Fast *and* Accurate Bitstate Verification for SPIN

Peter C. Dillinger and Panagiotis Manolios

Georgia Institute of Technology
College of Computing, CERCS Lab
801 Atlantic Drive
Atlanta, GA 30332-0280
{peterd,manolios}@cc.gatech.edu

Abstract. Bitstate hashing in SPIN has proved invaluable in probabilistically detecting errors in large models, but in many cases, the number of omitted states is much higher than it would be if SPIN allowed more than two hash functions to be used. For example, adding just one more hash function can reduce the probability of omitting states at all from 99% to under 3%. Because hash computation accounts for an overwhelming portion of the total execution cost of bitstate verification with SPIN, adding additional independent hash functions would slow down the process tremendously. We present efficient ways of computing multiple hash values that, despite sacrificing independence, give virtually the same accuracy and even yield a speed improvement in the two hash function case when compared to the current SPIN implementation.

Another key to accurate bitstate hashing is utilizing as much memory as is available. The current SPIN implementation is limited to only 512MB and allows only power-of-two granularity (256MB, 128MB, etc). However, using 768MB instead of 512MB could reduce the probability of a single omission from 20% to less than one chance in 10,000, which demonstrates the magnitude of both the maximum and the granularity limitation. We have modified SPIN to utilize any addressable amount of memory and use any number of efficiently-computed hash functions, and we present empirical results from extensive experimentation comparing various configurations of our modified version to the original SPIN.

1 Introduction

"Bitstate verification" [10] is a term that has been used by the model checking community to refer to explicit-state model checking with Bloom filters. Explicit-state model checkers, such as Holzmann's SPIN, have been used with great success in a variety of domains, including verification of finite-state, concurrent systems, such as cache coherence and network protocols.

Much of the research in model checking is focused on tackling the *state explosion problem*: a linear increase in the number of components leads to an exponential increase in the size of the resulting models. State explosion is a particularly acute problem in the context of explicit model checking, as memory requirements depend linearly on the size of the state space. Because the most efficient explicit-state techniques call for storing the set of all visited states in core memory, making the representation of the visited set

more compact means larger models can be explored more quickly. By resorting to probabilistic methods of storing the visited set, the representation can be made exceptionally compact, enabling much larger state spaces to be tackled efficiently. The drawback of probabilistic methods, of course, is that there is a possibility of omitting states with errors.

The Bloom filter is a popular choice of data structure for compactly storing sets [2]. The main parameter for tuning a Bloom filter is the number of hash functions used, and the bitstate mode of SPIN utilizes a Bloom filter with 2 hash functions. In [17], Wolper and Leroy promote using 20 hash functions instead of Holzmann's choice of just 2, for in many cases, SPIN would be more accurate if more hash functions were used. However, Holzmann notes that the choice of 2 "was adopted in SPIN as a compromise between runtime expense and coverage," and explains why using more hash functions is impractical [11]:

> In a well-tuned model checker, the run-time requirements of the search depend linearly on k[, the number of hash functions used]: computing hash values is the single most expensive operation that the model checker must perform. The larger the value of k, therefore, the longer the search for errors will take. In the model checker SPIN, for instance, a run with $k = 90$ would take approximately 45 times longer than a run with $k = 2$.

We have discovered a Bloom filter enhancement that gives virtually the same effect as using more independent hash functions, but at a fraction of the runtime cost. For example, this technique alone can produce the effect of 20 hash functions with only 2.3 times the cost of using 2 hash functions—far from Holzmann's factor of 10. In the process of incorporating our technique into SPIN, we discovered other ways of improving the speed and accuracy of bitstate verification in SPIN. More specifically, we show that making more intelligent use of the Jenkins hash function [13] can significantly speed up verification. We tackle issues associated with accommodating an arbitrary amount of memory, and show how this simple issue can easily make orders of magnitude of difference in the possibility of incomplete coverage.

This paper is oriented toward describing and evaluating implementation considerations we made when implementing our modified version of SPIN. The analysis of our techniques in this paper is mostly experimental; a more formal, mathematical analysis of our techniques will appear elsewhere. We refer to our system as "Triple SPIN," or 3SPIN, which is available on the Web for download [5].

Many experimental results are presented throughout. All timings were taken on a 2.53Ghz Pentium 4 with 512MB of RDRAM running Red Hat Linux 7.3. We used version 3.1.1 of the GNU C compiler with third-level general optimizations and all Pentium 4-specific optimizations enabled.

To combat the state explosion problem, in addition to hashing—the main topic of this paper—explicit state model checkers use techniques such as partial order reductions [6, 8] and symmetry reductions [3]. The improvements to bitstate verification discussed in this paper do not affect its compatibility with these techniques, but we have disabled reductions in all of our tests in order to easily measure accuracy.

This paper is organized as follows. In Section 2 we give an overview of Bloom filters and show that they are quite sensitive to the number of hash functions used, *e.g.*,

the expected number of omissions when using two hash functions can be several orders of magnitude greater than the number of expected omissions when using the optimal number of hash functions. Bloom filters employing more than two hash functions were thought to be impractical because of the running time overhead they incur, but in Sections 3 and 4 we present new techniques to address this issue. In Section 5 we address the memory limitations imposed by the current implementation of bitstate verification in SPIN, version 4.0.7. We wrap up with experimental results incorporating all the techniques from the paper in Section 6, and give conclusions and future directions for the research in Section 7.

2 Bloom Filters in Verification

In this section we overview Bloom filters and consider in more detail the trade-offs involved in a Bloom filter and how these apply in the realm of verification. We also present some analysis that sets up a framework for evaluating our results.

For the basics, we turn to Bloom himself [1]:

> [A Bloom filter] completely gets away from the conventional concept of organizing the hash area into cells. The hash area is considered as N individual addressable bits, with addresses 0 through $N-1$. It is assumed that all bits in the hash area are first set to 0. Next, each message in the set to be stored is hash coded into a number of distinct bit addresses, say $a_1, a_2, \ldots, a_d$. Finally, all d bits addressed by a_1 through a_d are set to 1.
>
> To test a new message a sequence of d bit addresses, say $a'_1, a'_2, \ldots, a'_d$, is generated in the same manner as for storing a message. If all d bits are 1, the new message is accepted. If any of these bits is zero, the message is rejected.

In this paper, we refer to the d functions that produce the indices into the bit vector as the "index functions" and use $m, k,$ and n to represent the size, in bits, of the Bloom filter, the number of index functions used, and the number of objects added to the Bloom filter, respectively.

Although Bloom filters can be very compact, the downside is that when a membership query indicates that an element is in the Bloom filter, there is a certain probability of an error—that is, of a *false positive*. If we assume that the index functions are independent and uniform, then the probability that an index function does not select a specific bit is $p = 1 - \frac{1}{m}$. After inserting i elements into the Bloom filter, the probability that a specific bit is still 0 is p^{ki}. Therefore, the probability of a false positive, after i elements have been added to the Bloom filter, is $\left(1 - p^{ki}\right)^k$.

While the false positive rate is the primary metric for evaluation and optimization in many applications of Bloom filters [2], the way we use Bloom filters in verification gives rise to two more meaningful metrics: the expected number of omissions and the probability of having no omissions.

We compute the expected number of omissions when attempting to add n distinct states by adding the probability of a false positive when i states have already been added

to the Bloom filter, as i ranges from 0 to $n-1$.

$$\sum_{i=0}^{n-1} \left(1 - p^{ki}\right)^k$$

To compute the probability of no omissions at all, we start by noting that in a Bloom filter containing i elements, the probability that adding a new element does *not* lead to an omission is just 1 minus the probability of a false positive, $1 - \left(1 - p^{ki}\right)^k$ The probability of there not being an omission at all is just the product of there not being an omission as i ranges from 0 to $n-1$:

$$\prod_{i=0}^{n-1} \left(1 - \left(1 - p^{ki}\right)^k\right)$$

Both verification metrics, the number of expected omissions and the probability of no omissions, depend on $m, n,$ and k. We have very little control over the values of m and n, as m is bound by the amount of memory we have available (the more the better) and n is the size of the transition system under consideration. Therefore, to obtain the best results for a fixed m and n, we have to choose the appropriate value of k. Figure 1 shows that the expected number of omissions is quite sensitive to the number of index functions used (note that we use a log scale on the y-axis). We ran 3SPIN, calling the Jenkins hash function k times, and using a 1MB Bloom filter on an instance of the PFTP problem consisting of 606,211 states. We varied k from 1 to 32 and we report the actual number of omissions, averaged over 100 runs. Notice that the number of omissions when using two index functions is about two orders of magnitude greater than the number of omissions when using eleven index functions.

The second curve in Figure 1 shows the number of expected omissions as given by the formula above for n equal to 606,211. There is a sizable gap between the two curves which at first one may think is due to less-than-ideal index functions, but the disparity is actually caused by a shortcoming of the theoretical analysis, which only considers one of two types of omissions. “Hash omissions” are those states that are omitted because of false positive Bloom filter queries. “Transitive omissions” are those states that are never reached because they are made unreachable by other omissions and, thus, are never queried against the Bloom filter. The gap in Figure 1 is mostly due to transitive omissions; *i.e.*, in our implementation there is an observable number of states (out of the 606,211 in total) that are never even considered. The theoretical analysis is far from useless, however, for as the figure shows, minimizing the number of hash omissions tends to also minimize all omissions.

More significantly, if the number of hash omissions is zero, the number of transitive omissions is also zero; consequently, the probability of no hash omissions is also the probability of no omissions altogether. Unlike expected omissions, the probability of no omissions matches almost exactly experimental results (see Table 1), justifying our “transitive omission” argument for the disparity in Figure 1.

We have seen that using the optimum number of index functions is very important in getting the most accuracy out of Bloom filters. While an analysis showing how to choose k is beyond the scope of this paper, a useful formula for estimating the best k

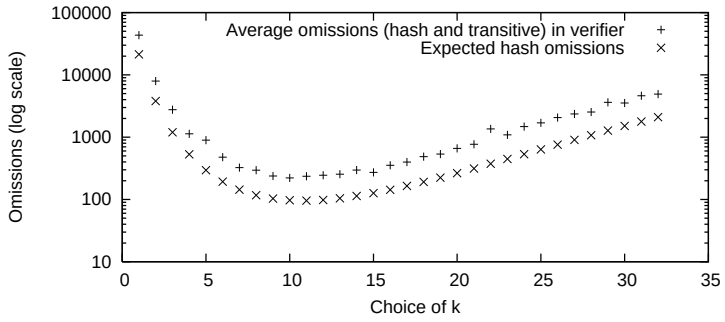

Fig. 1. We show the expected and observed omissions out of 606,211 states using 1MB for the Bloom filter, as k is varied. The theoretical optimum value for k is 11.

given m and n is

$$\lceil 3.8^{\frac{1}{\frac{m}{n}+4.2}} \cdot \frac{m}{n} \cdot ln\ 2 \rceil$$

This closed estimate for the best k in verification was derived by refining a formula from [2], $\frac{m}{n} \cdot \ln 2$, which estimates the k that minimizes the false positive rate. Validation of our formula's accuracy will appear in future work.

3 Double and Triple Hashing

While Bloom filters employing more than two index functions can improve accuracy, they were thought to be impractical because of the running time overhead they incur. In this section we describe techniques for efficiently computing index values from just two or three hash values. Our techniques are similar to the "double hashing" scheme for collision resolution in open-addressed hash tables. While we give a short overview of double hashing below, a good reference is Chapter 11 of [4], and for a more complete account see [14, 7].

3.1 Double Hashing Description

Open addressing refers to a type of hashing where elements are stored directly in a hash table. To insert an element the hash table is probed until an empty location is found, and a query consists of probing the table until either the element is found or it is clear that the element is not in the table. The probing sequence is obtained by applying a sequence of hash functions to an element. Just as with Bloom filters, applying multiple hash functions can incur a significant performance penalty. Double hashing is an efficient way of implementing open addressing which only uses two hash functions to generate a probing sequence. The first value (call it x) is the starting index of the probing sequence. Given some index in the probing sequence, the next is obtained by adding the second value (call it y). The addition is done modulo the number of indices to ensure that the sum is also a valid index.

Our double hashing scheme for Bloom filters is based on this idea: instead of using a sequence of index functions that are computed independently, use two functions, `a` and `b`, to compute values `x` and `y`, and use simple arithmetic on those values to generate all the required indices for each Bloom filter operation:

```
x := a(d) MOD m
y := b(d) MOD m
f[0] := x
i := 1
while i < k
      x := (x + y) MOD m
      f[i] := x
      i := i + 1
```

Note that `f[i]` $=$ `x` $+$ `iy` *MOD m*. Although in this pseudocode we store the index values into an array `f`, in actual code we would use the values as soon as they are computed, and, likewise, only compute as many values as are needed.

We MOD `a(d)` and `b(d)` because we are assuming that `a` and `b` are stock hash functions that have not been tailored to output values in our index space. SPIN requires similar MOD operations to get index values from hash functions, but the designers chose to only allow Bloom filter sizes that are powers of 2 so that efficient bit masking can be used for the MOD operations. Implementation of the MOD operations are discussed in Section 5.2, in which we describe how to efficiently loosen the power-of-two restriction.

As presented, the algorithm has a complication with respect to values of `y`. For example, if `y` $= 0$, only one unique index is probed. One way to fix this problem is to ensure that `y` is relatively non-zero and relatively prime to m. The way 3SPIN deals with this issues is discussed in Section 5.3.

3.2 Double Hashing Example

It may not be clear to those who are not intimately familiar with Bloom filters that our double hashing scheme can actually give higher accuracy than using just two index functions. Figures 2, 3, and 4 demonstrate that boosting k with double hashing can lead to better accuracy.

Figure 2 shows a Bloom filter to which elements 79, 49, and 81 are added. For this Bloom filter $k = 2$; that is, as many as two bits are set for each added element. If we interpret the figure as not using double hashing, the hash functions h_0 and h_1 serve as the index functions, and are defined as $h0(d) = d$ *MOD* 11 and $h1(d) =$ $(d$ *DIV* $11)$ *MOD* 11. We can make the $k = 2$ case of double hashing yield the same pair of indices for all inputs by making `a(d)` $= h0(d)$ and `b(d)` $= (h1(d) - h0(d))$ *MOD* 11. Recall from the double hashing algorithm that we compute the index functions for double hashing with $f_i(d) = (a(d) + i \cdot b(d))$ *MOD m*, and in our example $m = 11$.

Adding 82 in Figure 2 does not change any bits in the Bloom filter and, thus, would have caused a hash omission if we were exploring a state space. If we boost k to 3 with double hashing, however, the collision is avoided and state 81 would not be omitted, as illustrated in Figure 3.

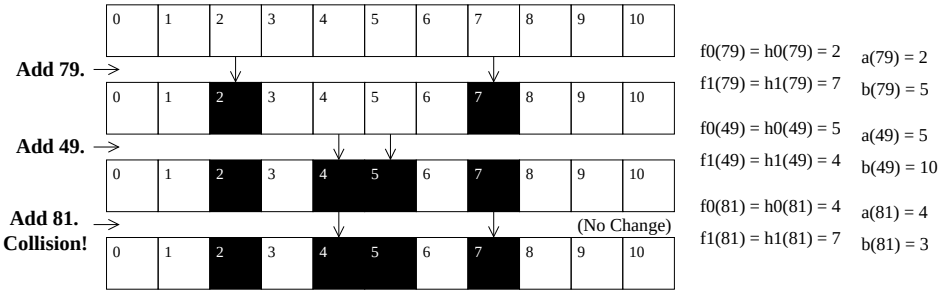

Fig. 2. We add data 79, 49, and 81 to a Bloom filter where $k = 2$ and $m = 11$. A collision occurs when 81 is queried/added. (We have set up the index functions to operates identically with or without double hashing when $k = 2$.)

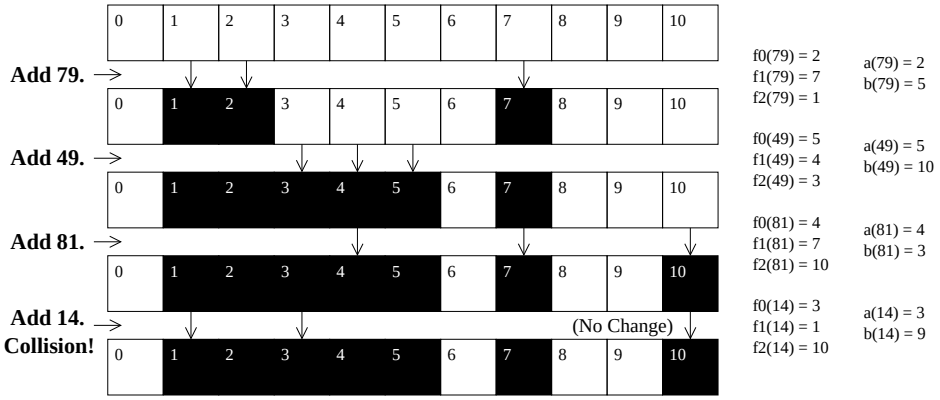

Fig. 3. We add data 79, 49, 81, and 14 to the same Bloom filter as in Figure 2 except $k = 3$ and double hashing is used. A collision occurs when 14 is queried/added.

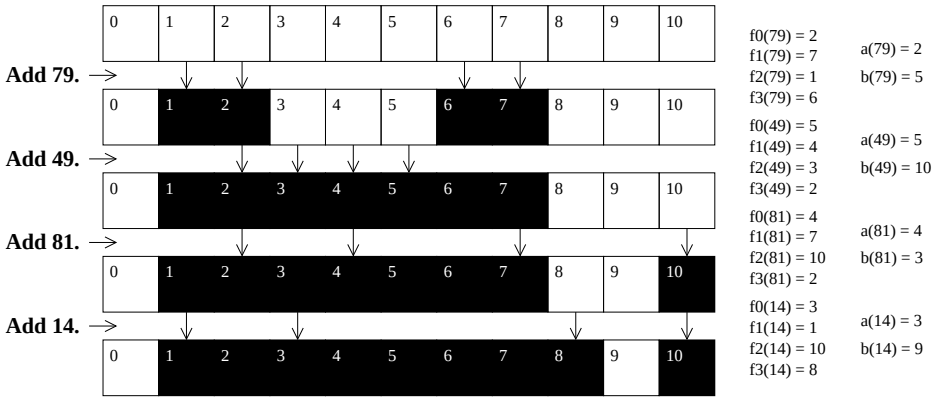

Fig. 4. This is the same example as in Figure 3 except $k = 4$. No collisions occur.

Likewise, adding 14 in Figure 3 would lead to an omission. Figure 4 shows that if 4 double-hashed index functions had been used instead, there would have been no omissions.

We have shown that using double hashing to implement more index functions can yield better accuracy than just using two hash values as indices, but more important is the degree of double hashing's accuracy and how that accuracy compares to using independent hash functions.

3.3 Double Hashing Accuracy

To test the accuracy of double hashing with respect to the expected number of omissions, we ran 3SPIN on a 606,211-state instance of PFTP using both double hashing and independent hash functions, while varying k. Figure 5 contains the results, where each data point is obtained by averaging over 100 runs. Notice that the number of omissions that occur with double hashing is very similar to the number of omissions we get when using independent hash functions. Also, the best choice of k for the independent case seems to be the best for the double hashing case as well.

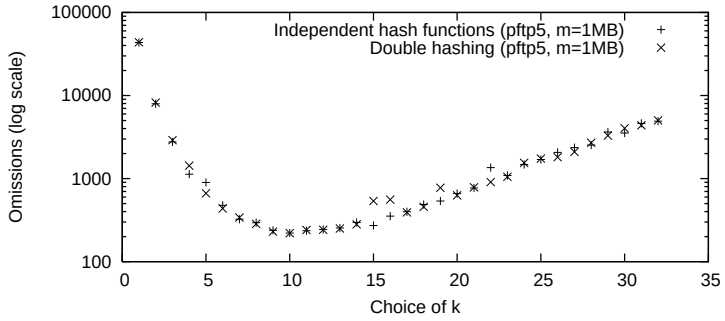

Fig. 5. The above shows the number of omissions from a 606,211-state instance of PFTP when varying the number of index functions and the specified index function implementation. Each data point is the average of 100 iterations. The curve with +'s is the same as in Figure 1.

To test the accuracy of double hashing with respect to the probability of no omissions at all, we ran 3SPIN on the same 606,211-state instance of PFTP with a 2MB Bloom filter using both double hashing and independent hash functions, with k set to 21. A theoretical analysis reveals that verification will be exhaustive 93.4% of the time, and as Table 1 demonstrates, double hashing performs very close to the theoretical expectation, though $4\frac{1}{2}$ times faster than the independent hash function implementation.

The competitive accuracy of double hashing breaks down, however, if one uses enough memory that using independent hash functions would have an almost undetectable probability of omissions, such as 1/16,000. Under such a setup, double hashing still has omissions about 2.5% of the time (see Table 3). This observation motivates something stronger than double hashing that has virtually the same speed.

Table 1. We show the results of verifying a 606,211-state instance of PFTP using 2MB for the Bloom filter, 21 index functions, and the specified implementation of those index functions. Each data point is the average of 20,000 iterations.

Implementation	Full coverage runs	Average running time
Independent	93.281%	19.88 seconds
Double Hashing	92.793%	4.43 seconds
Theoretical	93.383%	N/A

3.4 Triple Hashing

In order to achieve very low probabilities of omission, we have extended the idea of double hashing to what we call triple hashing. The idea is to use a function `c` to compute a value `z` which modifies `y` at each step, which is initially `b(d)`. The implementation of triple hashing is an obvious extension to that of double hashing:

```
x, y, z := a(d) MOD m, b(d) MOD m, c(d) MOD m
f[0] := x
i := 1
while i < k
      x := (x + y) MOD m
      f[i] := x
      y := (y + z) MOD m
      i := i + 1
```

Note that `f[i]` $= \mathtt{x} + \mathtt{iy} + \frac{\mathtt{i(i-1)}}{2}\mathtt{z}$ *MOD m*. The first of two intuitions that can explain the superiority of triple hashing is that we utilize more hash values, and thus collisions in the Bloom filter are less likely to occur. The second intuition is that because the function is more complicated, there is a smaller chance of several indices overlapping with several indices from a single previous addition.

This pseudocode for triple hashing suggests triple hashing would have significantly more per-k overhead than double hashing would, but most of the per-k overhead in double and triple hashing comes from main memory latency. Table 2 demonstrates this. The overhead of triple hashing vs. double hashing at $k = 20$ is not nearly enough to make (Double, $k = 21$) faster than (Triple, $k = 20$), assuming we do not have to compute any more hash values–an assumption that is addressed in Section 4.

Finally, in Table 3 we show that triple hashing can achieve much higher accuracy than double hashing. Triple seems to come much closer to what we expect from independent hash functions, but triple hashing is, of course, much faster than using independent hash functions, as Table 2 confirms.

4 The Jenkins Hash Function

The previous section gave an efficient way to reduce the problem of computing k index values from a state to the problem of computing just two or three. In this section we

Table 2. We present the running times of verifying a 606,211-state instance of PFTP using 2MB for the Bloom filter, the specified number of index functions, and the specified implementation of those index functions, although all implementations did the same amount of hash computation whether needed or not.

Implementation	Index functions	Average running time
Double	21	3.78 seconds
Triple	21	3.84 seconds
Double	20	3.61 seconds
Triple	20	3.73 seconds
Independent	21	9.36 to 19.88s (see Table 4)

Table 3. We show the results of verifying a 606,211-state instance of PFTP using 3MB for the Bloom filter, 30 index functions, and the specified implementation of those index functions. We use 100,000 iterations for each implementation, which is insufficient for quantifying the accuracies with any precision, but does give strong indication of the magnitudes.

Implementation	Proportion of runs with any omissions
Double Hashing	1 in 40
Triple Hashing	1 in 10,000
Theoretical	1 in 16,352

show how to get the most data out of a popular hash function and compute these two or three values in much less time than the default configuration of SPIN 4.0.7 computed the two hash values for $k = 2$ bitstate hashing.

4.1 Getting the most from Jenkins

Because of its high quality and fast speed, Bob Jenkins' LOOKUP2 hash function [13, 12] is a popular choice among implementers of hash tables and Bloom filters; after all, the function is the default hash function in SPIN 4.0.7. Even though it only produces a 32-bit value, the function is often used to produce larger values or sequences of values by calling the function multiple times with different seed values.

If we take a look at the LOOKUP2 function, however, we see that it propagates a full 96 bits of data as it iterates over the input. What the function returns is just a 32-bit fragment of the propagated 96 bits. Although the word returned is the only one that satisfies certain properties that can be tested in Jenkins' lookup2.c [12], we have found that for our purposes, extracting all three words from a single run of Jenkins is about as good as calling the function three times.

4.2 Accuracy Validation

First we ran tests on Jenkins to make sure that each of the three output words are uniform on their own. If this were not the case, two sufficiently large[1], unique, randomly-chosen inputs would have better than a 1 in 2^{32} chance of producing the same 32-bit word of output. Equivalently, a 32-bit output is not uniform if inputs have better than a 1 in 2^8 chance of their output matching one of 2^{24} unique outputs. Running exactly this test repeatedly for each output word yielded probabilities that quickly converged at 1 in 2^8, as desired.

Next we sought to evaluate pairwise independence among the three pairings of output words. We followed a similar procedure to that above, except we were attempting to establish the uniformity of a 64-bit output. Observing only one repeated 64-bit output, we were unable to put an upper bound on the entropy in the output, but our results indicate the entropy is likely greater than 60 bits for each pair of words, leaving no doubt that extracting more than one word from Jenkins gives us access to substantially more hash information, if not a full 96 bits.

From a more practical standpoint, we ran tests to validate that using all three words from each call to Jenkins gives about the same accuracy in a Bloom filter as calling Jenkins three times. Table 4 shows the results of 20,000 executions each for two versions of SPIN, both of which use 21 index functions. The "Slow Jenkins" version uses separate calls to Jenkins for each index function—up to 21 calls for each Bloom filter operation. The "Fast Jenkins" version uses three words from each call to Jenkins and, thus, incurs a maximum cost of seven Jenkins calls per operation. We actually observed slightly higher accuracy with "Fast Jenkins," but the results are not statistically significant enough to establish that relationship. The results do establish that both implementations yield accuracy exceptionally close to what is expected in theory.

NOTE: These tests do not utilize double or triple hashing; the combination of all techniques is tested and validated in Section 6.

Table 4. We show the results of verifying a 606,211-state instance of PFTP using 2MB for the Bloom filter, 21 index functions, and the specified implementation of those index functions. We ran 20,000 iterations of each implementation.

Implementation	Full coverage runs	Average running time
Slow Jenkins	93.281%	19.88 seconds
Fast Jenkins	93.339%	9.36 seconds
Theoretical	93.383%	N/A

[1] We tested using seven words of input.

4.3 Speed Boost

Table 4 also includes execution times for the "Slow Jenkins" version and the "Fast Jenkins" versions. Hash computation clearly dominates the total execution cost, because a 67% reduction in hash computation time resulted in a 53% reduction in overall required execution time.

The "Fast Jenkins" version utilizing three index functions runs more quickly than the Jenkins configuration of SPIN 4.0.7, which uses two index functions, because "Fast Jenkins" can generate about three index functions with a single call to Jenkins. The results of these tests are in Table 5. The Jenkins configuration of SPIN 4.0.7 is the $k = 2$ case of what we have been calling "Slow Jenkins".

Table 5. We show the execution times for verifying a 606,211-state instance of PFTP using 2MB for the Bloom filter. The number and implementation of the index functions is indicated in the table.

Implementation	Index functions	Average running time
SPIN Jenkins	2	2.57 seconds
Fast Jenkins	2	1.86 seconds
Fast Jenkins	3	2.09 seconds

5 Arbitrary Memory Utilization

Two restrictions on the amount of memory that can be utilized by a Bloom filter in SPIN can have profound effects on the accuracy of bitstate verification. The first and most clearly debilitating limitation is the upper limit on the amount of memory that can be dedicated to a Bloom filter, 512 Megabytes[2]. The second limitation is that the Bloom filter in SPIN can only be sized to be a power of two. The impact of both limitations is great: theoretical analysis shows that a user of accurate bitstate verification who dedicates 768MB of memory to the Bloom filter instead of 512MB could have a 1 in 10,000 chance of any omissions instead of 1 in 5.

5.1 Increasing the Maximum

The reason for SPIN's maximum of 512 Megabytes dedicated to the Bloom filter is that 512 Megabytes is equal to 2^{32} Mega*bits*, and 32-bit values are used to index into the bit vector. The problem is that as byte- or word-addressed memories get close to the size of their address space, single words become insufficient for addressing individual bits across most of memory. The computer market is currently experiencing this problem, in which many 32-bit machines are sold with more than 512 Megabytes of memory.

[2] SPIN 4.0.7 would actually only work with up to 256MB for us, but analysis suggests that this is an implementation bug and not a design flaw.

Any solution to this problem would almost certainly involve more computation, so the best solution is likely to be one that eliminates the need for some existing computation. An example of such existing computation is the process of dividing a bit vector index into a word or byte index and an index of the bit within that word or byte. These operations boil down to dividing by some small power of two and taking the modulus with that same power of two, which can be implemented with bit shifting and bit masking, respectively.

Our solution, which we call "parallel indexing," accommodates any addressable amount of memory and eliminates a little bit of previously required per-k computation by computing word indexes and bit-within-word indexes independently. Consider having two sets of index functions: $F_0, F_1, \ldots, F_{k-1}$ give the addresses of the words to retrieve and $f_0, f_1, \ldots, f_{k-1}$ tell which bit to extract from each word. We can apply triple hashing on an A, B, and C to get the F_i values and the same on a, b, and c to get the f_i values. Because none of these functions is ever required to return more bits than can be stored in a word, the computation is simple.

5.2 Precise Granularity

Modifying SPIN to use any specified amount of memory for the Bloom filter is simple, but ensuring that accuracy is maintained and that the implementation is efficient is not as easy. The simple answer to using any amount of memory is to allocate that much, and then MOD hash function results by the appropriate value whenever indexes are computed.

Accuracy The first problem with the simple answer is that MOD-ing by any value can affect the accuracy of the data structure. Consider a case in which one is *not* using "parallel indexing" and allocates about $\frac{2}{3}$rds of 2^{32} bits, about 341MB, for the Bloom filter. If we MOD the result of a 32-bit hash function to get an index, the first half of the indexes are twice as likely to be chosen as the second half. We can think of the MOD operation as putting the input values into m equivalence classes. No matter how hard we try to make the distribution among classes more uniform than what MOD gives us, if we have 50% more elements than equivalence classes, half of the classes are going to contain two elements and half are going to contain one element.

Our choice of indexing words as opposed to bytes in the parallel indexing scheme lessens the impact of the uniformity problem by a factor of four (in the 32-bit case), making the problem unlikely to ever have an observable impact. Whereas byte indexing gave a worst case of some indexes being twice as likely to be chosen as others, word indexing yields a worst case of some being 25% more likely. So even if m is a few Gigabytes, the difference is not significant, as Table 6 reveals.

Speed The simple solution's second problem is that MOD operations on arbitrary values are much more costly than, for example, taking a modulus with respect to a power of two, which can be implemented with bit masking. In fact, outside of SPIN we have observed C's unsigned modulus operator to be ten times as slow as bit masking on a

Table 6. We show the result of verifying a 723,035-state instance of LEADER using 2MB for the Bloom filter and 17 independent index functions, while varying the ratio of the probability of an index landing in the first half of the bit space over the second. Each data point is the average of 420 iterations.

Case	Ratio	% of full coverage runs
Byte indexing	2 : 1	16.16%
Word indexing	5 : 4	39.29%
Uniform	1 : 1	41.19%

Pentium 4. Which MOD operations can we optimize away if using double or triple hashing and the parallel indexing scheme?

The first observation is that the range on the bit-within-word indexes are always powers of two and, thus, can be optimized with bit masking.

While it is important for the a values in computing word indexes to have a fairly uniform distribution over all possible indexes, cheating on b and c does not sacrifice as much. In fact, we can MOD with respect to the greatest power of 2 less than m to compute values for b and c, enabling us to use bit masking for these.

Although we have reduced the number of unoptimized MOD operations for the initialization phase of each Bloom filter operation to just one (computing a), the most important MOD operations to optimize are those that happen within the iteration part of each Bloom filter operation, executing as many as k times for each Bloom filter operation.

In the triple hashing case, we can cheat even further on the ranges of values for b and c and eliminate the MOD for `y := y + z` altogether. More specifically, if $b + c \cdot k < m$ then `y` (initialized to b) will never overflow with respect to m, because `y` only needs to be incremented by `z` (whose value is c) $k-1$ times.

The following observation allows us to speed up the MOD for `x := (x + y)`: on each iteration of the loop we are guaranteed that $0 \leq \mathtt{x} < m$ and $0 \leq \mathtt{y} < m$. Thus, $0 \leq \mathtt{x} + \mathtt{y} < 2m$, leaving only two cases to handle: $(\mathtt{x} + \mathtt{y})$ *MOD* $m = \mathtt{x} + \mathtt{y}$ (if $\mathtt{x} + \mathtt{y} < m$) and $(\mathtt{x} + \mathtt{y})$ *MOD* $m = \mathtt{x} + \mathtt{y} - m$ (otherwise; $m \leq (\mathtt{x} + \mathtt{y}) < 2m$). We update this line from the pseudocode to reflect the optimization:

```
x := (x + y) MOD m
```

to be these lines:

```
x := x + y
if (x >= m) then x := x - m
```

The new code is much more efficient, as the graph in Figure 6 shows. The faster version in the graph implements triple hashing and all the optimizations discussed in this section, requiring just one unoptimized modulus per Bloom filter operation. The slower version does not use the optimization just described, requiring up to k unoptimized modulus operations per Bloom operation.

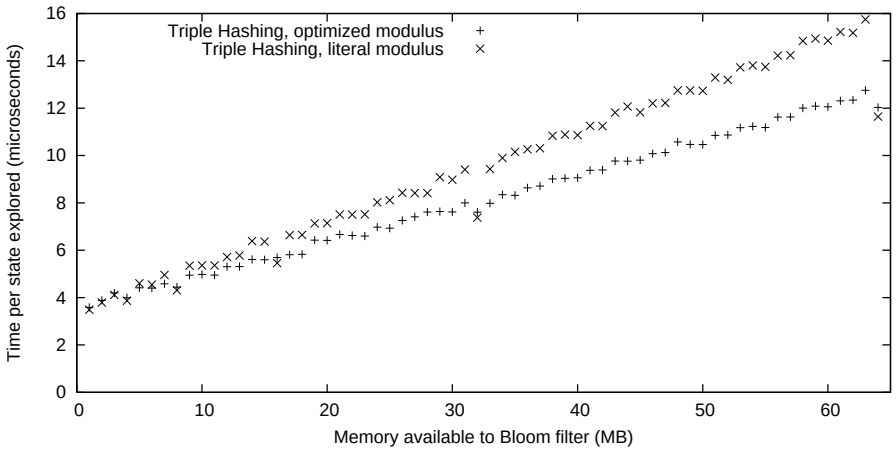

Fig. 6. This graph plots verification times for PFTP5 ($n = 606,211$) and depicts the difference in execution times resulting from optimizing $k-1$ modulus operations per Bloom operation. k is varied from 1 to 27 to be optimal with respect to m/n.

One thing to notice about the times reflected in the graph is that whenever the memory space is a power of two, both implementations run at the same slightly faster speed. Our modified version of SPIN dynamically picks the implementation best suited for the choices of m and k. There are implementations optimized for when m is a power of two and, orthogonally, for when $k \leq 2$.

From the observation that the power of two cases are optimized in the graph, we see that even after our optimizations for utilizing arbitrary memory, we can still incur an execution speed cost of up to a few percent. Such an overhead is likely to be well worth the cost if it enables someone to use nearly twice as much memory.

5.3 With Our 96-bit Jenkins

In this short section we reveal the synergy between the various approaches to improving the speed and accuracy of bitstate verification in SPIN.

With the 96 bits we get from a single call to Jenkins, we have enough hash information to effectively utilize triple hashing, parallel indexing, and precise memory utilization. We call our version incorporating all of these enhancements "Triple SPIN," or 3SPIN for short. Figure 7 has the precise breakdown of hash information used in 3SPIN.

A, *B*, and *C* give 3SPIN triple hashing on the word indexes, and *a* and *b* give double hashing on the bit-within-word indexes. Notice that we only use 4 bits for *b* even though it could be a 5-bit value; the reason is that making *b* odd (by multiplying by two and adding one) ensures that every bit-within-word index is unique up to $k = 32$. This guarantee ensures that each Bloom filter operation addresses k *unique* bit positions in the bit vector.

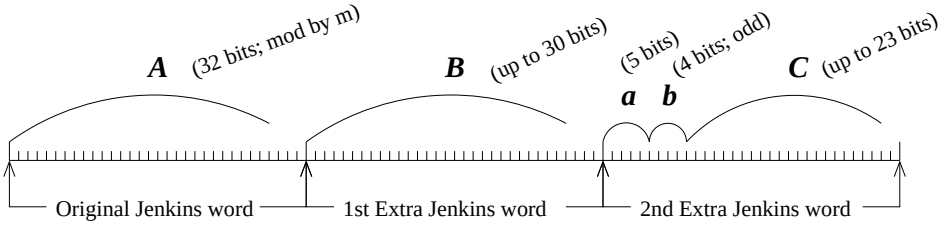

Fig. 7. The above diagram shows how we utilize the 96 bits of output from Jenkins.

6 Overall Evaluation

In this section we evaluate 3SPIN, the system incorporating all the techniques described in this paper. Figure 8 shows that the observed average omissions for the various implementations is so close that it is hard to detect any differences. As previous tests have shown, we would need many high-accuracy runs to have a chance of distinguishing the implementations based on accuracy.

Figure 8 also shows the execution times for the tests. Unlike the number of omissions, the execution times are *profoundly* different, with our techniques taking about 1/4 the time of the implementation not taking advantage of our improvements when $k = 14$. Notice also that our $k = 14$ takes less than twice as much time as our $k = 2$—a far cry from Holzmann's experience with independent hash functions [11, 10], which suggests $k = 14$ to be seven times as slow.

Figures 9 and 10 show the results when various amounts of memory are available for allocation to the Bloom filter. Notice that the versions that do not incorporate any of our enhancements for arbitrary memory allocation to the Bloom filter can only utilize an amount of memory equal to the greatest power of two not greater than m. For example, when 48MB is available, the unenhanced versions act as if only 32MBs are available, because that is the most the user can specify without requiring more than 48MB. If only 32MB is available, all versions using the best k (14 in this case) expect around 100 omissions, but when 48MB is available, 3SPIN expects about 1/100th as many omissions. Even though 3SPIN is using $k = 21$ to make best use of the 48MB, it runs in about 2/3rds the time. If, once again, only 32MB were available, 3SPIN would run in about half the time of the version with independent hash functions.

The version using two independent index functions was included in Figures 9 and 10 to reflect what is available in the latest version of SPIN, 4.0.7. According to these results, 3SPIN can utilize about 7 index functions (m = 14MB in this case) as fast as SPIN 4.0.7 can utilize two, and at that point 3SPIN expects about 1/13th as many omissions, partially because it is utilizing more memory and partially because it is using a more suitable k.

Our last set of experimental results (Table 7) just confirm that our results generalize to models other than those we have used in the rest of the paper.

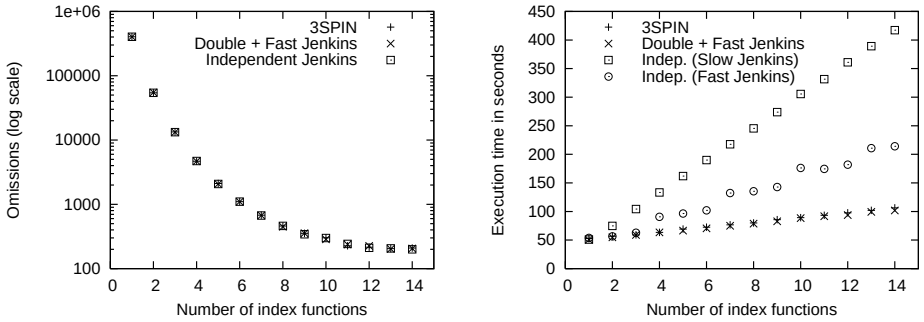

Fig. 8. On the left we have plotted the number of omissions from a 14,536,469-state instance of PFTP(D=1,Q=2) using 32MB for the Bloom filter and k values up to 14, the optimal for this m and n. The right shows the execution times for the same tests. Each data point represents about 5 iterations.

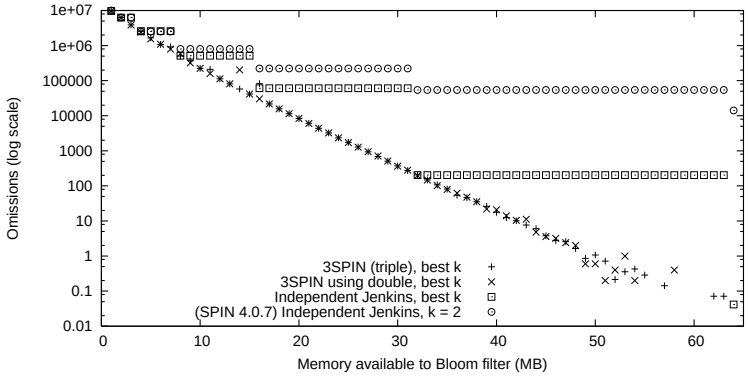

Fig. 9. Here we have plotted the number of omissions from PFTP(D=1,Q=2) for various implementations and various amounts of memory available to the Bloom filter. Notice that implementations only supporting power-of-two granularity will only utilize the greatest power of two less than or equal to the amount available. Each data point is the average over about 20 iterations.

Table 7. Validation of our approaches using models other than PFTP. In each case, all our techniques and optimizations are used. The k values are annotated with either "(opt)", indicating that the choice was optimal for m and n, or "(sub)" indicating we chose a k different from the optimal. All these models are included in the SPIN distribution.

Model	States	m	k	% runs full	Expected	Iterations
Peterson4	7,308,888	32MB	25 (opt)	99.11%	99.15%	336
Leader7	723,035	3MB	8 (sub)	77.34%	75.69%	331
Sort9	2,509,313	8MB	20 (opt)	59.88%	63.38%	329

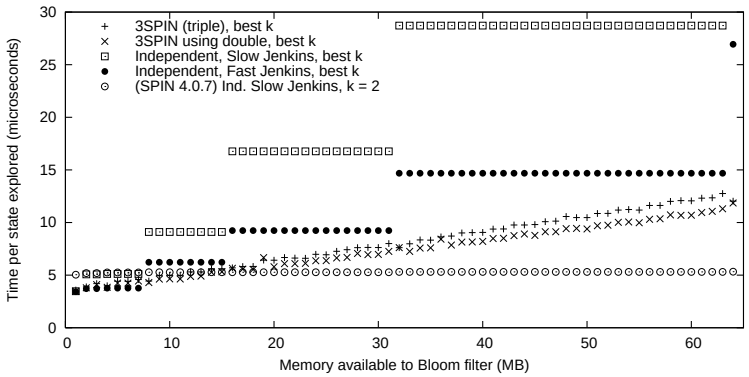

Fig. 10. This graph shows times for the executions in Figure 9. Note that four implementations use the optimal values for k, which range from 1 to 27 depending on m. The other implementation, SPIN 4.0.7, always uses $k = 2$, which explains why it becomes the fastest at $m = 14$MB.

7 Conclusions and Future Work

Early work by Holzmann and others has shown the utility of the Bloom filter data structure for probabilistically verifying systems with explicit state model checkers [9]. The main parameter for tuning a Bloom filter is the number of hash functions used, k, but there is a tension between accuracy and efficiency, as small values of k lead to fast running times, but the value of k that yields the best accuracy may be quite large. SPIN is optimized for speed, and, thus, it only allows k to be 1 or 2. Holzmann justified this choice by pointing out that running a well-tuned model checker with 2 hash functions can be $\frac{j}{2}$ times faster than using j hash functions. The belief was that one could get accuracy or efficiency, but not both.

We show that you can have your cake and eat it too. We have entitled this paper "Fast *and* Accurate Bitstate Verification for SPIN," because that is exactly what we provide with 3SPIN, a system we developed by modifying SPIN 4.0.7. Key components of 3SPIN include our double and triple hashing techniques for Bloom filters, which greatly reduce the execution time of highly-accurate bitstate verification. In fact, 3SPIN can use about 7 hash functions while running as fast as SPIN (using 2 hash functions).

3SPIN also has the ability to use as much main memory for the Bloom filter as is available, whereas SPIN only allows the size of the Bloom filter to be a power of 2, up to 512MB. The motivation behind this improvement is simple: using more available main memory for the Bloom filter *always* improves the expected accuracy of a bitstate search. For example, by using just 50% more memory than SPIN allows, we can be 2,000 times less likely to have an omission.

For future work, we plan to explore the use of Bloom filters in verification from a more analytical standpoint and to examine the impact of this work on techniques such as sequential multihashing [10]. We also plan to compare our techniques to other probabilistic verification techniques such as hash compaction [15, 16].

References

1. B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.
2. A. Broder and M. Mitzenmacher. Network applications of Bloom filters: A survey. In *Proc. of the 40th Annual Allerton Conference on Communication, Control, and Computing*, pages 636–646, 2002.
3. C.N. Ip and D.L. Dill. Better verification through symmetry. In D. Agnew, L. Claesen, and R. Camposano, editors, *Computer Hardware Description Languages and their Applications*, pages 87–100, Ottawa, Canada, 1993. Elsevier Science Publishers B.V., Amsterdam, Netherland.
4. T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.
5. P. C. Dillinger. 3SPIN Home Page. http://www.cc.gatech.edu/~peterd/3spin/.
6. P. Godefroid and P. Wolper. A partial approach to model checking. In *Logic in Computer Science*, pages 406–415, 1991.
7. G. H. Gonnet. *Handbook of Algorithms and Data Structures*. Addison-Wesley, 1984.
8. G. Holzmann and D. Peled. Partial order reduction of the state space. In *First SPIN Workshop*, Montrèal, Quebec, 1995.
9. G. J. Holzmann. Algorithms for automated protocol validation. Technical Report 69:32-44, AT&T Technical Journal, 1990.
10. G. J. Holzmann. An analysis of bitstate hashing. In *Proc. 15th Int. Conf on Protocol Specification, Testing, and Verification, INWG/IFIP*, pages 301–314, Warsaw, Poland, 1995. Chapman & Hall.
11. G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, Boston, Massachusetts, 2003.
12. B. Jenkins. http://burtleburtle.net/bob/c/lookup2.c, 1996.
13. B. Jenkins. Algorithm alley: Hash functions. *Dr. Dobb's Journal*, September 1997.
14. D. E. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison Wesley Longman Publishing Co., Inc., 2nd edition, 1997.
15. U. Stern and D. L. Dill. Improved probilistic verification by hash compaction. In *Correct Hardware Design and Verification Methods*. Volume 987 of Lecture Notes in Computer Science, 1995.
16. U. Stern and D. L. Dill. A new scheme for memory-efficient probabilistic verification. In *Joint Int. Conf. on Formal Description Techn. for Distr. Systems and Comm. Protocols, and Protocol Spec., Testing, and Verification*, pages 333–348, 1996.
17. P. Wolper and D. Leroy. Reliable hashing without collision detection. In *5th International Conference on Computer Aided Verification*, pages 59–70, 1993.