Precision On Demand: An Improvement in Probabilistic Hashing

Igor Melatti, Robert Palmer, and Ganesh Gopalakrishnan

UUCS-07-002

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

1 February 2007

Abstract

In explicit state (enumerative) model checking, state vectors are often represented in a compressed form in order to reduce storage needs, typically employing fingerprints, bit-hashes, or state signatures.

When using this kind of techniques, it could happen that the compressed image of a non-visited state s matches that of a visited state $s' \neq s$, thus s and potentially many of its descendants are omitted from search. If any of these omitted states was an error state, we could also have *false positives*. We present a new technique which reduces the number of omitted states, by requiring a slightly higher computation time, but without employing any additional memory.

Our technique works for depth-first search based state exploration, and exploits the fact that when a non-terminal state t is represented in the hash table, then one of the successors of t (the first to be expanded next, typically the left-most) is also represented in the visited states hash table. Therefore, instead of backing off when the compressed state images match, our algorithm persists to see if any of the left-most successors also matches (the number of successors which are considered for each state is user-defined, thus we name our approach Precision on Demand or POD).

This paper provides a scientific evaluation of the pros and cons of this approach. We have implemented the algorithm in two versions of the Murphi explicit state model checker, one based on hash compaction and the other based on Bloom filters, and present experimental results. Our results indicate that POD-hashing has the potential to reduce storage requirements - or increase the number of bugs likely to be caught when operating within a given amount of storage, with the execution time likely to increase by a factor of 1.8 or less.

Precision On Demand: An Improvement in Probabilistic Hashing *

I. Melatti[1,2] **, R. Palmer[1], and G. Gopalakrishnan[1]

[1] School of Computing, University of Utah
{melatti,rpalmer,ganesh}@cs.utah.edu
[2] Dip. di Informatica, Università di Roma "La Sapienza",
melatti@di.uniroma1.it

Abstract. In explicit state (enumerative) model checking, state vectors are often represented in a compressed form in order to reduce storage needs, typically employing fingerprints, bit-hashes, or state signatures.
When using this kind of techniques, it could happen that the compressed image of a non-visited state s matches that of a visited state $s' \neq s$, thus s and potentially many of its descendants are omitted from search. If any of these omitted states was an error state, we could also have *false positives*.
We present a new technique which reduces the number of omitted states, by requiring a slightly higher computation time, but without employing any additional memory.
Our technique works for depth-first search based state exploration, and exploits the fact that when a non-terminal state t is represented in the hash table, then one of the successors of t (the first to be expanded next, typically the left-most) is also represented in the visited states hash table. Therefore, instead of backing off when the compressed state images match, our algorithm persists to see if any of the left-most successors also matches (the number of successors which are considered for each state is user-defined, thus we name our approach Precision on Demand or POD).
This paper provides a scientific evaluation of the pros and cons of this approach. We have implemented the algorithm in two versions of the Murphi explicit state model checker, one based on hash compaction and the other based on Bloom filters, and present experimental results. Our results indicate that POD-hashing has the potential to reduce storage requirements - or increase the number of bugs likely to be caught when operating within a given amount of storage, with the execution time likely to increase by a factor of 1.8 or less.

1 Introduction

Explicit state model checking continues to have many strengths over symbolic model checking as well as abstraction/refinement methods in many real situations, including the verification of high level cache coherence protocol models [1],

* This work was funded in part by: NSF award CNS-0509379, Intel/SRC grant TJ 1318-001, and Microsoft HPC Institutes.
** Corresponding Author: Igor Melatti. Tel: +39 06 4991 8431 Fax: +39 06 8541842

verification of software systems through model checking [2], and direct model checking of application codes [3, 4]. Time/space trade-offs are a fundamental issue in computing - and clearly to explicit state model checking. In explicit state (enumerative) model checking [5], state vectors are often represented in a compressed form in order to reduce storage needs. Popular techniques in this area include the use of fingerprints [6], bit-state hashing [7], state signatures [8], or Bloom filters [9].

These techniques all have one main drawback: they may *omit* (i.e. not visit) some states. In fact, it could happen that the compressed image of a non-visited state s matches that of a visited state $s' \neq s$, thus s and potentially many of its descendants are omitted from search (those s descendants reachable other than through s may still be visited). Note that, if any of the omitted states is an error states, and none of the other reachable states is a bug, we have a *false positive* verification results. However, these techniques are usually able to provide an estimation of the *omission probability* (i.e. the probability to have at least one omitted state) they lead to. Intuitively, the more bits there are in the image of the compression scheme, the lower the probability of collision and therefore the lower the probability of omission. The currently reported values for the number go anywhere from 3 bits (Murphi with Bloom filters [10]) and 64 bits (for TLC [11] and SPIN [12]), passing by 40 bits (for Murphi [13]). Even in software model checkers such as CMC [14] that compress much larger state vectors than in Murphi models and TLA+ models, these are the reported numbers.

In order to counteract the omitted states problem, we present a new technique which reduces the number of omitted states (thus lowering the omission probability too), by requiring a slightly higher computation time, but without employing any additional memory. Our approach lies on depth-first search based state exploration; however, breadth-first variants may also be possible. In fact, our technique exploits the fact that when a non-terminal state t is represented in the hash table, then one of the successors of t (the first to be expanded next, say the left-most) is also represented in the visited states hash table. Therefore, instead of backing off when the compressed state images match, our algorithm persists to see if any of the left-most successors also matches (the number of successors which are considered for each state is user-defined, thus we name our approach Precision on Demand or POD). Intuitively, it can be seen that the probability of successively suffering from such collisions will multiplicatively diminish. While such an obvious idea (of course, on hindsight!), our extensive search among experts we know, as well as the surveying the available literature revealed *no* evidence of this idea having been proposed or thoroughly studied. This paper provides a scientific evaluation of the pros and cons of this approach, offering the following contributions: (i) A depth-first model-checking algorithm for POD Hashing that is orthogonal to the compressed state representation. (ii) Experimental validation of the benefits of POD Hashing. We also note that this paper addresses only safety (invariance) model checking; liveness is a topic of future work.

Summary of key results and observations Since the basic ideas behind this paper have already been expressed, we now provide a taste of the results. To a first approximation, methods such as 40-bit hash compaction [8] do work rather well. Therefore, for many small protocols, running the protocol under one of these algorithms and under exact state representations will result in *no* omissions: the model checker will print exactly the same number of visited states. However, for very large protocols, these algorithms do report fewer visited states than reported by an exact search, thereby confirming that state omissions can, indeed, be observed in real life. Our experiments are largely geared towards measuring this discrepancy: *is POD hashing able to report numbers closer to that reported by exact search?* The answer turns out to be "yes." However, running many experiments using these very large protocols and comparing these runs against POD hashing runs will take a very long time (several months on large clusters). Therefore our results are reported with respect to runs obtained for smaller length signatures of anywhere from 2 to 10. In the range of 2 to 10 signature bits, we do observe that

- the precision of existing algorithms does improve with more bits, and
- the improvement due to POD-hashing does remain; although, for *given protocols*, clearly, the improvement does decrease, because existing algorithms are able to offer better precision.

However, one may argue that if more bits are offered to a protocol, they must be then evaluated using bigger protocols. As we said already, our experimental limitations are unable to, at present, go beyond 10 bits. We do not foresee any discontinuities from 10 to 40 bits; we do not report these results because existing algorithms are able to lead to a very small number of omitted states, and turns out that 30 bits are always enough to have no omissions. Thus, there are too few (or no) states to be regained. Thus we assert, without proof:

> For a certain number of bits, and for protocols where traditional algorithms do cause the precision to drop, POD hashing is able to recover a significant amount of precision.

In our experiments so far, we found that a "look ahead" of 1 (i.e., only considering one successor of each state that is found to be on the visited states hash table) recovers a significant amount of the lost precision. Higher look-ahead has diminishing returns, and could increase run-times to unacceptable levels. With a lookahead of 1, search time increases by a factor of 1.8 on average.

The second observation to make is "so what?" In other words, are more bugs going to be caught by POD-hashing due to the additional precision it offers? Colin West has observed [15] that the same bug typically manifests in several states, because typical assertions do not depend on all the state vector bits. Intuitively, it is clear that this number can vary highly; we are not aware of many studies in this regard. In [16], we report our own study on one reasonably large example on one property that failed. We found that the bug manifested in about 7,900 states. Thus the real question seems to be: "what is the probability that all these 7,900 or so states are missed due to the reduced precision of existing algorithms?" In a model with one billion states, an omission probability of .00001 could miss all 7,900 of those error states (plus a few others). However, we do not further address questions along these lines because the next question may be "how do you know you have created the formal model correctly?" The bottom-line is that we would ideally like to omit fewer states, as we don't know where the bug is or how frequently the bug is manifested in the state space of a model. Given that there could be modeling errors anyhow, it seems natural to seek the highest precision possible during model checking and our experiments confirm that POD Hashing always gives higher precision.

Roadmap After a discussion of related work in Section 2, we present the Precision-On-Demand-Hashing model-checking algorithm in Section 3. Section 4 presents results of some experiments that demonstrate the efficacy of our approach with the data and further explanation contained in Appendix A. Section 5 concludes.

2 Related Work

Due to the well-known *state explosion* problem, the two main structures of the depth first visit algorithm (a typical pseudocode is shown in Fig. 1), i.e. the search stack `S` and the hash table for visited states `T`, are likely to fill up all the available system memory resources. Many techniques have been developed to counteract the state explosion problem. Application of efficient disk swapping techniques make resource consumption by the search stack a less difficult problem than that of resource consumption by the hash table. Indeed, the hash table is likely to contain *all* the system reachable states, thus it is the amount of available memory that is the primary limitation to the number of states that can be stored and hence visited. The most effective and widely used solution to this problem till now consists in storing in `T` fixed-sized state *signatures*, instead of the states themselves. State signatures are much smaller than full state descriptors, thus allowing for a huge reduction of memory requirements for `T`.

Bit state hashing was proposed initially in 1987 by Gerard Holzmann [17] as a technique to increase the coverage of automated analysis of a system via model checking when the reachable state space is too large to fit in main memory. The main idea is to compress (via some hashing function) a state having a fixed number of bits into an index into some large table. The element at the index being a single bit of information representing the state.

Wolper and Leroy [18] extend this idea by proposing the use of multiple hashing functions, and corresponding bit tables, to represent a visited state by some small number of bits. In this way the probability of an omission due to collision is significantly reduced.

Stern and Dill [8, 19] improve on the scheme by storing a 40 bit *signature*. They also analyze the effect of using linear probing in combination with a universal class of hashing functions. Hash compaction is widely used because it has been shown that the *omission probability* is typically low; being much better than one can give for testing or simulation.

Bloom filters [20, 9] are often even better than hash compaction (especially for the preliminary analysis of the omission probability). However they have been applied only recently to model checking [9], so their use is still not widespread. Here a Bloom filter is used in place of a standard hash table. The signature is then applied to the filter to set appropriate bits in the filter. The state is present only when all of the corresponding bits, in this case the authors use three (3) bits per state, are set in the filter.

In each of the above works, a state is considered present when the identical signature is found to be represented in the visited state set, regardless of the representation.

These kind of techniques, having a state omission probability greater than zero, are often referred to as *probabilistic model checking* [9]. However, this term has been recently and more properly used to indicate Markov Chain verification [21]. Nevertheless, in this paper we will use the phrase "probabilistic model checking" to signify techniques possibly leading to state omissions, such as hash compaction and Bloom filters.

3 Precision-On-Demand Hashing for Model-Checking

In this section we present a depth-first search based model-checking algorithm that includes POD Hashing. First, the necessary definitions. A *Nondeterministic Finite State System* (shortened *NFSS* in the following) $\mathcal{S}$ is a 4-tuple $(S, I, \mathcal{A}, \mathtt{next})$, where S is a finite set of states, $I \subseteq S$ is the set of the initial states, $\mathcal{A}$ is a finite set of labels and $\mathtt{next} : S \to 2^{S \times \mathcal{A}}$ is a function taking a state s as argument and returning a set $\mathtt{next}(s)$ of pairs $(t, a) \in S \times \mathcal{A}$. We assume that $\mathtt{next}(s)$ is ordered for all $s \in S$.

Given an NFSS $\mathcal{S} = (S, I, \mathcal{A}, \mathtt{next})$ and a *safety property* ϕ defined on states (i.e., $\phi : S \to \{true, false\}$), we want to verify if ϕ holds on all the states of $\mathcal{S}$ (i.e., for all $s \in S$, $\phi(s)$ holds).

This can be done both by depth first (DF) and by breadth first (BF) visit. However, our methodology only applies to DF visit as previously discussed, so we will deal with DF visit only. The algorithm in Fig. 1 shows a typical explicit DF visit verifying if a given $\mathcal{S}$ satisfies a given ϕ. We assume that the visit in Fig. 1 always take into consideration the order of $\mathtt{next}(s)$, i.e. the successor state of a state s are expanded following the order given by $\mathtt{next}(s)$.

Our goal here is to improve state coverage of probabilistic model checking techniques by lowering the number of omitted states. To this aim, we take advantage of an invariant property of DF visits, stated in Prop. 1.

Proposition 1. *The DFS algorithm guarantees that when a state s is* revisited*, the* first *successor*[3] *(if any) of s would be a visited state. A state is said to be 'visited' if it is present in the visited state set (*`T` *in Fig. 1).*

[3] Remember that we assume the set of successors of s to be *ordered* for any s.

```
LIFO_Stack S = ∅; /* DF stack */
HashTable T = ∅; /* for visited states */

/* Returns true iff φ holds in all the reachable states */
bool DFS(NFSS 𝒮, SafetyProperty φ)
{
 let 𝒮 = (S, I, 𝒜, next);
 foreach s in I { /* visit each initial state s */
  if (!IfNotVisitedCheckPush(s, φ))
   /* IfNotVisitedCheckPush returned false, thus s is
      an error state and s does not satisfy φ */
   return false;
  while (S ≠ ∅) { /* main loop */
   (s, i) = Top(S); /* this does not modify the stack */
   increment transition index on the top of the stack;
   if (|next(s)| <= i) {
    /* unexplored successors exist */
    (s_next, a) = i-th pair in next(s); /* next(s) is
                                           ordered */
    if (!IfNotVisitedCheckPush(s_next, φ))
     return false;
   }
   else /* all transitions from s have been expanded */
    Pop(S); /* thus s is removed from the stack */
 } /* while */ } /* foreach */
 /* error not found, 𝒮 satisfies φ */
 return true;
} /* BFS() */

/* returns false if s is an error state (i.e. does not
   satisfy φ), true otherwise */
bool IfNotVisitedCheckPush(state s, SafetyProperty φ)
{
 if (s is not in T) { /* s is a new state */
  if (!φ(s))
   return false;
  HashInsert(T, s);
  Push(S, (s, 1)); /* the transition index is initialized
                      to 1 */
 } /* otherwise s is already visited */
 return true;
} /* IfNotVisitedCheckPush() */
```

Fig. 1. Standard DF Visit

Note that Prop. 1 does not hold for BF visits. In fact, suppose that a state s is revisited in a BFS; it could happen that s is still in the consumption queue. In this case, none of its successors is in `T` (because s has not been *expanded* yet), thus Prop. 1 does not hold for s. Also note that Prop. 1 does not hold for other successors (if any) than the first successor. In fact, suppose that a state s is revisited in a DFS; it could happen that s is still in the consumption stack `S`. In this case, if we pick the n-th successor t of s, it could happen that s has been reached again, starting from its first visit, via one of the $n-1$ successors preceding t in $\texttt{next}(s)$. This can be avoided only picking always the first successor.

POD-hashing exploits Prop. 1 in order to lower the *omission probability* and regain some states that would be omitted with the standard techniques. The idea is simple: before declaring a state s as already visited, which could lead to an omission, we also check the *chain of its first successors*, up to a given maximum length. If there exist one state in the chain which is not visited, then by Prop. 1 we can infer that s is a *new* (i.e. non-visited) state. States such as s would be omitted by present-day methods. Omissions are less likely under POD-hashing since s *and* its chain first successors need to be previously visited states in order for POD-hashing to result in an omission.

POD-hashing is implemented simply by modifying function `IfNotVisitedCheckEnqueue` of Fig. 1 as shown in Fig. 2. Note that the user-supplied parameter `podh_i` limits the length of the successors chain (the precision demanded).

4 Experimental Results

To measure the effectiveness of our approach, we implemented a version of POD-hashing both within the Murphi verifier [22] (on the top of hash compaction) and in the 3Murphi verifier [10] (on the top of Bloom filters). We call the resulting verifiers PODMurphi and POD3Murphi, respectively [23]. Our experiments are geared to measure the improvements exactly, as we now discuss.

We first chose three medium-sized protocols p_1, p_2, p_3 (having between 10^6 and 10^7 states; recall our discussions on size limitations in Section 1) from the Murphi distribution, and we ran a *complete (i.e. non-probabilistic)* verification on them. This allows us to know the *exact* number of states $N(p_i)$ for each p_i. Next, we ran verifications with the standard compression based techniques (namely hash compaction or Bloom filters) as follows:

for hash compaction: the number of bits for the signature was varied from 2 up to 40, in increasing steps. As for the hash table, it is set to have $xN'(p_i)$ entries, where x varies from 1 to 3 with step .5, and $N'(p_i)$ denotes the next prime number after $N(p_i)$ (open addressing requires a prime number of entries in the hash table). In this way, we vary the settings from the least memory requirements to the wider ones;

for Bloom filters: we take advantage of a Bloom filters property, which allows to determine, given an estimation of the number of states and the amount of

```
bool IfNotVisitedCheckEnqueue(state s, SafetyProperty φ,
                              int podh_i)
{
 res = s is in T;
 res_next = true;
 if (res) {
  /* don't trust the signature, check first successors
     chain */
  s_tmp = s;
  count = 1;
  while (res_next && count <= podh_i) {
   if (|(s_next, a)| > 0) {
    (s_next, a) = first pair in next(s_tmp);
    res_next = s_next is in T;
    count = count + 1;
    s_tmp = s_next;
   }
   else
    /* s has no successors */
    res_next = false;
  } /* no more than podh_i while iterations */
 }
 if (!res || !res_next) {
  /* s is surely new */
  if (!φ(s))
   return false;
  HashInsert(T, s);
  Push(S, (s, 1));
 }
 /* otherwise it is assumed to be already visited (may
    still be an omission) */
 return true;
} /* IfNotVisitedCheckEnqueue() */
```

Fig. 2. PODH algorithm

available memory, which is the best number of bits for the Bloom filter [9]. We reverse this computation – obtaining the amount of memory given a number of bits and the estimation of the number of states. Note that we already now the latter to be exactly $N(p_i)$. However, for the sake of completeness, in our experiments we put the estimation of states to be $xN(p_i)$, for $x = 1, 1.5, 2, 2.5, 3$ as for the hash compaction case. In this way, we can again vary the number of bits as for the hash compaction case, and vary the Bloom filter size as well. This way, we are able to simulate the least memory requirements first, followed by increasing amounts of memory.

Finally, for each verification using hash compaction (resp., Bloom Filters), we also run 4 verifications with PODMurphi (resp., POD3Murphi). In these 4 verifications, the precision (i.e. the maximum length of the first successors chain, that is `podh_i` in Fig. 2) varies from 1 to 4; all the other options (number of bits for the signature/Bloom filter, memory a/o number of entries in the hash table) are the same. To keep the comparison fair we also fix the random values computed to generate signatures and use the same values for all model-checking runs.

Figs. 3, 4, 5 and 6 report the most meaningful results from our experiments. The values used to generate these graphics are an average on the 3 protocols we are considering; the full data may be found in App. A.

More in detail, Fig 3 (resp. 5) graphs the additional obtained state space coverage w.r.t. standard hash compaction (resp. Bloom filter). To this aim, the graphic has the number of entries in the hash table (resp. Bloom filter) on the x-axis (if the number is $xN(p_i)$, we simply report x), and $\frac{1}{3}\sum_{i=1}^{3}(N_{pod}(x,p_i)/N_h(x,p_i) - 1)$ (resp. $\frac{1}{3}\sum_{i=1}^{3}(N_{pod}(x,p_i)/N_b(x,p_i) - 1)$) on the y-axis, where:

- $N_{pod}(x,p_i)$ is the number of states of the precision-on-demand verification of p_i with $xN(p_i)$ entries in the hash table (resp. Bloom filter);
- $N_h(x,p_i)$ (resp. $N_b(x,p_i)$) is the number of states obtained with standard hash compaction (resp. Bloom filters) with with $xN(p_i)$ entries.

As for Fig. 4 (resp. 6), it graphs the *total* state space coverage obtained by the precision-on-demand algorithm on the top of hash compaction (resp. Bloom filter). To this aim, the x-axis is the same as in Fig. 3 (resp. 5), and $\frac{1}{3}\sum_{i=1}^{3} N_{pod}(x,p_i)/N(p_i)$ on the y-axis, where:

- $N_{pod}(x,p_i)$ is again the number of states of the precision-on-demand verification of p_i with $xN(p_i)$ entries in the hash table (resp. Bloom filter);
- $N(p_i)$ is the *exact* amount of reachable states for protocol p_i.

W.r.t. all the experiments we carried out, Figs. 3, 4, 5 and 6 only show a small (meaningful) part (see App. A for the complete data).

In fact, for our 3 protocols, existing algorithms (hash compaction and Bloom filters) are already able to avoid omissions (or to very few ones) when using 20 bits or more. Since here we are interested in measuring our approach in a setting

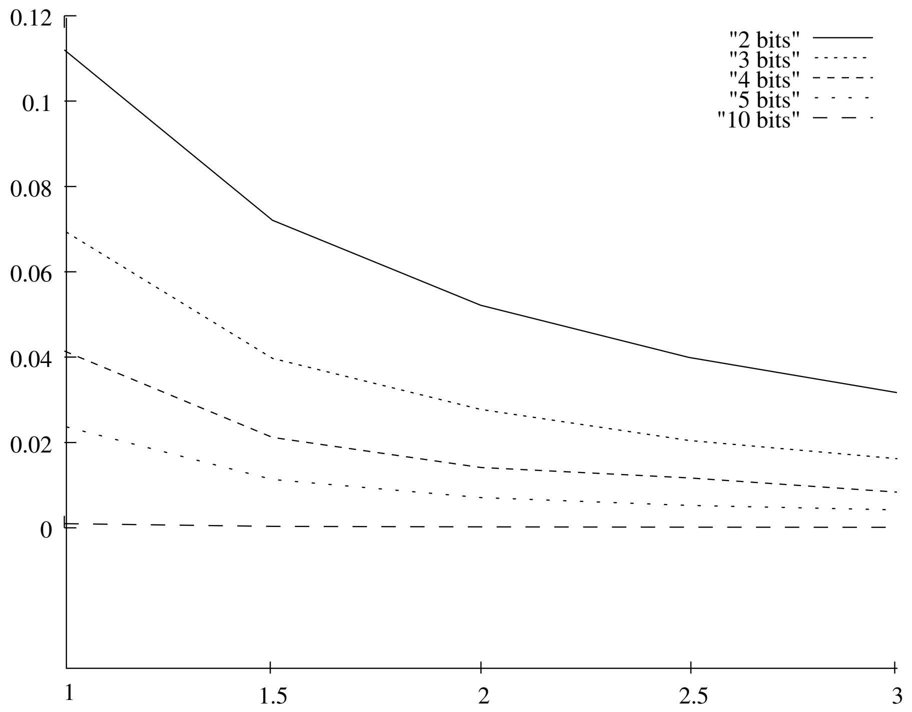

Fig. 3. Additional states for POD hash compaction (precision 1) w.r.t. standard hash compaction

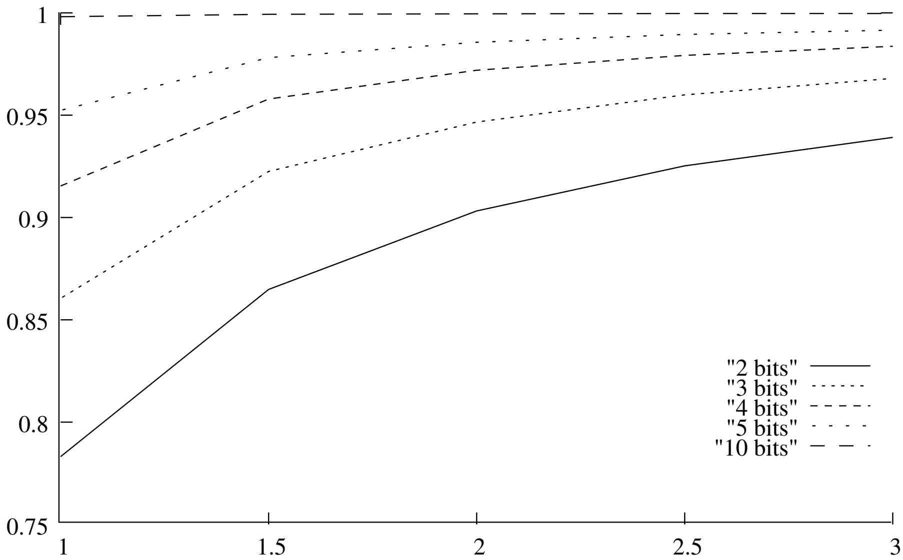

Fig. 4. POD Hash compaction (precision 1) total coverage

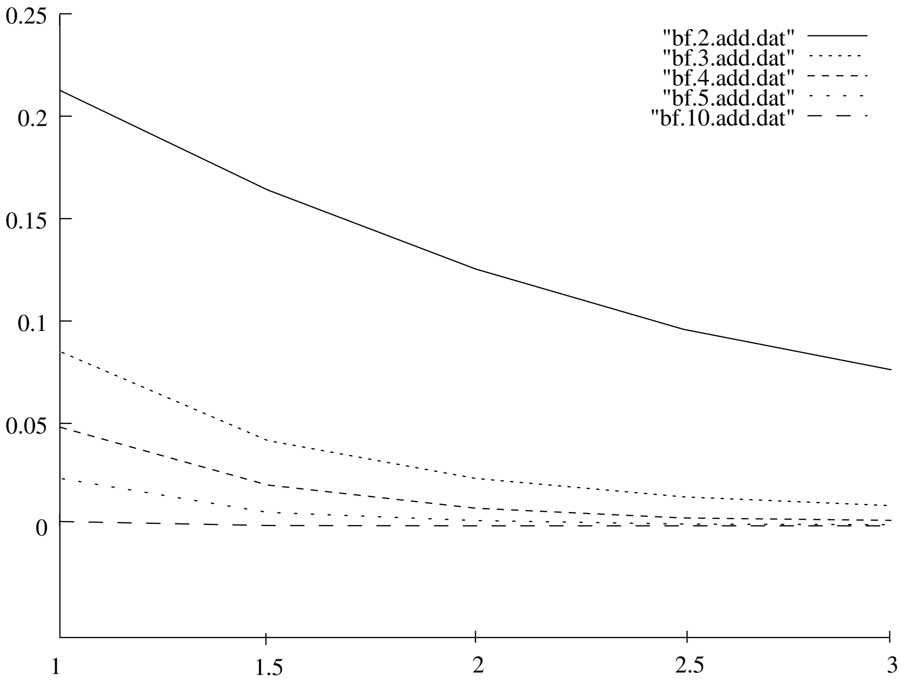

Fig. 5. Additional states for POD Bloom filters (precision 1) w.r.t. Bloom filters

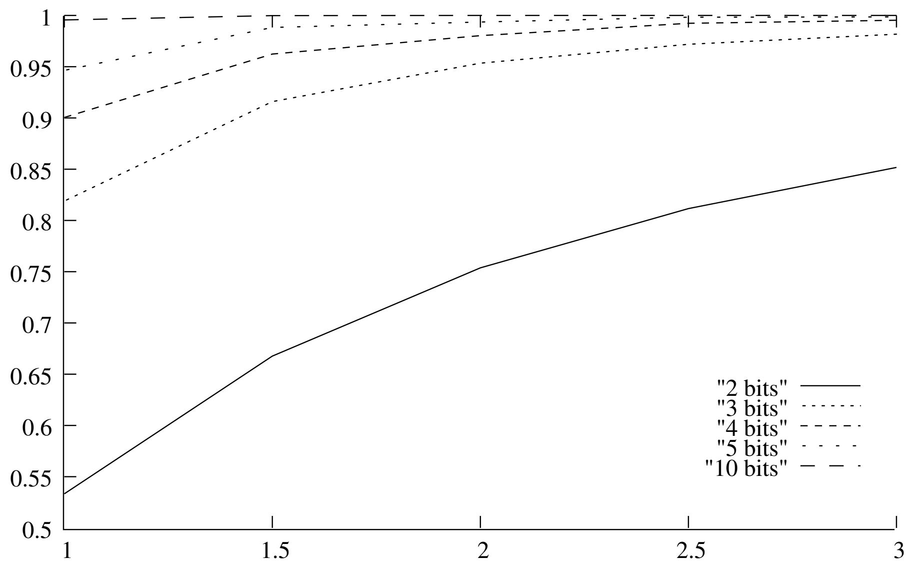

Fig. 6. POD Bloom filters (precision 1) total coverage

where they may be really useful, i.e. where omissions do happen, we limit our graphics to 10 bits.

Moreover, all the data in Figs. 3, 4, 5 and 6 are relative to precision (i.e. `podh_i` in Fig. 2) 1, since in this case we observe an acceptable time overhead of 80%. If we consider precision 2, coverage has small improvements, mainly due to protocols not being large enough, and time overhead is more than 200%; for precision 3 or higher, time overhead is completely unacceptable (see App. A).

However, from Figs. 3, 4, 5 and 6 it is clear that precision-on-demand hashing may lead to meaningful state space coverage improvements with acceptable time overheads, especially when the available RAM memory is barely enough to fit the given state space, even if represented with a small number of bits (up to 5). This can be obtained with small values of the precision - higher values lead to unacceptable time overheads.

Finally, note that the gaits of the graphics are as naturally expected. In Figs. 3, and 5, we have that the percentage of states "regained" by the precision-on-demand approach decreases when the hash table/Bloom filter increases, thus gaining more reliability; for the same reason, the percentage is higher for small values of the number of bits. On the other hand, Figs. 4, and 6 the total coverage naturally increases when enlarging the hash table/Bloom filter, since the underlying hash compaction/Bloom filter algorithm improves its reliability; for the same reason, the higher the value of the number of bits, the higher the percentage of states regained.

5 Concluding Remarks

We have presented a technique that increases the precision of probabilistic model checking that is orthogonal to the compressed state representation. We have implemented this algorithm in Murphi and 3Murphi for use with both the hash compaction state signatures and bloom filter visited state set representation. We have shown experimentally that Precision-On-Demand-Hashing may lead to meaningful state space coverage with acceptable time overhead. This is particularly evident when the available RAM is barely enough to fit the given state space. As errors are discrete and may be exhibited in a small fraction of the overall state space this increase in coverage is of significant value.

Our experiments validate our intuitions that more precision is in fact possible using existing probabilistic representations, however much work remains. Future work includes a rigorous mathematical analysis of the algorithm proposed in this paper. We also plan to explore heuristics to reduce the overhead of POD Hashing. We also plan to apply POD Hashing to models that are intractable under currently available memory to further validate the approach.

We have placed the code for our implementation of POD Hashing in a Murphi distribution that is available from our web page at:
`http://www.cs.utah.edu/formal_verification/software/murphi/murphi.POD/`.

References

1. Ching-Tsun Chou, Phanindra K. Mannava, and Seungjoon Park. A simple method for parameterized verification of cache coherence protocols. In *Formal Methods in Computer-Aided Design*, volume 3112 of *LNCS*, pages 382–398. Springer, 2004.
2. Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model checking programs. In *The Fifteenth IEEE International Conference on Automated Software Engineering (ASE'00)*, September 2000.
3. Stephen F. Siegel and George S. Avrunin. Verification of mpi-based software for scientific computation. In *Proceedings of the 11th International SPIN Workshop on Model Checking Software*, volume 2989 of *LNCS*, pages 286–303, Barcelona, April 2004. Springer.
4. Robert Palmer, Steven Barrus, Yu Yang, Ganesh Gopalakrishnan, and Robert M. Kirby. Gauss: A framework for verifying scientific computing software. In *SoftMC: Workshop on Software Model Checking*, number 953 in ENTCS, August 2005.
5. Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, December 1999.
6. T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. Zing: Exploiting program structure for model checking concurrent software. In *CONCUR 2004: Fifteenth International Conference on Concurrency Theory, London, U.K., September 2004*, LNCS. Springer-Verlag, 2004. Invited paper.
7. 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.
8. U. Stern and D.L. Dill. Improved Probabilistic Verification by Hash Compaction. In P.E. Camurati and H. Eveking, editors, *Correct Hardware Design and Verification Methods*, volume 987, pages 206–224, Stanford University, USA, 1995. Springer-Verlag.

9. Peter C. Dillinger and Panagiotis Manolios. Bloom filters in probabilistic verification. In Alan J. Hu and Andrew K. Martin, editors, *Formal Methods in Computer-Aided Design, 5th International Confrence, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004, Proceedings*, volume 3312 of *Lecture Notes in Computer Science*, pages 367–381. Springer, 2004.
10. 3Murphi distribution http://www-static.cc.gatech.edu/~peterd/3murphi/.
11. Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Pearson Education, Inc., 2002.
12. SPIN distribution http://www.spinroot.com.
13. D. L. Dill. The mur*phi* verification system. In *Proc. of CAV 1996*, volume 1102 of *Lecture Notes in Computer Science*, pages 390–393. Springer, 1996.
14. Madanlal Musuvathi, David Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. In *Operating Systems Design and Implementation*, dec 2002.
15. C. H. West. Protocol validation in complex systems. In *SIGCOMM '89: Symposium proceedings on Communications architectures & protocols*, pages 303–312, New York, NY, USA, 1989. ACM Press.
16. H. Sivaraj and G. Gopalakrishnan. Random walk based heuristic algorithms for distributed memory model checking. In *Proc. of PDMC 2003*, volume 89 issue 1 of *Electronic Notes in Theoretical Computer Science*, pages 51–67. Elsevier, 2003.
17. G.J. Holzmann. On limits and possibilities of automated protocol analysis. In H. Rudin and C. West, editors, *Proc. 6th Int. Conf on Protocol Specification, Testing, and Verification, INWG/IFIP*, Zurich, Sw., June 1987.
18. P. Wolper and D. Leroy. Reliable hashing without collision detection. In *5th International Conference on Computer Aided Verification*, pages 59–70, 1993.
19. U. Stern and D. L. Dill. A new scheme for memory-efficient probabilistic verification. In Reinhard Gotzhein and Jan Bredereke, editors, *IFIP TC6/WG6.1 Joint International Conference on: Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification*, volume 69 of *IFIP Conference Proceedings*, pages 333–348. Kluwer, 1996.
20. Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.
21. M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with PRISM: A hybrid approach. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002, Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, volume 2280 of *Lecture Notes in Computer Science*, pages 52–66. Springer, 2002.
22. Murphi distribution http://sprout.stanford.edu/dill/murphi.html.
23. PODMurphi and POD3Murphi distributions http://www.cs.utah.edu/formal_verification/software/murphi/murphi.POD/.

A Complete experimental results

In the following we show our complete experimental results. Tabs. 1, 2 and 3 show the results for the comparison with hash compaction, while Tabs. 4 5, and 6 are for the comparison with Bloom filters.

The meaning of the columns in Tabs. 1, 2, 3, 4, 5, and 6 is explained in the following:

		entries: 1.0			entries: 1.5		
Bits	**Pod**	$\|S_p\|/\|S\|$	$\|S_p\|/\|S_h\|-1$	t_p/t_h-1	$\|S_p\|/\|S\|$	$\|S_p\|/\|S_h\|-1$	t_p/t_h-1
2	0	0.703292	0	0	0.806334	0	0
2	1	0.783104	0.112023	1.015	0.864817	0.0720884	0.926916
2	2	0.795138	0.128485	2.19311	0.869918	0.0781937	2.26011
2	3	0.797432	0.131589	3.54318	0.870596	0.0790005	3.34186
2	4	0.797992	0.132345	4.64147	0.870705	0.0791309	4.31679
3	0	0.804049	0	0	0.887094	0	0
3	1	0.86048	0.0695989	0.909043	0.922467	0.0396999	0.826081
3	2	0.866496	0.0768605	1.82222	0.924359	0.0417842	1.65245
3	3	0.867462	0.0780169	3.20052	0.924551	0.0419955	3.07973
3	4	0.867565	0.0781413	3.53191	0.924574	0.0420214	3.34833
4	0	0.878763	0	0	0.937941	0	0
4	1	0.915374	0.0414452	1.00496	0.957874	0.0211959	1.14104
4	2	0.918074	0.0444541	1.68947	0.958505	0.0218606	1.85249
4	3	0.918363	0.0447749	2.70919	0.958541	0.0218976	2.48586
4	4	0.918394	0.0448093	3.44634	0.958543	0.0219003	3.30862
5	0	0.930181	0	0	0.967121	0	0
5	1	0.952361	0.0237765	0.811676	0.978088	0.0113272	0.782245
5	2	0.953395	0.024873	2.05643	0.978253	0.0114971	1.98282
5	3	0.953464	0.0249467	2.7676	0.978261	0.0115049	2.3752
5	4	0.953467	0.0249496	3.87569	0.978261	0.0115052	4.15267
10	0	0.997188	0	0	0.998934	0	0
10	1	0.998153	0.000967622	0.739666	0.99929	0.000355843	0.761308
10	2	0.998156	0.000970738	1.49267	0.99929	0.000355843	1.55173
10	3	0.998156	0.000970738	2.63475	0.99929	0.000355843	3.27359
10	4	0.998156	0.000970738	3.20584	0.99929	0.000355843	3.27383
20	0	0.999998	0	0	1	0	0
20	1	0.999998	5.9076e-07	0.873612	1	1.96393e-07	0.977131
20	2	0.999998	5.9076e-07	1.98169	1	1.96393e-07	2.02334
20	3	0.999998	5.9076e-07	2.53129	1	1.96393e-07	2.49458
20	4	0.999998	5.9076e-07	3.0901	1	1.96393e-07	3.17944
30	0	1	0	0	1	0	0
30	1	1	0	0.731591	1	0	0.762373
30	2	1	0	1.70497	1	0	1.60129
30	3	1	0	2.27051	1	0	2.68197
30	4	1	0	3.79598	1	0	3.5519
40	0	1	0	0	1	0	0
40	1	1	0	0.75266	1	0	0.67726
40	2	1	0	1.7396	1	0	1.66186
40	3	1	0	2.86081	1	0	2.77338
40	4	1	0	3.47984	1	0	3.24856

Table 1. Comparison of Precision-on-Demand Hashing with Hash Compaction (1)

		entries: 2.0			entries: 2.5		
Bits	**Pod**	$\|S_p\|/\|S\|$	$\|S_p\|/\|S_h\|-1$	t_p/t_h-1	$\|S_p\|/\|S\|$	$\|S_p\|/\|S_h\|-1$	t_p/t_h-1
2	0	0.858276	0	0	0.889598	0	0
2	1	0.903199	0.05218	0.876686	0.925225	0.039956	0.849561
2	2	0.905869	0.0551982	2.15998	0.926941	0.0418431	2.04032
2	3	0.906184	0.0555536	3.18225	0.927075	0.0419897	3.14655
2	4	0.906217	0.0555905	4.22778	0.927094	0.0420109	3.38347
3	0	0.921071	0	0	0.940594	0	0
3	1	0.946671	0.0277363	0.839033	0.959892	0.0204752	0.818773
3	2	0.947682	0.0288182	1.68564	0.960491	0.0211075	1.6663
3	3	0.947736	0.0288767	3.02155	0.960518	0.0211348	2.68144
3	4	0.947739	0.0288797	3.35265	0.960526	0.0211432	3.29945
4	0	0.958399	0	0	0.967942	0	0
4	1	0.971977	0.0141431	0.940244	0.979245	0.0116827	0.881217
4	2	0.972252	0.0144272	2.07166	0.979387	0.0118279	2.04393
4	3	0.972256	0.0144324	2.69367	0.979391	0.0118315	2.55873
4	4	0.972257	0.0144327	3.25018	0.979389	0.0118298	3.2385
5	0	0.978649	0	0	0.984268	0	0
5	1	0.985602	0.00709713	0.781074	0.989442	0.00525285	0.745969
5	2	0.985691	0.00718819	1.56853	0.989482	0.00529335	1.53342
5	3	0.985692	0.00718875	2.36557	0.989482	0.0052938	2.43516
5	4	0.985692	0.00718875	3.54728	0.989482	0.00529396	3.29191
10	0	0.999291	0	0	0.999536	0	0
10	1	0.999532	0.000240896	0.736728	0.999685	0.000149222	0.692695
10	2	0.999532	0.000240896	1.51674	0.999685	0.000149248	1.44668
10	3	0.999532	0.000240896	2.95448	0.999685	0.000149248	2.76894
10	4	0.999532	0.000240896	3.14308	0.999685	0.000149248	3.01792
20	0	0.999999	0	0	1	0	0
20	1	1	2.34677e-07	0.999357	1	2.60752e-08	0.759468
20	2	1	2.34677e-07	2.01603	1	2.60752e-08	1.85639
20	3	1	2.34677e-07	2.79974	1	2.60752e-08	2.84422
20	4	1	2.34677e-07	3.1633	1	2.60752e-08	3.00291
30	0	1	0	0	1	0	0
30	1	1	0	0.808604	1	0	0.773071
30	2	1	0	1.8136	1	0	1.90854
30	3	1	0	2.93716	1	0	2.94077
30	4	1	0	3.53742	1	0	3.5305
40	0	1	0	0	1	0	0
40	1	1	0	0.673548	1	0	0.796757
40	2	1	0	1.62326	1	0	1.46002
40	3	1	0	2.73104	1	0	2.85476
40	4	1	0	3.44992	1	0	3.14522

Table 2. Comparison of Precision-on-Demand Hashing with Hash Compaction (2)

		entries: 3.0		
Bits	**Pod**	$\lvert S_p\rvert/\lvert S\rvert$	$\lvert S_p\rvert/\lvert S_h\rvert - 1$	$t_p/t_h - 1$
2	0	0.910214	0	0
2	1	0.939126	0.0316714	0.834185
2	2	0.94032	0.0329601	1.68886
2	3	0.940369	0.0330124	2.83719
2	4	0.940382	0.0330262	3.33393
3	0	0.952533	0	0
3	1	0.968001	0.0162082	0.813883
3	2	0.968358	0.0165794	1.63708
3	3	0.968382	0.0166045	2.73781
3	4	0.968382	0.0166048	3.27964
4	0	0.975472	0	0
4	1	0.983644	0.00836972	0.986545
4	2	0.983737	0.00846424	2.02297
4	3	0.983739	0.00846605	2.48981
4	4	0.98374	0.00846678	3.27215
5	0	0.987439	0	0
5	1	0.991592	0.0042031	0.667249
5	2	0.991616	0.00422678	1.40931
5	3	0.991611	0.00422235	2.927
5	4	0.991611	0.00422235	3.27835
10	0	0.999608	0	0
10	1	0.99972	0.00011156	0.953296
10	2	0.99972	0.00011156	1.85133
10	3	0.99972	0.00011156	2.7558
10	4	0.99972	0.00011156	2.95363
20	0	1	0	0
20	1	1	6.19209e-08	0.895715
20	2	1	6.19209e-08	1.8668
20	3	1	6.19209e-08	2.63149
20	4	1	6.19209e-08	3.03621
30	0	1	0	0
30	1	1	0	0.734951
30	2	1	0	1.85589
30	3	1	0	2.86061
30	4	1	0	3.37047
40	0	1	0	0
40	1	1	0	0.859386
40	2	1	0	1.55097
40	3	1	0	3.0275
40	4	1	0	3.52515

Table 3. Comparison of Precision-on-Demand Hashing with Hash Compaction (3)

		entries: 1.0			entries: 1.5		
Bits	**Pod**	$\|S_p\|/\|S\|$	$\|S_p\|/\|S_h\|-1$	t_p/t_h-1	$\|S_p\|/\|S\|$	$\|S_p\|/\|S_h\|-1$	t_p/t_h-1
2	0	0.438143	0	0	0.572188	0	0
2	1	0.53334	0.212646	1.20936	0.667714	0.163982	1.18678
2	2	0.562415	0.2757	2.36827	0.688856	0.199214	2.16163
2	3	0.572801	0.298144	3.49464	0.695005	0.209341	3.15827
2	4	0.577416	0.308009	5.19132	0.697137	0.212861	4.45959
3	0	0.754046	0	0	0.879117	0	0
3	1	0.818918	0.0851844	0.92227	0.915896	0.0416548	0.767852
3	2	0.827769	0.0965707	1.83692	0.91843	0.0444737	1.61534
3	3	0.829371	0.0985943	2.71971	0.918667	0.0447367	2.56281
3	4	0.829694	0.0989982	3.61386	0.918703	0.0447763	3.17761
4	0	0.858871	0	0	0.943548	0	0
4	1	0.900472	0.0482113	0.874019	0.962353	0.0199226	0.874174
4	2	0.904401	0.0527332	1.69701	0.963308	0.0209344	1.6143
4	3	0.904917	0.0533265	2.87222	0.963395	0.0210259	2.76474
4	4	0.904972	0.0533899	3.37262	0.9634	0.021031	3.22122
5	0	0.925045	0	0	0.981854	0	0
5	1	0.946483	0.0232112	0.803756	0.988303	0.00656701	0.902938
5	2	0.947405	0.024189	1.68442	0.988382	0.00664743	1.58054
5	3	0.947456	0.0242422	2.4105	0.988384	0.00664914	2.34536
5	4	0.947459	0.0242458	3.60145	0.988384	0.00664914	3.24667
10	0	0.993524	0	0	0.999611	0	0
10	1	0.995631	0.00212191	0.743393	0.999769	0.000157698	0.748728
10	2	0.995647	0.00213793	1.56647	0.999769	0.000157698	1.6114
10	3	0.995647	0.00213796	2.30424	0.999769	0.000157698	2.34014
10	4	0.995647	0.00213796	3.09751	0.999769	0.000157698	3.10352
20	0	0.999974	0	0	1	0	0
20	1	0.999982	9.18992e-06	0.789389	1	0	0.738838
20	2	0.999982	9.18992e-06	1.50385	1	0	1.6043
20	3	0.999982	9.18992e-06	2.51549	1	0	2.57089
20	4	0.999982	9.18992e-06	3.09331	1	0	3.05422
30	0	0.999999	0	0	1	0	0
30	1	0.999999	3.65053e-07	0.603338	1	0	0.71637
30	2	0.999999	3.65053e-07	1.32936	1	0	1.48357
30	3	0.999999	3.65053e-07	2.07767	1	0	2.28913
30	4	0.999999	3.65053e-07	2.81281	1	0	3.05678
40	0	1	0	0	1	0	0
40	1	1	2.60752e-08	0.727988	1	0	0.740912
40	2	1	2.60752e-08	1.51601	1	0	1.52486
40	3	1	2.60752e-08	2.26129	1	0	2.32947
40	4	1	2.60752e-08	3.28018	1	0	3.09854

Table 4. Comparison of Precision-on-Demand Hashing with Bloom Filters (1)

		entries: 2.0			entries: 2.5		
Bits	**Pod**	$\|S_p\|/\|S\|$	$\|S_p\|/\|S_h\|-1$	t_p/t_h-1	$\|S_p\|/\|S\|$	$\|S_p\|/\|S_h\|-1$	t_p/t_h-1
2	0	0.668632	0	0	0.739739	0	0
2	1	0.753685	0.125382	1.00679	0.811539	0.0959362	0.953835
2	2	0.768075	0.145993	1.99478	0.821253	0.108635	1.89413
2	3	0.771276	0.150538	2.90655	0.822969	0.110852	2.80648
2	4	0.772206	0.151852	4.32951	0.823368	0.111366	4.01024
3	0	0.931837	0	0	0.958353	0	0
3	1	0.953438	0.0231222	0.755483	0.971895	0.0141057	0.76853
3	2	0.954383	0.0241253	1.57541	0.972257	0.0144811	1.60172
3	3	0.954424	0.0241683	2.34387	0.972267	0.0144913	2.36475
3	4	0.954428	0.0241725	3.29088	0.972268	0.0144918	3.18483
4	0	0.972027	0	0	0.988558	0	0
4	1	0.980363	0.00858171	0.781614	0.992339	0.00382321	0.784588
4	2	0.98054	0.00876388	1.54939	0.992371	0.0038557	1.64133
4	3	0.980544	0.00876805	2.78596	0.992372	0.00385649	2.67791
4	4	0.980544	0.00876805	3.36565	0.992372	0.00385649	3.41322
5	0	0.991065	0	0	0.997403	0	0
5	1	0.993656	0.00261591	0.756606	0.998289	0.00088833	0.777048
5	2	0.993665	0.00262505	1.58453	0.998291	0.000891171	1.54864
5	3	0.993665	0.00262505	2.31823	0.998291	0.000891171	2.31146
5	4	0.993665	0.00262505	3.1294	0.998291	0.000891171	3.14673
10	0	0.999939	0	0	0.999995	0	0
10	1	0.999958	1.9428e-05	0.724544	0.999996	1.37628e-06	0.757414
10	2	0.999958	1.9428e-05	1.49093	0.999996	1.37628e-06	1.51457
10	3	0.999958	1.9428e-05	2.51567	0.999996	1.37628e-06	2.6795
10	4	0.999958	1.9428e-05	3.01134	0.999996	1.37628e-06	3.10818
20	0	1	0	0	1	0	0
20	1	1	0	0.717159	1	0	0.74054
20	2	1	0	1.4755	1	0	1.4802
20	3	1	0	2.55631	1	0	2.57719
20	4	1	0	3.03625	1	0	3.05374
30	0	1	0	0	1	0	0
30	1	1	0	0.698274	1	0	0.73348
30	2	1	0	1.49096	1	0	1.49391
30	3	1	0	2.19317	1	0	2.22515
30	4	1	0	3.76203	1	0	3.43162
40	0	1	0	0	1	0	0
40	1	1	0	0.744917	1	0	0.741504
40	2	1	0	1.50784	1	0	1.53266
40	3	1	0	2.34085	1	0	2.21374
40	4	1	0	3.07698	1	0	3.00522

Table 5. Comparison of Precision-on-Demand Hashing with Bloom Filters (2)

		entries: 3.0		
Bits	**Pod**	$\|S_p\|/\|S\|$	$\|\|S_p\|/\|S_h\|-1$	t_p/t_h-1
2	0	0.790984	0	0
2	1	0.851715	0.0761798	0.903338
2	2	0.858164	0.0840734	1.81475
2	3	0.859052	0.0851526	2.64127
2	4	0.859216	0.0853513	4.09065
3	0	0.972073	0	0
3	1	0.981652	0.0098517	0.767619
3	2	0.981816	0.0100193	1.57453
3	3	0.981819	0.0100225	2.32147
3	4	0.981819	0.0100225	3.19358
4	0	0.992619	0	0
4	1	0.995274	0.00267536	0.775379
4	2	0.995293	0.00269353	1.54261
4	3	0.995293	0.00269363	2.68069
4	4	0.995293	0.00269363	3.30014
5	0	0.99808	0	0
5	1	0.998645	0.000566235	0.771267
5	2	0.998646	0.000566405	1.6054
5	3	0.998646	0.000566405	2.31176
5	4	0.998646	0.000566405	3.12056
10	0	0.999996	0	0
10	1	0.999997	2.87627e-07	0.762402
10	2	0.999997	2.87627e-07	1.515
10	3	0.999997	2.87627e-07	2.57783
10	4	0.999997	2.87627e-07	3.11525
20	0	1	0	0
20	1	1	0	0.73925
20	2	1	0	1.4727
20	3	1	0	2.31244
20	4	1	0	3.04259
30	0	1	0	0
30	1	1	0	0.754957
30	2	1	0	1.51936
30	3	1	0	2.67128
30	4	1	0	3.23514
40	0	1	0	0
40	1	1	0	0.760657
40	2	1	0	1.53989
40	3	1	0	2.26966
40	4	1	0	3.54911

Table 6. Comparison of Precision-on-Demand Hashing with Bloom Filters (3)

`Bits`**:** number of bits for the signature/Bloom filter;

`Pod`**:** precision for the Precision-On-Demand algorithm, that is `podh_i` in Fig. 2;

$|S_{pod}|/|S|$**:** number of states computed with the corresponding precision and number of bits, divided by the total number of states (without omissions). The values in the table are an average between the 3 protocols we chose. More in detail, the value in each entry is $\frac{1}{3}\sum_{i=1}^{3} N_{pod}(p_i)/N(p_i)$, where $N_{pod}(p_i)$ is the number of states of the precision-on-demand verification of p_i, made with the corresponding precision and number of bits. This entry gives the total obtained state space coverage;

$|S_{pod}|/|S_h| - 1$ **(resp.** $|S_{pod}|/|S_b| - 1$**):** number of states computed with the corresponding precision and number of bits, divided by the number of states obtained with hash compaction (resp. Bloom filters) with the same number of bits. Then, 1 is subtracted to give only the ratio of states which are *added* by the precision on demand algorithm. The values in the table are an average between the 3 protocols we chose. More in detail, the value in each entry is $\frac{1}{3}\sum_{i=1}^{3}(N_{pod}(p_i)/N_h(p_i) - 1)$ (resp. $\frac{1}{3}\sum_{i=1}^{3}(N_{pod}(p_i)/N_b(p_i) - 1)$), where $N_{pod}(p_i)$ is as described earlier, and $N_h(p_i)$ (resp. $N_b(p_i)$) is the number of states obtained with hash compaction (resp. Bloom filters) with the corresponding number of bits. This entry gives the additional obtained state space coverage w.r.t. standard techniques;

$t_{pod}/t_h - 1$ **(resp.** $t_{pod}/t_b - 1$**):** computation time observed with the corresponding precision and number of bits, divided by the computation time obtained with hash compaction (resp. Bloom filters) with the same number of bits. Then, 1 is subtracted to give only the *time overhead* required by the precision on demand algorithm. The values in the table are an average between the 3 protocols we chose. More in detail, the value in each entry is $\frac{1}{3}\sum_{i=1}^{3}(t_{pod}(p_i)/t_h(p_i) - 1)$ (resp. $\frac{1}{3}\sum_{i=1}^{3}(t_{pod}(p_i)/t_b(p_i) - 1)$), where $t_{pod}(p_i)$ is the computation time of the precision-on-demand verification of p_i, made with the corresponding precision and number of bits, and $t_h(p_i)$ (resp. $t_b(p_i)$) is the computation time obtained with hash compaction (resp. Bloom filters) with the corresponding number of bits. This entry gives the computational time overhead w.r.t. standard techniques.