# Compressed Set Representations and their Effectiveness in Probabilistic Model Checking

Marck van der Vegt
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
m.e.m.vandervegt@student.utwente.nl

## ABSTRACT

Model checking is a technique employed in many areas such as the design of safety critical systems. Designers of such systems can construct models, which can give insight into the behavior of the system when verified by using model checking algorithms. One type of information that could be gained is reachability information (Will the system ever fail?). Model checking does not come without any challenges however: The state space explosion is a well-known phenomenon that plagues many techniques such as model checking [1]. This means that so called exhaustive model checking algorithms have to keep track of very large amounts of states. Most of these algorithms use sets of these states, which can take up a significant amount of memory. This research investigates the effectiveness of using compressed representations of sets in reducing the amount of memory required by these algorithms, and their impact on performance. We find that these set representations are capable of reducing the memory usage of the model checking algorithms by 74% on average, though at quite a high performance hit. Due to the relatively small impact of bit sets on the total memory usage of model checking, these alternative set representations may have limited applicability.

## Keywords

Model checking, Memory Usage, Bit Set, Compression

## 1. INTRODUCTION

The behavior of a finite system can be verified by using a model checking algorithm. This allows designers of such systems to create a model of the system, and be certain that a given bad property will never hold, or that a good property will always hold. Depending on how accurate the models are, these models can give insight into their system counterpart. Because these models are checked by a computer, even complicated systems can be modeled and checked, which would otherwise be impossible to do by hand. By means of probabilistic model checking, even systems that depend on random events (such as network protocols that have to deal with packet loss) can be verified to be working as intended.

Many of these model checking algorithms depend on using sets, to indicate that some property holds for the elements of that set. Due to the so-called state space explosion problem, the state space generated by a model can easily exceed millions or even billions of elements [2]. Because of this, even relatively memory efficient set representations that require only 1 bit per element (so called bit sets), can take up significant amounts of memory. This in turn has an impact on performance, as the increase in memory usage can increase the amount of cache misses.

The goal of this research is to investigate the possibility of using compressed set representations in these exhaustive model checking algorithms. These sets would use less memory, which in turn means that larger models can be checked before running out of memory. The effectiveness of compression largely depends on the compressibility of the input data, however. Furthermore, the performance of the algorithm will likely degrade, because of the added steps of compressing and decompressing elements of the set. This research will give an analysis of the effectiveness of these compressed set representations in reducing the amount of memory required and their impact on performance. Additionally, the parallelization opportunities of these data structures will be analyzed.

The main goal of this research will be to answer the following questions:

**RQ1** To what extent can a compressed set representation reduce the memory usage of model checking algorithms and what impact does this have on performance?

**RQ1.1** How well suited are these compressed sets representations for parallelization?

## 1.1 Related Work

The state space explosion is a well studied problem, meaning that extensive research has been done in reducing the amount of memory used by these model checking algorithms. This includes research aimed at improving the algorithms themselves as well as research aimed at improving the underlying data structures, such as sets. One area of research is the reduction of the amount of states required for model checking and another is the reduction of memory required per state.

### *State Compression and Reduction*

Reducing the amount of memory required per state can be performed by reusing parts of the state that also appear in different states [3]. A set of values that is the same across different states will be replaced by a pointer to a single copy of those values, to avoid storing the same set of values multiple times. Even though the research tackles

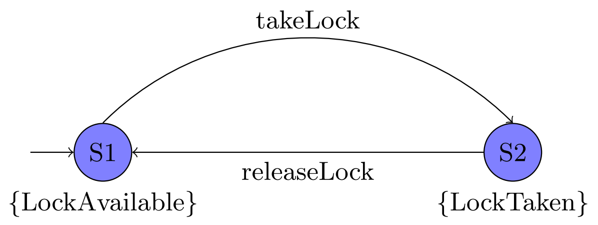

Figure 1: Basic example of a LTS: a simple lock

the same problem, it does so by focusing on a different aspect than our research.

### *PTrie*

Jensen et al. [4] describe their creation called PTrie, capable of reducing memory usage by $60 - 70\%$ while only degrading performance by 3% on average in their domain of Petri net model checking. Because these results are for a different domain, the effectiveness for this research's domain (LTS, MC and MDP model checking) will have to be measured.

## 2. BACKGROUND

We will now give some context information about the topics to be discussed in this paper.

### 2.1 Model Checking

When using model checking, the designer of a system creates a mathematical model of a system using a modeling language. Once this model is created, the model can be used in model checking algorithms to verify properties of interest. If the model was sufficiently accurate, the results of the verification of the model can give information about the original system.

An example for using model checking is a safety critical system. A complex system such as a nuclear reactor might have a large amount of states it could be in. Model checking can then be used to verify that under any circumstances the reactor cannot reach an undesirable state, which could be catastrophic for such a system.

There are several techniques for model checking, but this research will only go into Labeled Transition Systems (LTS), Markov chains and Markov Decision Processes (MDP). We will give a very basic and simplified overview of these systems.

#### *Labeled Transition System.*

An LTS is similar to Finite State Machine (FSM) in the sense that it consists of a collection of states, which are interconnected by means of transitions requiring an action. These actions are inputs to the model and can be seen as outside influences such as human interactions (e.g. button presses). Some differences between FSMs and LTSs are that LTSs can have infinite states, and that these states can have labels (which consists of so-called atomic propositions). Starting with an initial state, a trace is a singular sequence of actions, indicating a singular execution of the an LTS. Figure 1 shows a very basic example of an LTS: a model for a lock. In this example LockAvailable and LockTaken are atomic propositions.

#### *Markov Chain.*

A Markov chain is similar to a LTS, but the transitions are made by means of random choice: each transition has a probability associated with it, indicating how likely that particular transition is. This allows modeling systems that have random behavior, such as packet loss in network protocols. Unlike LTSs, Markov chains do not take actions, instead, transitions are performed purely based on the associated probabilities.

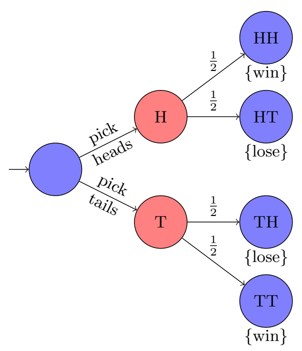

Figure 2: Basic example of an MDP: a coin flipping bet

#### *Markov Decision Process.*

An MDP is similar to a Markov chain, but each transition requires an action, just like with LTS. The current action determines what transitions are available for the current state, and the probabilistic distribution for taking those transitions. Figure 2 shows an example MDP model of a coin flipping bet. First the action is chosen (picking either tails or heads), which then results in a 50/50 chance to win.

### 2.2 State Space Explosion Problem

Systems are rarely constructed monolithically, instead they are constructed by combining smaller components [1], which can be done by using parallel composition. One of the problems with exhaustive model checking is that the amount of states to be checked can grow exponentially in the amount of these smaller components. Another extension that eases the modeling process is the addition of variables. Though these additions improve the expressiveness of modeling languages, they can have scaling problems however.

#### *Parallel Composition.*

When using parallel composition, two models are combined together into one model which encompasses the behavior of both models running side-by-side. This is useful for modeling systems like network protocols, where each participant of the communication abides to the same protocol, but acts independently. The problem that arises however is that to represent the parallel composition of two models, both containing $n$ states, requires $n^2$ states in the worst case. The parallel composition of three models would require $n^3$ states, continuing to scale exponentially in the number of models: $n^m$ (where $n$ is the amount of states per model and $m$ is the amount of models).

#### *Variables.*

Another method of easing the modeling process is the addition of variables. Realistic systems typically have a large number of possible configurations that would need to be explicitly represented as variables in the form of opaque states [5]. These variables can be updated and checked for just like in a normal programming language. The usage of variables can greatly increase the number of potential states however, as each state needs to be duplicated the same number of times as the domain size of the used variables. This means that adding a 8-bit counter to a model could potentially increase the amount of states in the state space by a factor of 256.

### *Scaling.*

Both these additions (parallel composition and variables) increase the expressiveness of modeling languages, but also allow relatively small models to have very large state spaces (state space explosion). The state space can become so large, that exhaustive model checking is no longer viable: exploring every state would require either too much memory or time.

## 2.3 Compressed Set Representations

A bit set is a representation for a set. Internally a bit set is a long array of bits, storing a single bit for each element in the domain of the values to be stored. When a bit in this array is set to 0, the corresponding element *is not* a member of the set, while if it set to 1, the element *is* a member of the set. Finally, a one-to-one function maps an element to be inserted/retrieved from the set to an index in the bit array so that arbitrary elements can be stored in the set.

One way to reduce the effect of the state space explosion is by using a compressed set. Similar elements can be grouped, or relevant portions of the set can be decompressed when required. This reduces memory usage and can potentially also increase performance due to better CPU cache usage.

### *Run Length Encoding (RLE) in Bit Sets.*

RLE can be used to compress bit sets by detecting long runs of bits having the same value and packing them together. This packing is performed by storing the index at which the run stars and the length of the run. When using the notation $\langle start, length\rangle$, the bit string 111001111 becomes $\langle 0, 3\rangle$ $\langle 5, 4\rangle$.

Though RLE can be very effective at compressing certain kinds of data (i.e. containing long runs of 0's and 1's), it becomes less useful when used 'dynamically' (i.e. requiring frequent insertions or deletions). Retrieving data at a certain index might require traversing a large portion of the runs [6]. Furthermore, changing a bit in the very beginning might mean that whole sequence has to be moved to the right, to make room for the new bit to be inserted.

### *Trie.*

A different way of storing a bit set is by using a trie (also known as a prefix tree). Although tries were traditionally used for optimizing string searches, tries can also be used for bit strings. A trie works by recursively grouping elements with the same prefix together. Informally, the words *worker*, *worked*, *works* would be stored as: *work* + {*er*, *ed*, *s*}, reusing the common prefix *work* three times. Similarly, the bit strings $000100, 000110, 000011$ can be stored as $000 + \{100, 110, 011\}$

### *Binary Decision Diagram (BDD).*

A bit set can be seen as a Boolean function: an element is the input to the function, and the function returns whether the element is a member of the set. A BDD represents this Boolean function as an acyclic directed graph. Traversing this graph, using the bits in the input to decide what edges to follow yields the associated output. Though BDDs are memory efficient, they can have an unacceptable impact on performance [7].

## 2.4 Integer Sets

The model checking algorithms make use of integer sets. Instead of storing all the states for which a certain property holds, an index is stored which corresponds to a state. This saves having to store the states multiples times, or computing the hash of a state in the case of a hash based set. An integer set has the usual set operations, such as insert, delete and union. A difference with a normal set however, is that the elements of a integer set are totally ordered, meaning that normal set operations can also be applied to ranges of elements. A formal description of the integer set operations is described below.

$$\begin{aligned}
\mathit{Insert}(S, i) &= S \cup \{i\} \\
\mathit{Remove}(S, i) &= S \setminus \{i\} \\
\mathit{Contains}(S, i) &= i \in S \\
\mathit{RangeApply}(A, B, r) &= \\
\{i \,|\, (i \in r \wedge i \in B) &\vee (i \notin r \wedge i \in A)\} \\
\mathit{RangeUnion}(A, B, r) &= \mathit{RangeApply}(A, A \cup B, r) \\
\mathit{RangeIntersect}(A, B, r) &= \mathit{RangeApply}(A, A \cap B, r) \\
\mathit{RangeComplement}(S, r) &= \mathit{RangeApply}(S, \bar{S}, r)
\end{aligned}$$

Where $S$, $A$ and $B$ are integer sets, $r$ is an integer range, and $i$ an index.

## 3. SET USAGE

To ensure that the set representations are effective, it is important to analyze the usage of these sets in the model checking algorithms. After all, each set representation will have their own strengths and weaknesses, so for high efficiency, these will have to match up with the set usage by the model checking algorithms.

For example, a bit set using RLE will have bad performance when it comes to reading/writing at random indices, because such a set would require iterating from already known position until the requested index is found (similar to a linked list). Now this might sound like a major drawback to such a set, but it all depends on the way the set is accessed. If the set were used in such a manner that the indices requested are strictly increasing, there would be no need for backtracking, reducing the overhead.

## 3.1 Algorithms

To calculate properties of interest, several stages can be identified. We will only focus on two stages: the precomputation and value iteration stage. These two stages both make use of sets, but in different ways. We will now describe the two stages to be able to analyze their usage of the sets.

### *Precomputations.*

During the precomputation stage, the state space is preprocessed in such a way that it makes the upcoming stages easier. The Modest Toolset[1] makes use of the method described by [8], by calculating which states are definitely going to reach a target state, and those states which are definitely *not* going to reach a target state. The information about these states can then be used in the value iteration stage.

[1]Model checking toolset used in this paper, more information on this in Section 5.1

By logging every read and write operation to the sets used by this algorithm in an experiment, we were able to determine that this algorithm performs read and write operations in 'bursts', which means that a run of write operations would be followed by a run of read operations. Using this information, it could be more beneficial to store this burst of write operations, and only add them to the set once the first read operation comes in.

#### *Value Iteration.*

For calculating the probability of reaching a target state, the Modest Toolset has multiple options, though we will only focus on value iteration. Value iteration is an approach, which calculates the probability of reaching a target state in a finite $n$ steps. Because these steps are finite, the final calculated probability is only an approximate answer. This does not mean it is inaccurate however, as using more and more steps gives a more accurate answer, allowing for arbitrary precision.

### 3.2 Effectiveness of Precomputations

The precomputation stage is meant as a way of speeding up the actual value iteration stage, but can take significant time to perform. It is therefore only useful to perform these precomputations if the speedup gained is greater than than the time it took to perform these precomputations.

To check whether these precomputations are actually worth performing, a small test was performed to investigate the effectiveness of these precomputations. The results can be seen in Table 1. Fastest times are underlined. $t$ indicates time spent on value iteration in seconds while $p$ indicates time spent on precomputations in seconds. $S_0$ and $S_1$ indicate whether that particular precomputation was performed. Some properties require precomputations to be performed regardless of the chosen precomputation, which is why the 'None' column has non-zero entries for $p$.

The table shows that indeed, precomputations speed up the value iteration stage, though they can take up significant amounts of time. Because the precomputation make use of several bit sets, they will be turned on for evaluating the effectiveness of the alternative set representations.

## 4. SET REPRESENTATIONS

The following set representations will be tested:

| **Representation** | **Implementation** |
|---|---|
| Uncompressed bit set | Existing Modest Toolset implementation |
| Prefix tree | PTrie |
| RLE + index list + uncompressed hybrid | CRoaring.Net |
| Hash set | C# .NET implementation `HashSet<int>` |

These set representations have been chosen because they allow read and write operations directly on their representation. A higher compression ratio could probably be achieved by using a more sophisticated compression algorithm such as DEFLATE, though this would have an unacceptable impact on performance, as every read/write operation would require decompression/compression respectively.

### 4.1 Uncompressed Bit Set

In this form, the set is stored as a long bit string, where each element of the bit string corresponds to a single state in the state space. A 1 in that bit string indicates that the corresponding state is a member of the set, while a 0 indicates the opposite.

#### *Memory Usage.*

Because of the way these uncompressed bit set are stored, they always uses the same amount of memory, regardless of the amount of elements contained within the set. The memory usage in bytes of a set in this representation can be calculated by dividing the amounts of bits stored by 8.

#### *Time Complexity.*

The operations insert, delete and contains, all have time complexity $O(1)$, because they are simple read/write operations on the bit string. Though this scales well, the operations union and intersect have time complexity $O(|D|)$ (where $D$ is the domain of the indices), because every bit has to be checked in order to perform these operations. In the worst case, two completely empty sets get intersected. Even though the result is trivial (the empty set), it would still require accessing every single bit of both sets.

#### *Parallelization.*

Using a single read/write-lock for a big uncompressed bit set will likely have a big impact on the performance. If two threads are writing/reading at opposite ends of the bit set, there should be no need for locking out either thread. Instead, the bit set can be divided into chunks, each having their own read/write-lock. This will prevent threads from accessing the same data at the same time, while not having to create a large amount of locks, which would be a waste of memory.

### 4.2 PTrie

PTrie is a set implementation that is capable of storing arbitrary sized elements, without using any hashing function, while being on par with other optimized set implementations that are hash based[9].

#### *Memory Usage.*

Due to the way PTrie is designed, the last byte of all entries is always stored in a bucket, which means that for every entry added to the PTrie, at least 1 byte is always required. Next, the tree structure which leads to those leaf nodes strongly depends on the kind of data that was put into the PTrie. Indices with similar prefixes will yield a small tree, while indices with differing prefixes will result in a larger tree. Due to the usage of many (64 bit) pointers in the non-leaf nodes, the actual tree structure itself can potentially end up consuming quite a lot of memory. Accurately calculating the memory usage of a PTrie can be performed by keeping track of all allocations and deallocations. This is done by replacing all calls to memory management functions in the PTrie implementation by ones that also keep track of the amount of bytes used by the set. Each call to an allocation function increases the memory usage counter, and each call to a deallocations function decreases the counter. This means that this counter always indicates the current memory used by the PTrie (assuming the PTrie implementation has no memory leaks).

#### *Time Complexity.*

Table 1: Test results for the effectiveness of precomputations

| Model info | | Precomputation info | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | None | | $S_0$ | | $S_1$ | | $S_0 + S_1$ | |
| Model | States | $t$ | $p$ | $t$ | $p$ | $t$ | $p$ | $t$ | $p$ |
| beb4 N=5 | 3,492,162 | 8.60 | 0.00 | 9.05 | 0.65 | 31.36 | 23.46 | 31.08 | 23.42 |
| consensus6 K=2 | 1,258,240 | 167.89 | 2.39 | 153.60 | 2.98 | 202.57 | 71.08 | 203.16 | 71.19 |
| csma4-2 | 761,962 | 8.58 | 2.40 | 8.82 | 2.61 | 24.30 | 18.70 | 23.49 | 17.91 |
| dpm6-10-4 | 131,314 | 6.30 | 0.10 | 6.17 | 0.12 | 5.73 | 0.24 | 5.66 | 0.28 |
| echoring100 | 871,634 | 38.58 | 0.00 | 30.84 | 0.36 | 67.84 | 33.56 | 64.50 | 33.82 |
| elevatorsb-11-9 | 538,326 | 7.72 | 0.00 | 8.16 | 0.24 | 4.75 | 1.07 | 4.96 | 1.28 |
| zeroconf-1000-4-false | 306,585 | 2.29 | 0.00 | 2.35 | 0.16 | 10.142 | 7.873 | 9.86 | 7.64 |

The time complexity was not provided by its creators. It can however be seen as being similar to a self balancing search tree: we would expect the basic operations to have $O(\log n)$ time complexity, provided that sometimes some extra operations are required to keep the tree in an efficient state. Because the indices store in the PTrie always have the same amount of bits, the tree will always have the same depth, leading to a time complexity for the basic operators to be $O(1)$ instead.

Sadly no specialized intersect or union operations were provided in the PTrie paper, which means that the naive implementations (using only the basic operations) leads to a time complexity of $O(|D|)$.

### *Parallelization.*

Making a PTrie thread-safe can be quite tricky, as a single delete or insert can drastically change the tree structure. Putting a read/write-lock on each non-leaf node might not work as that would be prone to deadlocking. If two insertion operations might require changing the tree structure owned by each other, resulting in a deadlock. Instead, a single read/write-lock can be used for the whole PTrie, though this will likely degrade performance heavily.

## 4.3 RLE

RLE can be used to store a whole bit set, but this is not tested in this paper. Instead, it is used as a chunk type of the Roaring format.

### *Memory Usage.*

The memory usage of run length encoding bit set strongly depends on the amount of runs present in the set, and the format that is used to store the runs. In the worst case, an alternating bit pattern would require 16 times the amount of memory compared to a uncompressed bit set when using the format that is used in Roaring, which uses 16 bits for the starting index and 16 bits for the run length. A mostly empty or mostly bit set would require only a fraction the amount of memory of a uncompressed bit set. The memory usage of a run length encoded bit set can accurately be calculated by multiplying the amount of runs with the memory usage of a single run.

### *Time Complexity.*

The time complexity for a run length encoding based bit set is similar to that of a linked list: for each access of the bit set, iteration is required from an already known position. In the worst case this would require iterating trough the whole list of runs just to access a single index. This means that the complexity for inserting, deleting and contains is all $O(r)$ (where $r$ is the amount of runs). The same hold for union and intersection, each run will be iterated over yielding yet again $O(r)$. This shows that the run length encoding can be quite efficient, but only if the data contains very long runs. A pattern which alternates would ruin both the memory usage as well as the performance.

### *Parallelization.*

A run length encoded bit set becomes really difficult to parallelize, because making a change in the beginning of the list of runs can affect everything that comes after that. One way to solve this however, is to divide the domain of the indices up in chunks and put a lock on that instead.

## 4.4 Roaring

Roaring is a hybrid technique for storing compressed bit sets, and due to its good performance, it has been adopted by several production platforms (e.g. Apache Lucene, Apache Spark, Apache Kylin an Druid)[6]. Instead of applying a single compression technique to the while bit set, the elements are partitioned in chunks of $2^{16}$ (65536) bits, which can have different compression techniques applied to them.

### *Memory Usage.*

As of the writing of this paper, there are 3 different chunk types in Roaring: uncompressed, sorted index array and run length encoding. To accurately calculate the memory used by a Roaring bit set, the bit set is serialized according to the Roaring specification [10]. Because this serialization is really similar to the memory used by the set itself, the byte count of the serialization is approximately equal to the memory usage of the set.

### *Time Complexity.*

The uncompressed chunk is similar to the previously mentioned uncompressed bit set, causing little overhead over that representation. Next, the run length encoding chunk functions similar to previously mentioned run length encoding bit set. The sorted index array chunk works by storing the (16 bit) indices of the states that are a member of the set. This allows for more efficient storage of sparse chunks. A chunk which has only one member can be stored by only storing a single index of that member. This does have worse time complexity however. Looking up an index now has to be done in a sorted array, which can be done using a binary search which has time complexity $O(\log n)$. Insertions and deletions have the added extra cost of potentially having to move all elements from the index list one position the right/left, which has complexity $O(n)$. The union and intersection of two sets still has to go through all indices of both index lists, giving $O(\min\{|S_1|, |S_2|\})$ complexity for intersection and $O(\max\{|S_1|, |S_2|\})$ complexity for union.

*Parallelization.*

The Roaring format is already divided up in chunks, so it makes sense to use a read/write-lock for each chunk. Making a change to such a chunk could affect the whole chunk in the case of RLE or index list, so a lock is definitely necessary to ensure thread safety.

## 4.5 HashSet

The reason bit sets are being used instead of more conventional set representations such as hash sets, is that they were expected to be more performant. To make sure that this is actually the case, the default C# HashSet is used in the tests.

*Memory Usage.*

The memory usage of a HashSet scales linearly with the amount of elements. To calculate the memory usage of a HashSet in C#, a C# technique called reflection can be used to determine the size of the internal components of the HashSet which are normally inaccessible.

*Time Complexity.*

The basic operations (insert, delete and contains) all have $O(1)$ time complexity. Because the elements of a HashSet have no particular order, the range based operations have $O(n)$ time complexity. In order to get the next element in the range, each index has to be checked.

*Parallelization.*

For parallelization, a ConcurrentDictionary from the C# standard library can be used. This provides fine-grained locking to ensure thread safety [11].

## 5. METHODOLOGY

## 5.1 Implementation

We will now describe the implementation details that are relevant to testing the set representations in probabilistic model checking. The code written for this paper can be found in the `marckvdv-compressed-sets` branch of the Modest Toolset repository.

*Modest Toolset*

The Modest Toolset is model checking toolset capable of checking the model types mentioned in Section 2.1, among others. The toolset currently makes use of uncompressed bit sets in several sections of the model checking algorithms. To test the effectiveness of the alternative set representations in the context of probabilistic model checking, the set representations mentioned in Section 4 have been implemented for the Modest Toolset. Because the toolset is written in C#, the set implementations used also had to be written in C#, or capable of interoperating with C#. For the Roaring bit set, an existing wrapper called CRoaring.Net is used. For PTrie (which only has a C++ implementation) we created a small C interface which could then be used in the toolset (using the C interoperation).
An obstacle in getting different set representations to work in the toolset, was the fact that the uncompressed bit set that was already present was tightly woven in to the code. This was necessary because special tricks were required to allow the toolset to allocate arrays greater than $2^{32}$ elements, which is normally the limit for C# arrays.

## 5.2 Test Setup

Table 2: Test machine specifications

| Specification | Value |
|---|---|
| Operating System | Windows 10 (Home edition version 1709) |
| CPU | Intel® Core$^{TM}$ i7-6700HQ @ 2.60GHz |
| Memory | 8 GiB SODIMM DDR3 @ 1600 MHz |
| Runtime | Microsoft .NET v4.7.2 |

We will now describe the setup that was used to test the performance and memory usage of the previously described set representations.

*QVBS Benchmark Models*

Because compression techniques can have different effectiveness based on the data type, it is important to test using a varied set of input data. The Quantitative Verification Benchmark Set (QVBS) contains a large number of models which can be used to test model checking software[12]. For testing the set representations we used only a small subset from QVBS, whereby each model is large enough such that time and memory measurements are also accurate. Any constant errors to the time or memory measurements will diminish once the model checking task becomes larger. For example, let's say that the memory measurement is always 100 bytes too low, then the relative error for a model checking task which requires only 2 hundred bytes is way higher than that of a task requiring 2 million bytes.

*Performance Measurement*

As indicated by the research question, we are interested in both the performance and the memory usage of the set representations. For measuring the performance, the Modest Toolset reports the time spent during the different phases of calculating the properties associated with the model. Determining the memory used by the various set implementations is more tricky however. The memory used by these sets contributes to only a small portion of the total memory used by the model checking algorithms. This, combined with the fact that C# is a garbage collected language means that the total memory used by the program is an inaccurate measurement. Instead, accurate memory usage is calculated by each set itself, using the method described in the 'Memory Usage' sections of 4. To not degrade the performance too much the memory usage of a set is only calculated once it is no longer being used.
All tests were performed on the same machine to make sure that the performance measurements are comparable. The specifications for the machine used for the benchmarks can be seen in Table 2.

## 6. RESULTS

The raw results of the tests performed can be seen in Table 3 and plotted in Figure 3, both in the appendix. We used a similar set of models as used in Table 1 to test the selected set representations. In this table, total time indicates the total time required to perform the model checking task, which includes time spent on stages which do not use make use of the new sets, such as state exploration. This in turn means that the total times cannot be compared as such, i.e. a total that is twice as small does not necessarily mean that the set representation is twice as fast. The 'Precomputation memory' concerns the total memory used by the set representations in the precomputation stage, while 'Atomic proposition memory' the total memory used by the sets representing the atomic propositions. Next, the total memory column is the sum of the previous two columns.

## 7. DISCUSSION

Regarding the results shown in Table 3, we will now evaluate the performance of the set representations one by one.

### *Uncompressed.*

This entry acts as a baseline: to improve the current usage of bit sets in the MODEST TOOLSET, the tested set representation should outperform the uncompressed bit set. The performance of the uncompressed bit set is clearly the highest, achieving a time which way lower than the other set representations for almost all the test cases. As explained in Section 4.1, the memory usage of an uncompressed bit set is constant, wasting memory when the set it represents is mostly empty, which is the case in the 'elevators' model.

### *Roaring.*

When comparing Roaring the uncompressed bit set, the Roaring bit set consistently uses less memory. Within the tested models, Roaring uses around 74% less memory on average, while increasing the total run time by 110%. This degraded performance was to be expected however, as most operations are insertions and removals, which is faster on uncompressed bit sets [13].

### *PTrie.*

In the models used in our tests, the PTrie implementation did not perform well. Even though PTrie is able to save some memory in some of the bit sets, there are also plenty of bit set which use way more memory than the uncompressed variant. In fact, when it comes to total memory usage, PTrie gets consistently outperformed by the uncompressed bit set. The creators of PTrie compare the performance of PTrie to that of conventional set implementations [9], and indeed, PTrie consistently outperforms HashSet when it comes to memory usage. Furthermore, the performance takes quite a hit: the total times of the PTrie implementation are more than ten times as high as that of the uncompressed bit set.

### *HashSet.*

The HashSet implementation performs well when it comes to performance, being slightly slower than the uncompressed bit set. The memory usage of this 'naive' method can be quite bad however, requiring more than 30 times the memory for some models when compared to the uncompressed bit set.

## 8. CONCLUSION

To conclude, alternative set representations are capable of reducing the memory usage in probabilistic model checking, though it comes at a high price when it comes to performance. In particular, the Roaring bitmap is capable of providing consistently lower memory usage, while having a relatively high impact on performance. Cutting out the interoperation with C and optimizing the library for this setting would likely decrease this impact on performance. Due to the bit sets contributing to only a small part of the total memory required by model checking, using these alternative set representations might not be worthwhile unless the models that are being checked become really large such that any reduction to the memory usage is favorable.

For the future, it would be interesting to see a Roaring bit map to be used more passively: using only uncompressed chunks while memory is available to achieve high performance, while using the compressed chunks either memory is running low or union and intersection operators are being used more often. This ensures that the compression is not a hindrance when it is not needed, while still being present when needed.

## 9. REFERENCES

[1] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking.* Springer, 2018.

[2] Ernst Moritz Hahn, Arnd Hartmanns, Christian Hensel, Michaela Klauck, Joachim Klein, Jan Kretínský, David Parker, Tim Quatmann, Enno Ruijters, and Marcel Steinmetz. The 2019 comparison of tools for the analysis of quantitative formal models - (QComp 2019 competition report). In *Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III*, pages 69–92, 2019.

[3] Petr Rockai, Vladimír Still, and Jiri Barnat. Techniques for memory-efficient model checking of C and C++ code. In *Software Engineering and Formal Methods - 13th International Conference, SEFM 2015, York, UK, September 7-11, 2015. Proceedings*, pages 268–282, 2015.

[4] Peter Gjøl Jensen, Kim Guldstrand Larsen, and Jirí Srba. PTrie: Data structure for compressing and storing sets via prefix sharing. In *Theoretical Aspects of Computing - ICTAC 2017 - 14th International Colloquium, Hanoi, Vietnam, October 23-27, 2017, Proceedings*, pages 248–265, 2017.

[5] Arnd Hartmanns. *On the analysis of stochastic timed systems.* PhD thesis, Saarland University, 2015.

[6] Daniel Lemire, Gregory Ssi Yan Kai, and Owen Kaser. Consistently faster and smaller compressed bitmaps with roaring. *Softw., Pract. Exper.*, 46(11):1547–1569, 2016.

[7] Peter Gjøl Jensen, Kim Guldstrand Larsen, Jirí Srba, Mathias Grund Sørensen, and Jakob Haahr Taankvist. Memory efficient data structures for explicit verification of timed systems. In *NASA Formal Methods - 6th International Symposium, NFM 2014, Houston, TX, USA, April 29 - May 1, 2014. Proceedings*, pages 307–312, 2014.

[8] V. Forejt, M. Kwiatkowska, G. Norman, and D. Parker. Automated verification techniques for probabilistic systems. In M. Bernardo and V. Issarny, editors, *Formal Methods for Eternal Networked Software Systems (SFM'11)*, volume 6659 of *LNCS*, pages 53–113. Springer, 2011.

[9] Peter Gjøl Jensen, Kim Guldstrand Larsen, and Jiří Srba. Ptrie: Data structure for compressing and storing sets via prefix sharing. In Dang Van Hung and Deepak Kapur, editors, *Theoretical Aspects of Computing – ICTAC 2017*, pages 248–265, Cham, 2017. Springer International Publishing.

[10] Roaringformatspec: specification of the compressed-bitmap roaring format. `https://github.com/RoaringBitmap/RoaringFormatSpec`.

[11] Concurrentdictionary.
`https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentdictionary-2?view=netframework-4.8`.

[12] Arnd Hartmanns, Michaela Klauck, David Parker, Tim Quatmann, and Enno Ruijters. The quantitative verification benchmark set. In *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part I*, pages 344–350, 2019.

[13] Samy Chambi, Daniel Lemire, Owen Kaser, and Robert Godin. Better bitmap performance with roaring bitmaps. *CoRR*, abs/1402.6407, 2014.

## APPENDIX

Table 3: Memory usage and performance of the tested set representations - raw data

| Model | States | Total time (s) | Precomputation memory | Atomic proposition memory | Total memory |
|---|---:|---:|---:|---:|---:|
| **Uncompressed** | | | | | |
| beb4 N=5 | 3,492,162 | 9.42 | 1,746,112 | 1,746,112 | 3,492,224 |
| consensus6 K=2 | 1,258,240 | 158.36 | 2,201,920 | 1,887,360 | 4,089,280 |
| csma4-2 | 761,962 | 10.08 | 1,333,472 | 1,428,720 | 2,762,192 |
| dpm6-10-4 | 131,314 | 6.43 | 196,992 | 164,160 | 361,152 |
| echoring100 | 871,634 | 33.93 | 1,524,440 | 3,050,880 | 4,575,320 |
| elevatorsb-11-9 | 538,326 | 8.22 | 134,592 | 67,296 | 201,888 |
| zeroconf-1000-4-false | 306,585 | 2.65 | 153,312 | 76,656 | 229,968 |
| **Roaring** | | | | | |
| beb4 N=5 | 3,492,162 | 20.78 | 859,275 | 167,616 | 1,026,891 |
| consensus6 K=2 | 1,258,240 | 286.97 | 41,359 | 6,684 | 48,043 |
| csma4-2 | 761,962 | 33.64 | 117,813 | 82,905 | 200,718 |
| dpm6-10-4 | 131,314 | 13.64 | 39,695 | 25,385 | 65,080 |
| echoring100 | 871,634 | 114.82 | 1,097,096 | 1,439,592 | 2,536,688 |
| elevatorsb-11-9 | 538,326 | 7.43 | 524 | 519 | 1,043 |
| zeroconf-1000-4-false | 306,585 | 3.99 | 138,736 | 21,144 | 159,880 |
| **PTrie** | | | | | |
| beb4 N=5 | 3,492,162 | 114.08 | 12,413,401 | 978,146 | 13,391,547 |
| consensus6 K=2 | 1,258,240 | 1502.44 | 23,378,927 | 12,240 | 23,391,167 |
| csma4-2 | 761,962 | 244.67 | 11,369,867 | 2,494,590 | 13,864,457 |
| dpm6-10-4 | 131,314 | 40.26 | 1,645,196 | 392,180 | 2,037,376 |
| echoring100 | 871,634 | 982.00 | 9,126,114 | 6,861,218 | 15,987,332 |
| elevatorsb-11-9 | 538,326 | 70.32 | 811,746 | 1,458 | 813,204 |
| zeroconf-1000-4-false | 306,585 | 19.08 | 822,354 | 42,936 | 865,290 |
| **HashSet** | | | | | |
| beb4 N=5 | 3,492,162 | 13.22 | 145,993,408 | 7,419,776 | 153,413,184 |
| consensus6 K=2 | 1,258,240 | 258.44 | 156,310,944 | 55,616 | 156,366,560 |
| csma4-2 | 761,962 | 23.21 | 530,715,146 | 26,281,200 | 556,996,346 |
| dpm6-10-4 | 131,314 | 8.01 | 16,182,336 | 2,909,600 | 19,091,936 |
| echoring100 | 871,634 | 71.62 | 91,880,384 | 62,002,304 | 153,882,688 |
| elevatorsb-11-9 | 538,326 | 10.06 | 10,772,128 | 6,896 | 10,779,024 |
| zeroconf-1000-4-false | 306,585 | 3.30 | 10,651,776 | 269,408 | 10,921,184 |

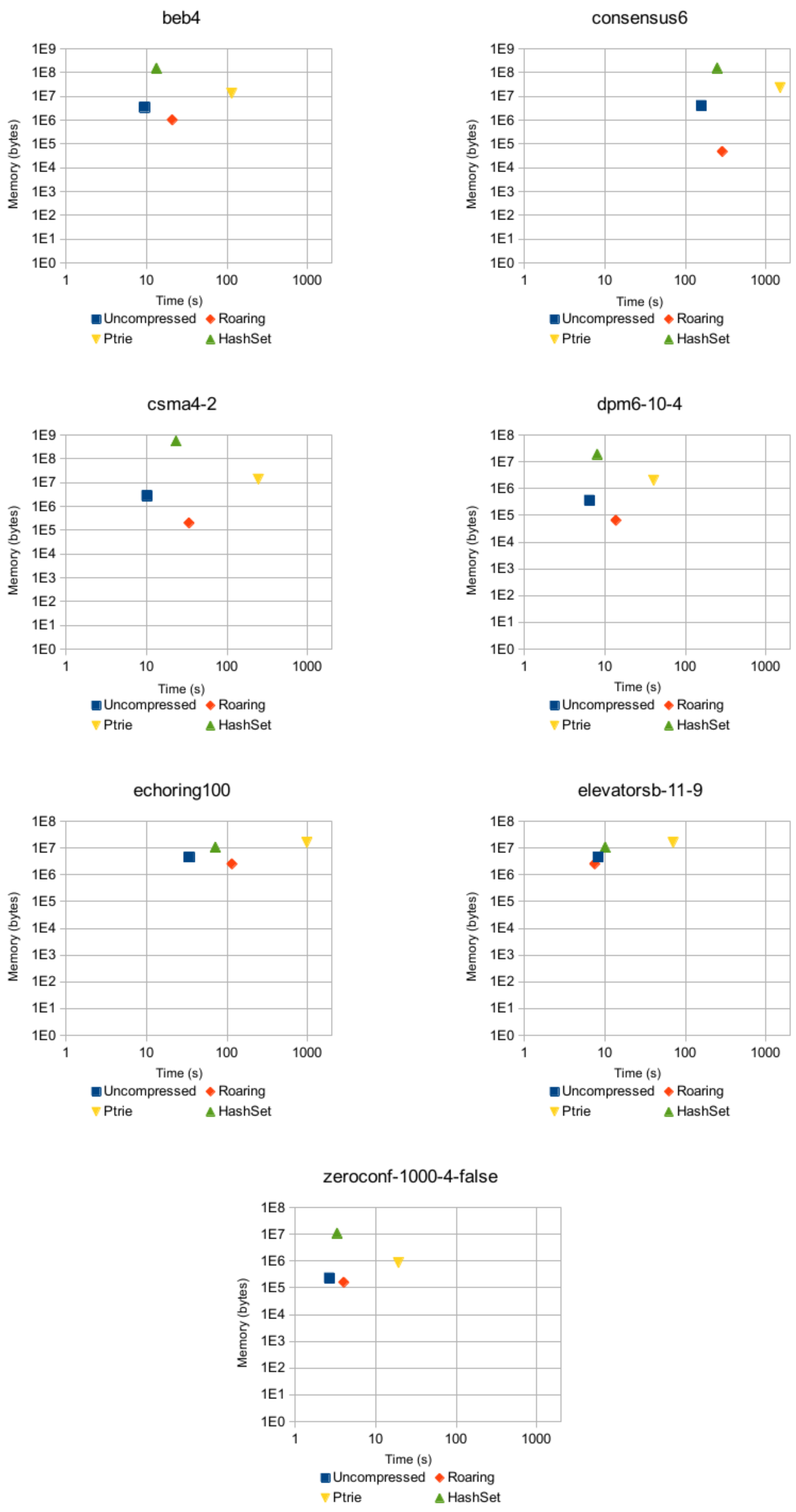

Figure 3: Graphs of all the models and the set representations