdot S}. \quad [6]$$

Approximation Error

Many factors can affect the accuracy of analytical estimates of the effects of the bitstate hashing techniques. If, for instance, a state s is reached during the search that produces a hash value colliding with the value for a previous, and distinct, state t, then we count this as a hash collision, even though neither state s nor state t were lost as a result of the collision. The loss may occur only on the *successors* of s, which will now not be explored starting from s itself.

If the state space is well connected [H87] there is likely to be more than one path through the reachability tree that leads to any given state. So a successor of state s may remain reachable from its other ancestors in the reachability tree. It is therefore unpredictable what the net effect of a hash collision will be. Assuming an average of one lost state for every hash collision is in itself an approximation of which the accuracy can only be validated with empirical tests. Introducing greater precision in the formulas, though tempting, is largely self-defeating since the effect of the underlying assumptions about the typical applications of these techniques will soon dominate.

The accuracy of our upper- and lower-bound estimates for the probabilities of hash collision with single or multiple hash functions can only be verified with accurate empirical tests. We will present these tests in Section 4 and confirm that the first order approximations we have made above are reasonable.

Predicted Coverage

The coverages realized for $N = 2^{19}$ and $S = 1376$ bits, and M shrinking from 2^{33}

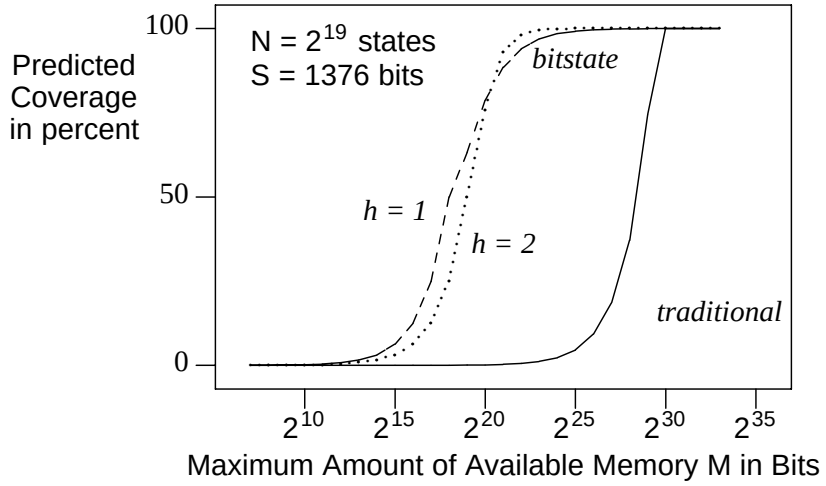

Figure 1. Bitstate Hashing Compared with Traditional Storage.

(sufficient to store all states exhaustively) to 2^7 (insufficient to store even one state) as predicted by equations [1] — [6], are plotted in Figure 1. Except for the estimate for $h=1$ with $N \geq M$, the curves are based on upper-bound estimates for the probability of collision (equations [1], [3], and [5]) and thus give conservative lower-bound estimates for the coverage. The lower part of the curve for $h=1$ (from equation [2]) gives an upper-bound on the coverage, that is predicted to increase in precision with shrinking M.

Note again that for all coverage predictions only the *relative* values of M and N are important. Equivalent coverage plots can be produced if instead of the memory-size M the number of states N is varied. We have chosen to vary M, and use fixed values for N and S that match the values for which we will collect precise empirical evidence in Section 4.

For all three methods, the coverage is 100% when sufficient memory is available, and eventually shrinks to 0%. The three methods differ in how the coverage drops with shrinking M. The double bit hashing scheme ($h=2$) is superior when $2N < M$. Above that limit, the single-bit strategy is predicted to perform slightly better.

Increasing the number of simultaneous hash functions h has two effects.

1. It increases the coverage for higher values of M, and decreases it for lower values.
2. It reduces the range of values of M for which optimal coverage is reached.

The effect is illustrated in Figure 2, which plots the lower-bound estimates for the coverage realized derived from equations [3] and [5].

Based on the observations made earlier at equation [5], an improved estimate for $h=20$ may be obtained by defining an interpolation function of the two curves shown in Figure 2: starting at the upper knee in the right curve, and ending at the lower knee in the left curve. (More precisely, the end point for the interpolation lies on the curve for $h=1$, not $h=2$, but as Figure 1 illustrates, these two curves merge in the area of interest. The accuracy of this approximation is confirmed by the measurements that will be shown in Figure 6.)

Choosing h

In the absence of reliable prior knowledge of the value of the ratio M/N there cannot be a fixed optimal choice for the value of h. Lower values of h produce a higher predicted coverage over a longer range of possible ratios than higher values. To first order of

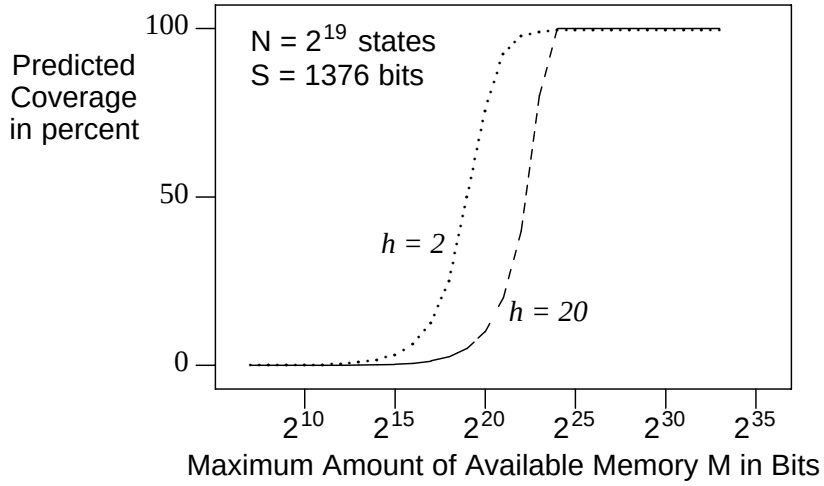

Figure 2. Coverage for Growing Numbers of Hash-Functions.

approximation, in a well tuned verifier, the runtime requirements of the verification depend linearly on the product of h and S [H88]. Increasing the value of h can cause a linear increase in runtimes, but produces only an improvement in coverage within an exponentially shrinking range.

> First note that, using ordinary hash functions, the performance of a reachability analyzer must *minimally* depend linearly on $h \cdot S$, since every state of S bits requires the calculation of h statistically independent hash values. It turns out that all other runtime expenses can be tuned to be smaller than this unavoidable expense. SPIN comes reasonably close to the lower limit, and uses optimized hash functions that avoid (expensive) multiplication and division operations. In principle it would be possible to come still closer, by using the strategies outlined in [H88].

The value $h = 2$ was adopted in SPIN as a compromise between runtime expense and coverage.

3. Alternative Strategies

A search strategy based on the value $h = 20$ was discussed in [WL93], and named the *multihash method*. As noted above, the runtime requirements of this type of search can be ten times than that of a bitstate search with $h = 2$. The expected coverage of a multihash search can exceed that of the double bit hash within a band of applications that is limited on the right-hand side at the point where exhaustive searching becomes possible, and on the left-hand side at the point where there is insufficient memory to store 20 bits per state. This limits the domain of application to $20 \leq M/N \leq S$. When $20 > M/N$ a single or double bit hashing method will exceed the coverage realized by a multihash method.

With N unknown, we cannot predict which algorithm will perform best in practice. A compromise between the multihash method and the single or double bit hash method appears to be possible though, combining the advantages of both. In Section 6 we consider a method that uses the sequential instead of the simultaneous use of h independent hash functions. This *sequential multihash* method incurs the same runtime costs as the simultaneous multihash method, but can increase the benefit for a wider range of possible applications.

The Hashcompact Method

An alternative strategy recommended in [WL93] is called *hashcompact*. In the hashcompact method the state descriptor is compressed from S bits to 64 bits, using a single hash

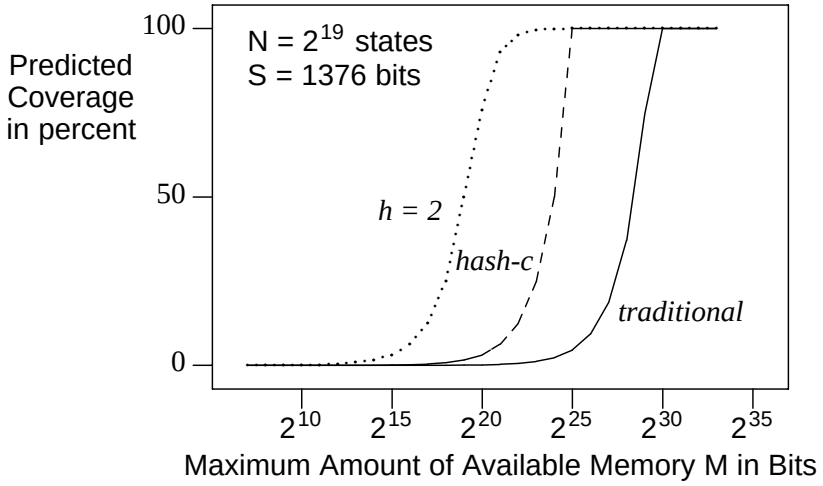

Figure 3. Hash-Compact Compared with Traditional Storage and Bitstate Hashing.

function. The resulting 64-bit values are then stored in a normal lookup table with collision resolution.

The largest number of states that can be stored with the hashcompact method is unavoidably $M/64$. For values of $M < 64N$ the effective coverage is maximally:

$$\frac{M}{64N} \cdot 100\% \;=\; 2^{m-n-6} \cdot 100\ \% \qquad [7]$$

If we assume that the hash values for the storage of the 64 bit keys are not derived from the 64 bits themselves (as would be the normal procedure), but from the original S bits of the state descriptor, with an *independent* hash function [SD95,SD96], the effective number of bits of information stored with this method is $64 + log(H)$, where H is the number of slots in the conventional hash table.

> Alternatively, the states can be compressed from S down to $64 + log(H)$ bits with a single hash function. The first $log(H)$ bits are then used to index the hash table, and the remaining 64 bits are stored.

The coverage achieved by the hashcompact method, assuming *perfect* coverage for all values of $M \geq 64N$, is compared with both traditional storage and with bitstate hashing for $h = 2$ in Figure 3.

For a range of values of M, between the knees in the two right-most curves of Figure 3, the hashcompact method can realize an improvement in coverage. For values $M < 64N$, however, a double bit hashing method is predicted to produce better results.

We can also use a different argument to compare the expected performance of the hashcompact method and double bit hashing. As the ratio of M/N approaches unity, clearly, all storage method must suffer a loss of coverage. The double bit hashing method works optimally up to a ratio for M/N of roughly 100, and then smoothly degrades (cf. Section 5, see also [H91]). This means that in a bit array of M bits, we can reliably store up to $M/100$ states, and gradually lose coverage for $N > M/100$. The hashcompact method can reliably store up to $M/64$ states, and loses coverage quickly for $N > M/64$.

This means that the hashcompact method should be expected to improve over the double bit hashing method in the range

$$\frac{M}{100} \leq N \leq \frac{M}{64}$$

or, equivalently:

$$64 \cdot N \leq M \leq 100 \cdot N$$

which for $N = 2^{19}$ means:

$$2^{25} \leq M < 2^{26} \quad .$$

For other values of M the advantage of a hashcompact method disappears, and, for smaller ratios of M/N, can turn into a disadvantage, also illustrated in Figure 3.

4. Measurement

The effect of the hashcompact method was measured in [WL93] with two experiments, which are reproduced below in Table I.

Table I (From Fig. 7, 'Experimental Results,' in [WL93])

Protocols	Algorithm	Stored States	Memory use	States missed?
DTP	exhaustive	427,567	73 Mbytes	No
	bitstate	427,446	33.8 Mbytes	Yes
	hashcompact	427,567	3.6 Mbytes	No
PFTP	exhaustive	409,257	34 Mbytes	No
	bitstate	405,969	33.9 Mbytes	Yes
	hashcompact	409,257	3.6 Mbytes	No

For the first of these experiments, the DTP protocol, (Data Transfer Protocol) from [HGP92] was first modified as follows. In the original version of the protocol there are 251,409 reachable states [HGP92]. By doubling the number of slots in the message channels, the number of reachable states is increased to 427,567. For the second experiment, using PFTP (the Promela File Transfer Protocol from [H91]), the number of reachable states is also relatively close to 450,000 states.

The hashcompact test in each case consumed $M = 3.6$ Mbyte (= 28.8 Mbit), or approximately 450,000 times 64 bits. For optimal performance, the hashcompact method also requires that M is slightly above the value $64 \cdot N$, which is matched in these experiments. All approximation techniques further assume that $M < S \cdot N$ (see Section 1), and therefore the domain of application for the hashcompact method is restricted to

$$64 \cdot N \leq M < S \cdot N \quad . \qquad [8]$$

when compared to an exhaustive storage technique. In general, the ratio M/N is not known, and the left-hand side of [8] cannot be guaranteed to hold.

Although the data from Table I looks conclusive, it presents only part of the range of possible applications of the algorithms. The measurements in Table II compare the coverage realized with bitstate hashing and with hashcompact for equal amounts of memory M, within a more complete domain of application. For these measurements, the amount of memory M was reduced in five steps from $M = 2^{28}$ bits (2^{25} bytes) to $M = 2^{21}$ bits (2^{18} bytes). To facilitate the comparison with the data in Table I, the memory size is specified in Mbytes. The data in Table II is for the same version of the DTP protocol as tested in Table I. The measurements for the PFTP protocol yield equivalent results.

The data from Table II is included in Figure 4, which also indicates the location of the two data points (**x** and **o**) selected in [WL93] for Table I. For the DTP protocol, for instance, the performance of the hashcompact method under optimal conditions (at the knee in the middle curve) was compared with the performance of the bitstate method under suboptimal conditions (well to the right of the knee in the leftmost curve). Moving the data point for the bitstate algorithm (the circle) across the horizontal part of the curve can produce both better and worse results than the hashcompact method. An assessment of the relative performance of the two methods, therefore can only be based on a side by side comparison of the two methods under equal memory constraints, as illustrated in Figure 4.

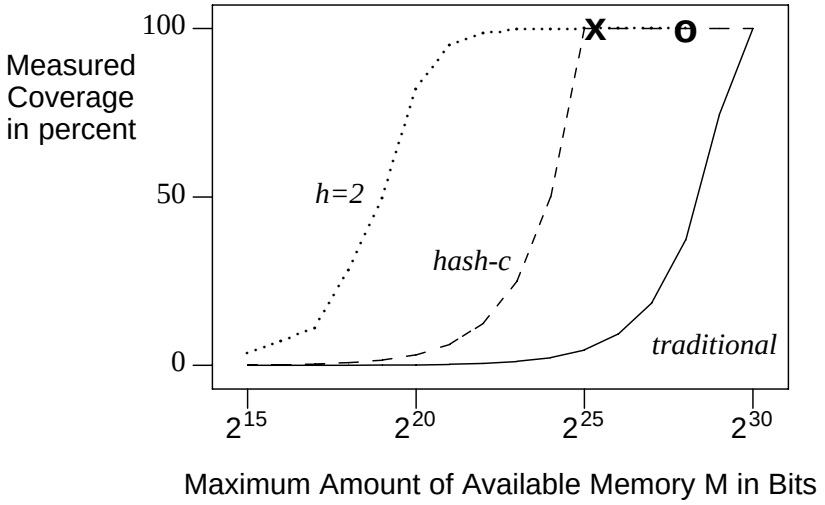

Figure 4. Measured Coverage of Hash-Compact (hc) and Double Bitstate Hashing (h=2), DTP Protocol. Data-Points **x** and **o** are from Table I.

Table II — More Complete Comparison of Bitstate Hashing and Hashcompact (DTP)

Algorithm	Stored States	Memory use (Mbytes)	Coverage (%)
exhaustive	427,567	71.83	100
bitstate	427,446	26.84	99.9
hashcompact	427,567	26.84	100
bitstate	427,459	4.19	99.9
hashcompact	427,567	4.19	100
bitstate	427,097	2.09	99.9
hashcompact	262,144	2.09	61.3
bitstate	425,576	1.05	99.5
hashcompact	131,072	1.05	30.6
bitstate	420,961	0.524	98.5
hashcompact	65,536	0.524	15.3
bitstate	366,438	0.262	85.7
hashcompact	32,768	0.262	7.7

Hashcompact, then, can be seen to improve over the bitstate hash method when $M \geq 64N$, i.e., in the first part of the curve to the right of the point marked with "x" in Figure 4. To the left of the "x" the bitstate hash method works better. Since it is not possible to predict if the value of N will be within the domain targeted by the hashcompact method, we cannot know *a priori* whether the hashcompact method will improve or degrade performance.

Accuracy of the Predictions

With the data from a larger set of measurements, we can also verify the accuracy of the analysis from Section 2. First, in Figure 5 we show data for measurements with $h = 1$ and $h = 2$, compared with the estimates from, respectively equations [1]-[2], and [4]-[5].

The estimates for the double bit hashing method ($h = 2$) are quite accurate. Those for the single bit method ($h = 1$) can be seen to overestimate the coverage realized for especially the lower part of the curves.

Figure 6 compares estimated and measured coverages for the values $h = 2$, and $h = 20$,

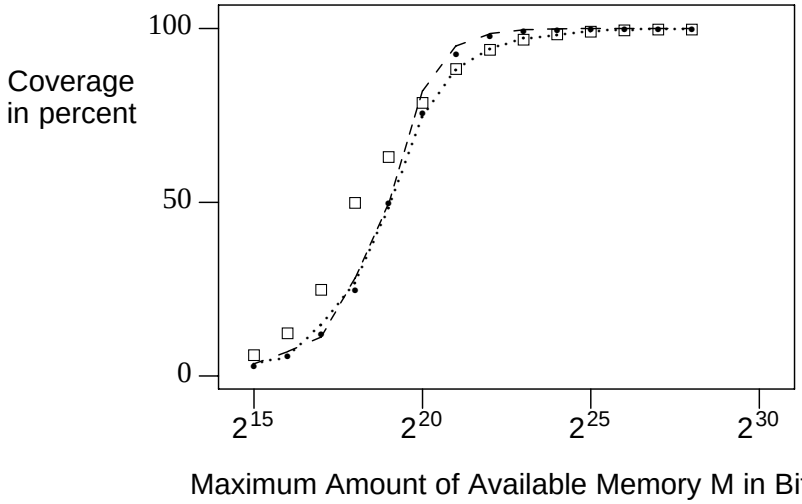

Figure 5. Measured and Estimated Coverage, DTP Protocol.
For $h = 2$, the dashed line plots measured values, and the bullets • calculated values.
For $h = 1$, the dotted line plots measured values, and the boxes □ calculated values.

based on equations [3]-[5]. It confirms that a relatively accurate prediction for the coverage of a multihash method with $h = 20$ may be obtained by defining an interpolation function between the calculated values for $h = 2$ and $h = 20$.

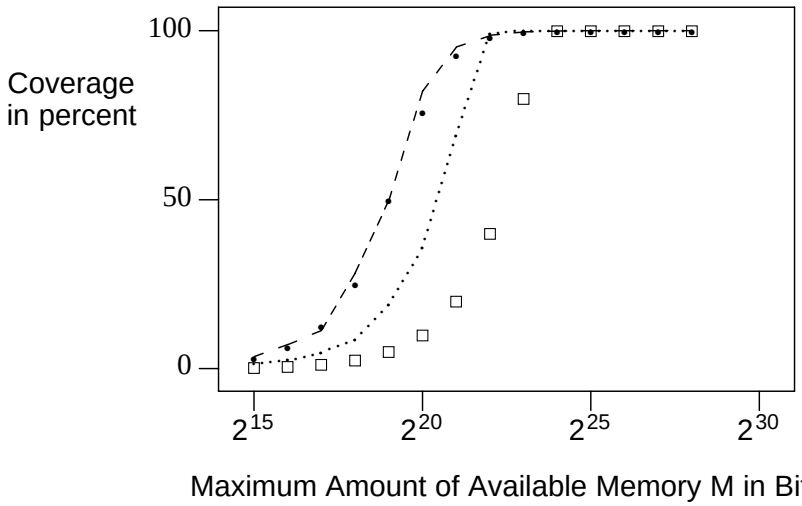

Figure 6. Measured and Estimated Coverage, DTP Protocol.
For $h = 2$, the dashed line plots measured values, and the bullets • calculated values.
For $h = 20$, the dotted line plots measured values, and the boxes □ calculated values.

5. The Hash-Factor

When a bitstate search has been completed, it is of value to have a reliable indication of the coverage that was realized. The only true measure would, of course, be to compare the number of states that is reached with the bitstate methods to those that are reached with a traditional exhaustive search, but in cases of interest the latter will generally be infeasible. SPIN uses a *hash-factor* to serve as a predictor function [H91], which is

defined as:

$$Hf = \frac{M}{N'} \qquad [9]$$

with N' equal to the number of states that was reached, i.e., $N' \approx (1 - P_h) \cdot N$.

Figure 7 plots the hash-factor function [9] against the coverage measured for the double bit hashing scheme, for the same application as before.

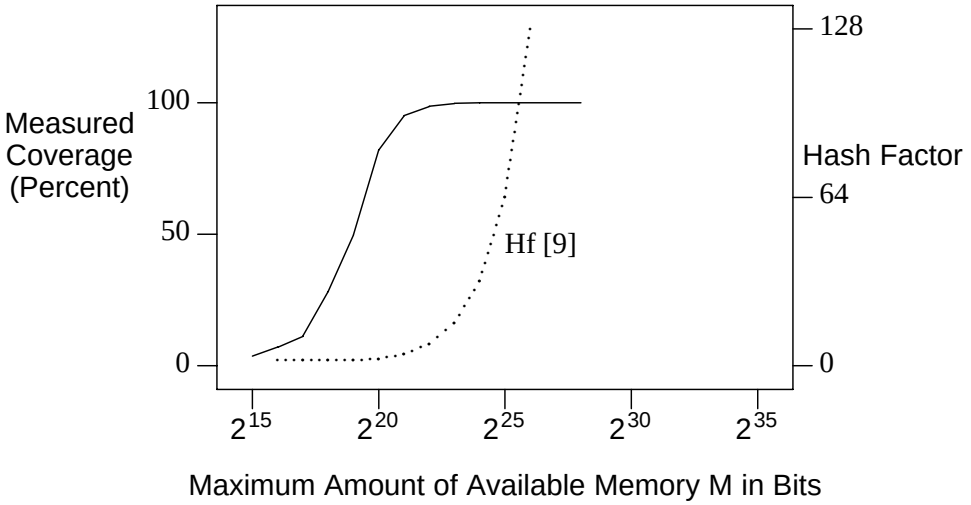

Figure 7. Comparison of Two Predictor Functions for Coverage
The dotted lines give the predictions, the the solid line the measurements for $h = 2$.

The dotted line in Figure 7 tracks the predictor function. SPIN recommends to trust only results of bitstate searches for values of the hash-factor greater than 100. A hash-factor close to 100, but smaller, tends to underestimate the actual coverage realized. A coverage of 99.9% in Figure 7, for instance, corresponds to a hash-factor of 128. A coverage of 93% (solid curve) corresponds to a hash-factor of approximately 4 (dotted curve). The measurement confirms that the hash-factor gives a reasonable indication of coverage.

6. Sequential Multihash

A strategy sketched in previous descriptions of the bitstate hashing technique ([H87], [H88], [H91]), though not implemented until now, is to perform reachability analysis with single- or double bit hashing repeatedly, each time with a different, and independent, hash function. If the probability of collision is p after the first run, under ideal circumstances (more about this below) it reduces to p^2 after a second run, and to p^h after h runs with independent hash functions. Since $p < 1.$, provided that $M > 0$ and $S < \infty$, ideally we have $p^h \to 0$ as $h \to \infty$.

For every hash-collision that occurs between distinct states in each of an initial series of runs with this algorithm, there must exist an independent hash-function that separates these states. Given a sufficiently large value of h, such a collision will be resolved.

This means that under ideal circumstances, for any value of N, and any desired coverage, there exists a value of h for which that coverage will be realized with a sequence of bitstate runs. As with the simultaneous multihash technique, which uses h hash functions in a single run of the algorithm, the expense of the sequential multihash technique is linear in h. Under ideal circumstances, then, the multiple hashings cannot conspire to limit the coverage. Each new application extends the coverage, and with that the range of the method, although even then only by an exponentially shrinking amount.

The ‘ideal circumstances’ alluded to above occur when states can be assumed to be generated more or less randomly. If the reachability graph is well connected, there will be many different paths that lead to the same state in the graph, and the truncation of a small number of paths due to hash-collisions cannot prevent states from the suffix of those paths from being reached. In typical applications the reachability graph represents the joined behavior of N asynchronously executing concurrent processes. The inherent nondeterminism of these applications makes that the assumption of well connectedness is not unreasonable [H87]. The ideal circumstances also require, however, that the total number of hash collisions will be small in comparison to the number of independent paths through the reachability graph.

If either of these two assumptions is not valid, the sequential multihash method clearly cannot extend the coverage arbitrarily far. In the worst case, for instance, all states appear in the reachability graph on a single path. The probability of reaching the $n-th$ state along this path then equals the probability that *none* of the $n-1$ earlier states were affected by hash-collisions. In general, then, it is not useful to explore sequential multihash techniques beyond a modest number of runs.

The hash functions used in SPIN are based an efficient implementation of a CRC (cyclic redundancy check) calculation, with a predefined polynomial [cf. H91]. Two independent hash functions are created by applying the CRC calculations over the state vector in either increasing or decreasing byte-order. The number of independent hash functions can be extended by supplying more CRC-polynomials. In the most recent release of SPIN (version 2), 32 additional hash functions have been provided for this purpose.[2]

Table III — Ideal Performance of Sequential Bitstate Hashing

p_1	h	p_h ($\times 10^{-6}$)
0.0	1	0.00
0.1	6	1.00
0.2	9	0.51
0.3	11	1.77
0.4	15	1.07
0.5	19	1.91
0.6	26	1.71
0.7	37	1.86
0.8	59	1.92
0.9	124	2.12

Table III summarizes how many repetitions h of a bitstate search are *minimally* necessary to reduce the expected number of missed states due to hash collisions to less than 1 (that is to increase the expected coverage to 100%), for a given probability of collision for a single run of p_1 ranging from 0.0 to 0.9, and assuming a state space size equal to 427,567 states, as in Table II. The requirement for h translates into:

$$p_h \leq \frac{1}{427567} \approx 2.34\ 10^{-6}. \qquad [10]$$

A good choice for h for sequential bitstate hashing, is again a tradeoff between expected precision and runtime costs. If necessary, the number of hash functions provided in SPIN could be increased to any number desired. The required CRC polynomials can be generated mechanically.

2. SPIN is available via anonymous ftp from the machine netlib.bell-labs.com, in directory /netlib/spin. More information on SPIN itself can be found in [H91],[H97].

Accuracy of the Predictions

To check the validity of the assumptions on which the sequential bitstate hashing technique is based, we performed a series of measurements on the DTP protocol, used in all experiments reported in this paper. For a first test, all 427,567 reachable states were generated and stored in a large disk file. Then we repeated several tests, in each test computing 128 different single-bit hash-functions for each state, each hash-function based on a different CRC polynomial. Each test thus simulated the result of 128 sequential runs with a single-bit hash under ideal circumstances (more about this below). For each of these 'runs' we counted the total number of the 427,567 hash values computed that were distinct, as a measure of the maximal number of states that could be distinguished in a single run with the chosen hash-function.

The results for two of these tests is shown in Figure 8. The top line in this figure gives the results for hashing into a bitstate of 2^{21} bits (262 Kb), i.e., producing 21 bit hash-values. The bottom line shows the results for a bitstate of 2^{18} bits (32 Kb). Note that for exhaustive storage of the 427,567 reachable states 71.83Mb is needed, cf. Table II.

> For comparison with the earlier results: the first of these test cases (the top curve in Figure 8) corresponds to the bottom two rows in Table II where a direct double-bit hash function gives just 85,7% coverage, and hash-compact gives 7.7% coverage. To stress the techniques still further, the second of these tests uses another factor of eight less in the memory. Single run verifications give extremely poor coverage in these cases. Sequential bitstate hashing seems to be able to reduce the number of missed states surprisingly well.

In a single run in the first test we measured a probability of collision for single-bit hashing of $p_1 = 0.01$. Table III predicts that in this case at least 6 sequential runs would be needed to approximate 100% coverage. The measurement accords with this prediction. In a single run in the second test we measured a probability of collision of $p_1 = 0.51$. Table III gives 19 sequential runs as a minimum, and also here the results of the measurement confirm this.

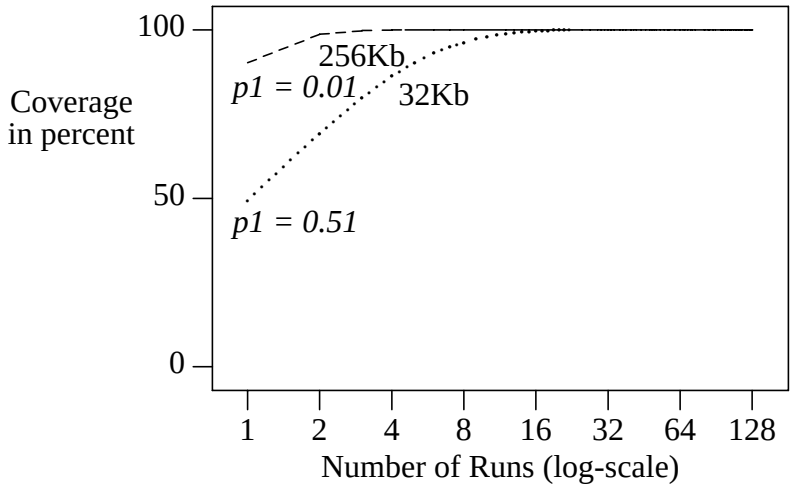

Figure 8. Measured Performance of Sequential Bitstate Hashing.

In both cases, though, the tests were done under idealized circumstances, with no interference from the depth-first search order itself that is typically used in exploring reachable states. As noted above, if states are not generated randomly, but in a predetermined order, the results could be negatively impacted. To measure how large this effect can be, we performed 128 more verification runs with different hash-functions, each time using the verifier itself in single bitstate mode, saving the states generated into large disk-files. Then we measured the total number of distinct states that had been accumulated in these

runs, by combining the results of all 128 disk files.

The result for the first test appears at the top of Figure 9, which reproduces the lines from Figure 8 as solid lines, for reference. The uppermost dotted line in Figure 9 gives the measured coverage for sequential runs with the SPIN verifier, generating states in standard depth-first search order. In this first test, with relatively low probability of loss, the results remain close to the predictions from Table III, reaching 99.7% coverage in 6 sequential runs.

This is different for higher values of p_1. The result for the second test appears as the bottom line in Figure 9, with just above it the reference curve from Figure 8 drawn solidly.

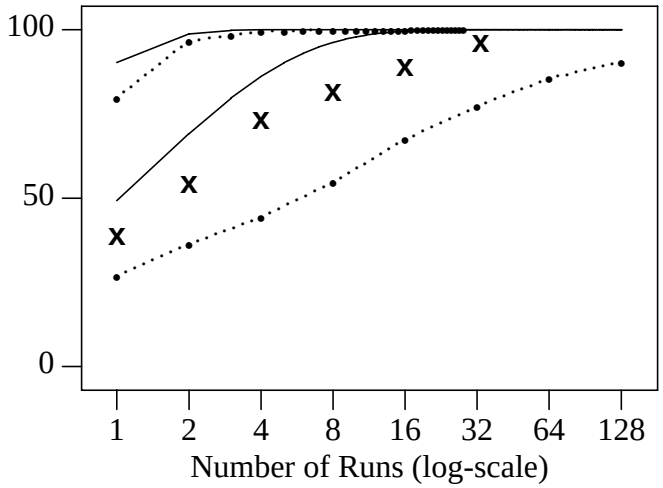

Figure 9. Reducing Effect of Dfs Order on Coverage.
Solid lines: randomly generated states (equal to the two curves in Figure 8).
Dotted lines: the same states generated in normal DFS order
The points marked "**x**" are for a randomized DFS, see 'Randomization.'

In Figure 9 we see that the negative effect of ordered, instead of random, generation of states can be significant for a high initial probability of collision. Instead of the predicted 19 sequential runs for close to 100% coverage, in these tests the coverage reaches only 90% after 128 runs. Each new run performed, however, can be seen to effectively increase coverage. In this case, 128 runs in a memory arena of only 32 Kb, barely enough to store 0.04% of the reachable states, succeeded in exploring 90% of all reachable states. To phrase it differently, more states were explored in 128 tests (386,398) than there are bits in the memory arena that was used to explore them (262,144).

Randomization

The ideal performance from Table III can be approximated more closely if one changes not only the hash-function but also the depth-first search order with each new run. Another way to randomize the search order is to randomize the decision on whether or not to store (or to test for) a newly reached state in the bitstate array. Initial tests with a combination of these methods show that the bottom curve in Figure 9 can be moved considerably closer to the ideal curve above it, e.g., realizing 72.3% (instead of 44%) coverage after 4 sequential runs, improving to 95.2% (instead of 77.2%) coverage after 32 runs. (The data points are marked with an **x** in Figure 9.)

There are of course many trade-offs to be made with such randomization techniques. A more detailed study of these techniques could therefore be of interest, especially for cases where there is a substantial mismatch between problem size and memory constraints, as in the example from Figure 9 (approx. three orders of magnitude difference). For a

probability of loss of 0.1, where the difference in problem size and memory size is two orders of magnitude (256k vs 73 Mb), the sequential bitstate hashing method performs remarkably well.

7. Conclusion

The performance of the bitstate hashing method has been documented and reproduced many times since the technique was first introduced in 1987. In this paper we gave an analytical approximation of the expected coverage of the bitstate hashing method and compared it with two alternative strategies. The alternatives can be shown to improve over the bitstate hashing method in a limited domain of applications. The original bitstate hashing scheme tends to make fewer assumptions about its domain of application, and can outperform its competitors for large problem sizes and/or small memory bounds.

Once a first bitstate analysis of a given problem has been performed, it is often possible to derive an approximation for the actual number of reachable states N. For values in the range $M/2 < N < M/S$ it can be profitable to use a compression scheme that reduces each state from S down to exactly M/N bits. Repeating the analysis with that setting can then be used to maximize the coverage of the run. In general, it should not be expected that compression to a fixed number of bits (such as 20 or 64 bits as proposed in [WL93], 32 or 40 bits as proposed in [SD95,SD96], or indeed 1 or 2 bits as proposed in [H87]) can achieve the same effect in a single run of the algorithm.

We have also shown that a sequentially repeated bitstate hash technique can outperform all other hashing methods for very large problem sizes. No method, of course, can remove the fundamental computational complexity of the reachability analysis problem itself. Sequential bitstate hashing, like other memory management techniques such as state space caching (e.g., [GoHo92]) trades memory use for runtime, and can incur exponentially increasing runtime cost to approximate the results of an exhaustive search in shrinking memory.

Acknowledgements

Doug McIlroy, Rob Pike, Jim Reeds, and Mihalis Yannakakis helped with the analysis. Pierre Wolper, Ulrich Stern, Rob Gerth, and Jean Charles Gregoire gave important feedback on earlier drafts of this paper. The kind help of all is gratefully acknowledged.

References

[C94] Cattel, T. (1994) 'Modelization and verification of a multiprocessor realtime OS kernel,' *Proc. 7th FORTE Conference*, Bern, Switzerland, pp. 35-51.

[C91] Chaves, J. (1991) 'Formal methods at AT&T, an industrial usage report,' *Proc. 4th FORTE Conference*, Sydney, Australia, pp. 83-90.

[GoHo92] Godefroid, P., Holzmann, G.J., and Pirottin, D. (1992) 'State space caching revisited,' *Proc. 4th Int. Conference on Computer Aided Verification*, Montreal, Canada, LNCS Vol. 663, pp. 178-191.

[H87] Holzmann, G.J. (1987) 'On limits and possibilities of automated protocol analysis,' *Proc. 7th IFIP WG 6.1 Int. Workshop on Protocol Specification, Testing, and Verification*, North-Holland Publ., Amsterdam, pp. 137-161.

[H88] Holzmann, G.J. (1988) 'An improved protocol reachability analysis technique,' *Software, Practice and Experience*, Vol. 18, No. 2, pp. 137-161.

[H91] Holzmann, G.J. (1991) *Design and validation of computer protocols*, Prentice Hall, Englewood Cliffs, NJ.

[HGP92] Holzmann, G.J., Godefroid, P., and Pirottin, D. (1992) 'Coverage preserving reduction strategies for reachability analysis,' *Proc. 12th IFIP WG 6.1 Int. Workshop on Protocol Specification, Testing, and Verification*, North-Holland Publ., Amsterdam, pp.

349-363.

[H94a] Holzmann, G.J. (1994) 'The theory and practice of a formal method: NewCoRe,' *Proc. 13th IFIP World Computer Congress*, Hamburg, Germany.

[H94b] Holzmann, G.J. (1994) 'Proving the value of formal methods,' *Proc. 7th FORTE Conference*, Bern, Switzerland, Chapman & Hall, pp. 385-396.

[H97] Holzmann, G.J. (1997) The model checker SPIN, *IEEE Trans. on Softw. Eng.*, Special issue on Formal Methods in Software Practice, Vol. 23, No. 5, May 1997.

[L94] Lin, F.J. (1994) 'Specification and validation of communications in client/server models,' *Proc. 1994 Int. Conference on Network Protocols ICNP*, Boston, Mass., pp. 108-116.

[SD95] Stern, U., and Dill, D. (1995) 'Improved probabilistic verification by hash compaction,' *Proc. IFIP WG 10.5 Advanced Research Working Conf. on Correct Hardware Design and Verification Methods*, pp 206-224.

[SD96] Stern, U., and Dill, D. (1996) 'A new scheme for memory-efficient probabilistic verification,' *IFIP TC6/WG6.1 Joint Int. Conf. on Formal Description Techn. for Distr. Systems and Comm. Protocols, and Protocol Spec., Testing, and Verification, FORTE/PSTV96*, North-Holland Publ., pp. 333-348.

[WL93] Wolper, P. and Leroy, D. (1993) 'Reliable hashing without collision detection,' *Proc. 5th Int. Conference on Computer Aided Verification*, Elounda, Greece, Springer-Verlag, LNCS, pp. 59-70.