Bit-packing and Hashing Evaluation in Explicit-state Model Checking

VALENTIJN HOL, University of Twente, The Netherlands

MCSTA is a model checking tool for models written in the *Modest* language. It can be used to formally verify certain properties of a model. During model checking a model's state space needs to be (exhaustively) explored which takes a significant amount of time and memory – depending on model complexity. This research evaluates the impact of bit-packing state member variables by examining the execution time and memory consumption. We find that in all cases there is a decrease in memory usage, and in most cases a minor decrease in execution time. Furthermore, we propose an algorithm for selective field unpacking, to improve the performance of bit-packing, whilst maintaining near-optimal state sizes. Lastly, we evaluate the current hash function used in state space exploration based on hash-table bucket lengths, and find it to be performing adequately with little room for improvement.

Additional Key Words and Phrases: Modest, MCSTA, Bit-packing, Hashing, Optimisation, Modelling

1 INTRODUCTION

Software systems are incredibly prevalent in daily life. One can hardly go anywhere without running into some of them, whether they be public transport payment systems such as Oyster cards, or traffic lights control systems. All of these systems should behave predictably, and according to their specifications. There is therefore a need to verify that these systems behave as expected. Accordingly, several ways of ensuring these properties have sprung up, including but not limited to peer review, software testing, and lastly formal methods [2, pg. 3–4]. It is the latter of these three that is most infrequently used, but is nonetheless essential for safety critical systems such as medical technology or traffic control.

Formal methods are a way to establish system correctness with "mathematical rigor" [2, pg. 7]. This can be accomplished through the use of model checking. Model checking generally involves two steps, first is creating the model in a modelling language such as *Modest* [11, pg. 8–12], or in a tool like UPPAAL [19]. This model is an abstract representation of the system's behaviour, and describes the states a system could be in. Subsequently, one can use a model checking tool to verify whether certain properties hold (such as "The system does not deadlock" or "What is the probability that some event X happens").

To calculate the values of the aforementioned properties a model checking tool will need to explore all the states that the system could find itself in (with a few exceptions when using more clever approaches such as partial exploration). Unfortunately for real world systems the total state count can quickly explode. Even a simple model of exponential backoff can reach billions of states [8]. This so-called state-space explosion can cause model checking to run into real-world constraints of both memory and processor time.

TScIT 37, July 8, 2022, Enschede, The Netherlands

1.1 Problem Description

Given the above description it is evident that models can quickly become complex for real-world systems. This correspondingly involves a state-space explosion and a correlated increase in run-time and memory consumption for any model-checking tool.

It is therefore pertinent to examine what part of an existing model-checker (in this case MCSTA, part of the *Modest* Toolset [9]) can be optimised to improve both/either the run-time, and/or memory consumption. The primary step that can take an inordinate amount of time is state-space exploration. Here all states are (exhaustively) explored and mapped, to be used later for further analysis. In case of real-world models one can end up with several dozen million individual states. These states are collected in a hash-table for deduplication during future exploration steps.

This research is focused on evaluating the potential for improvement of two distinct areas relevant for state-space exploration. First, Section 2 will provide background information regarding MCSTA and the way models are compiled. Secondly, Section 3 will give an overview of the benchmark methodology used in both the hashing, and bit-packing sections. Section 4 contains an evaluation of the current hash function used in MCSTA, and a conclussion on the potential for improvement drawn. Lastly, an in-depth examination of the efficacy of bit-packing state variables is performed in Section 5. Both the run-time and memory consumption effects are examined.

2 BACKGROUND

MCSTA is an explicit-state model checker, and part of the wider Modest Toolset [9]. Explicit-state checkers work based on exploring individual states, and using algorithms that act thereupon. This is in contrast to symbolic model checking, where work is done on wider sets of (more abstract) states instead [3, 6]. During this section a brief overview of the main components relevant to this research is given.

2.1 Variables and State Layout

In *Modest* models, variables can be declared in either the global or process scope, where a process can be instantiated multiple times. There are several primitive and non-primitive types that can be used for variable declaration, but within the context of this research only Boolean and (bounded) integer primitives are relevant. An example of the two scopes in the *Modest* language can be seen in Listing 1.

The model will be parsed and compiled, where part of the compiled output is a `State` struct which keeps track of all variables and their values for a particular state instance. During the state space exploration a great many (depending on the complexity of the model) state instances will be created and inserted into a hash-set. For the model in Listing 1 the `State` struct would roughly look as seen in Listing 2.

2.2 Checker Operation

After a model is compiled it will be model checked. This involves two steps, the first of which is the state space exploration step. In

```
// Global scope variables
// Bounded integer in the inclusive range 0..2
int(0..2) clients;
bool is_success;
// What is the probability that
// some Host set is_success to true
property Success = Pmax(<> is_success);

process Host()
{
  // Process scoped variable,
  // multiple process instances can exist.
  int(0..3) address;

  if (!is_success) {
    {= is_success = true =}; stop
  } else {
    stop
  }
}

// Create two host instances.
par{ :: Host() :: Host()}
```

Listing 1. Trivial Modest model

```
struct State
{
  byte Host_address;
  byte Host_2_address;
  byte clients;
  bool is_success;
}
```

Listing 2. State struct layout generated for model in Listing 1

this step the declared processes are executed and for each transition in one such process a new state is generated. These states are saved in a hash-set to eliminate duplicate states, see subsubsection 4.1.1 for more. The second step is the property checking step, where all declared properties are evaluated based on the results from the state space exploration step. In the context of this research only the state space exploration step will be evaluated and improved upon.

As an example, imagine that the model in Listing 1 is executed. During the state space exploration three distinct states will be discovered; the initial state, the state where Host 1 sets *is_success*, and one where Host 2 does so instead. In the subsequent step the probability that *is_success* is set to *true* by some Host is calculated. In this case it is trivial to determine, both ending states have *is_success* = *true*, thus the probability would be 1.

3 BENCHMARK METHOD

This research will rely on an extensive benchmark suite in order to increase the confidence in the observed results. These benchmarks were sourced from the Quantitative Benchmark Set, which was created to facilitate the reliable benchmarking of formal verification tools and their relative improvements [12]. As this benchmark suite is quite diverse a selection was made to ensure the observed results were ran within some bounded time.

All models with at least one instance with a listed state space between 2 million and 140 million states were used. The lower bound was chosen to ensure changes in run-time would be measurable, and the upper bound was chosen for practical execution of the benchmarks. If a model could not be compiled, or crashed for a reason unrelated to this research, then it was excluded. Most models were ran with two to three instances of varying parameters, which resulted in different state space sizes within the same model. An instance is one combination of a model and a set of parameters. For example, we can have a model named `FMS`, and provide two different sets of parameters, namely {9} and {11}. This results in two instances. In total, the benchmark suite contains 58 instances.

These instances will be grouped by their model type for plotting. Each model is of a particular type, with 5 relevant for this research. Markov decision processes (MDP [16]), probabilistic timed automata (PTA [16]), discrete- and continuous time Markov chains (DTMC and CTMC [1]), and Markov automata (MA [5]).

All benchmarks were ran on the same machine, containing a Ryzen 5900X alongside 48 GB of DDR4 3200MT memory.

4 HASH TABLE

In this section we will evaluate the potential for improvement for the hash table implementation used in MCSTA. At present, MCSTA will insert every encountered state during state space exploration into a hash-set for deduplication and cycle-detection. To manage a large set of states MCSTA is equipped with the ability to selectively offload states unlikely to be used again to disk [10]. The insertion of a new state requires the computation of the hash-code of this state, as well as an additional equality check in case a state with the same hash code was found.

4.1 Background

A hash is a shortened representation of an object's state, expressed in a set amount of bits. Since hashes only have a finite number of bits some objects will map to the same hash as a different, unique object. This phenomenon is called a hash collision. These hashes are frequently used for data structures such as hash-sets, where an object's hash is used to calculate into which bucket it must be deposited. Buckets will frequently contain multiple objects, in part due to the hash collisions, and also due to the finite amount of buckets within a hash-table.

This leads to the problem of defining a hash function such that one gets something close to a uniform distribution of objects across the hash-table buckets. As shown by McKenzie et al [15], hash functions – when not chosen carefully – can easily produce non-uniform distributions. This would negatively affect the performance of the data structure. The degradation is most easily explained by imagining the buckets into which objects are deposited as linked lists, where the list is appended to every time a new state gets pushed into the same bucket. If the hash function used for this hash-table is non-uniform certain buckets will get a disproportionate amount of states, which all have to be looked at when accessing this bucket, and thereby reduce performance.

The use of the linked list mentioned above is one method of collision resolution called separate chaining. There are two classes of collision resolution, namely *Open Addressing* – where states are always kept within the hash-table and searched for with various methods – and *Separate Chaining* – where states can be stored in external data-structures in case of conflict [4, pg. 237]. Examples of the former are Linear Probing and Double Hashing, where the latter can use various different data-structures.

4.1.1 MCSTA Hash Function. Within the context of MCSTA each state will be hashed when it is inserted into the hash-table to check for duplicate states. This hash-table uses separate chaining as collision resolution. The hashing function is compiled alongside the state object, and is based on a simple multiply-add approach. It is defined as follows:

$$h_0 = 17, h_k = 486187739 * h_{k-1} + i_k, \text{for } k > 0 \quad (1)$$

where i_k denotes the k-th variable in a `State` instance.

MCSTA's hashing function had not yet been evaluated. If the hashing function was producing poor hash codes then this would significantly affect performance due to the duplication check mentioned before. Additionally, if there were a large amount of buckets with a notable (> 5) amount of states in them, then the collision resolution mechanism could be looked at for improvement.

4.1.2 DJB2 Hash Function. In order to evaluate the hash function currently used in MCSTA some point of reference would be useful. There are various hash functions one could pick, but for this evaluation it does not need to be an exhaustive list. This evaluation does not intend to find the best hash function, merely to verify whether the current hash function performs adequately. DJB2 was chosen by us for its simplicity of implementation, and acceptable performance in other papers [17]. Its approach is much the same as the one of MCSTA, but it uses different constants:

$$h_0 = 5381, h_k = 33 * h_{k-1} + i_k, \text{for } k > 0 \quad (2)$$

where i_k denotes the k-th variable in a `State` instance.

4.2 Experimental Design

To evaluate the current hashing function an analysis has been performed on the state of the hash table at the end of the state space exploration step. The benchmarks defined in Section 3 have been used. The primary evaluation is based on the average bucket length, calculated according to Equation 3.

$$\textit{Average Bucket Length} = \frac{\textit{Num. stored items}}{\textit{Num. non-empty buckets}} \quad (3)$$

4.3 Results, Discussion, Conclusion

In Figure 1 the average bucket length is plotted on the y-axis. The instance index is noted on the x-axis. As this is sorted based on increasing order of bucket length, the largest DJB2 instance is not necessarily the same as the largest Default instance. This nonetheless gives a good indication of how the hash functions perform over a wide variety of cases. In most instances the two hash functions are competitive, but at the extremes DJB2 can reach excessive bucket lengths. The interval of average buckets lengths for the default hash function is $[1.14, 2.89]$, whereas it is $[1.21, 239.10]$ for DJB2.

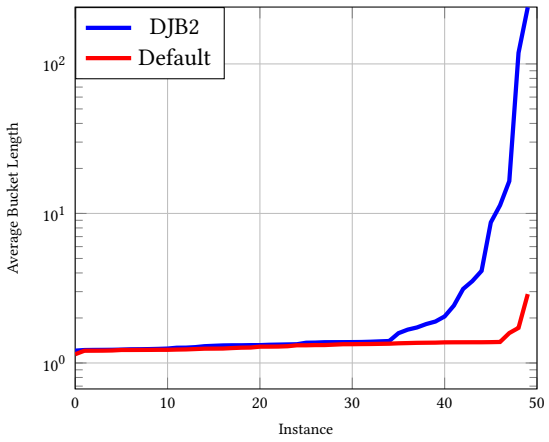
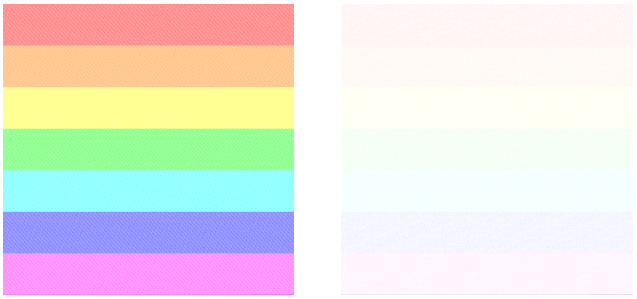


Fig. 1. Average bucket size, increasing order



(a) Default hashcode (b) DJB2 hashcode

Fig. 2. Hilbert map displaying state distribution across the hash buckets for the CLUSTER benchmark.

In Figure 2 two Hilbert maps are displayed, both based on the `Cluster` benchmark. This benchmark was one of the worse instances for DJB2 with an average bucket length of 16.45, compared to 1.33 for the default hash function. These maps give a general idea of how states might be spread out across the available buckets. Ideally there is no white space, as every colored pixel represents one bucket with one or more states inside. The colors of the pixels are irrelevant, and merely used to make sparse distributions easier to see. One can clearly see that the bucket distribution is cohesive when using the default hash function, but incredibly sparse in the case of the DJB2 hash function.

Based on this brief evaluation we can conclude that the current hash function performs well enough. The vast majority of benchmarks have average bucket lengths of near 1, with even the worst-performing benchmark only just over 2. As a consequence there is little reason to investigate further improvements when it comes to the hash function or collision resolution. Note that this is not to say there is no better hash function possible, as this evaluation was merely comparing DJB2 to the current default. There were several other hash functions which behave significantly worse, and were excluded for their terrible performance. For example, the 'lose lose' [14, pg. 135] hash function would have bucket lengths in the hundreds of thousands for most benchmarks.

5 BIT-PACKING

In this section the effect of bit-packing state variables is evaluated. MCSTA compiles a model and all its variables into various automata and a state structure. This state structure will contain various variables with limited ranges, leading to opportunities for shrinking the memory footprint of state space exploration.

5.1 Background

Bit-packing is the trimming of unused bits for integer variables with defined value ranges smaller than existing aligned integer types (bytes, shorts, ints, longs). By means of example, consider the primitive used to represent a Boolean in most languages. In a language such as C++ or Rust it is a full byte, where 0 commonly means *false*, and any value not 0 means *true*. Since Booleans have but two possible values (*true*, and *false*) all that would be needed is a single bit to store the Boolean's state. In general, $\lceil \log_2(x) \rceil$ bits are needed to hold a defined range's type, where x denotes the amount of values a variable can be – 2 in the case of a Boolean. In the same byte currently used to store a single Boolean one could instead store 8 Booleans packed together by making use of bit shifts and masks.

This is closely related to the current state layout used in MCSTA. As can be seen in Listing 2, the smallest aligned integer type possible is chosen for bounded integers. In case of the example model there are two bounded integers declared: *clients* and *address*, with 3 and 4 possible values, respectively. Both currently use a full byte to hold those values, but they could be expressed in as few as 2 bits each. Likewise, *is_success* could be expressed in a single bit.

At present the full `State` struct uses a total of 4 bytes to track the variables. It could, however, make do with roughly a fourth of the bits, as shown in Equation 4.

$$\frac{BitsNeeded}{BitsUsed} = \frac{2 * 2 + 2 + 1}{4 * 8} = 0.21875 \tag{4}$$

By making use of the restrictive nature of the declared types each state instance could be shrunk to use only a single byte for all the variables, instead of 4, shrinking the memory footprint of each individual state. This would, however, come at the cost of additional instructions to extract the current value when using a field (in most cases, a `SHIFT` and `AND` instruction). Whether bit-packing would improve performance depends entirely on the access patterns of the data, as a more cache-friendly access pattern might gain performance for memory-constrained applications such as model checking.

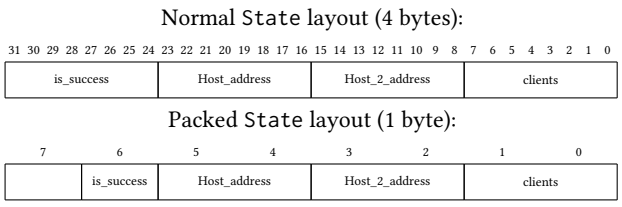


Fig. 3. Bit-packing example for the State layout in Listing 2

Algorithm 1: Bit-packing with Best Fit Decreasing

```
Input  : fields ← All variables of a model
Output : packedFields ← All provided fields with bin info
MAX ← 32
/* Binary search tree with duplicates          */
bins ← ∅
Sort fields by bits needed, descending
foreach field ∈ fields in order do
    /* searchFit finds the closest fitting bin  */
    fittingBin ← bins.searchFit(field.Bits)
    if fittingBin ≠ NIL then
        bins.remove(fittingBin)
        fittingBin ← fittingBin − field.Bits
        field.setPackedInfo(fittingBin)
        bins.insert(fittingBin)
    else
        bins.insert(MAX − field.Bits)
        field.setPackedInfo(fittingBin)
    end
end
return fields
```

5.1.1 Bin Packing. While it is trivial to calculate the amount of bits that could be saved in an optimal scenario, allocating these variables into 'bins' is not. Bit-packing is merely a slight rephrasing of the NP-complete bin-packing problem [7]. The bin-packing problem poses the following conundrum: Given a set of numbers, as well as a maximum bin-size, pack the set of numbers into as few bins as possible – without having the sum of numbers in any particular bin exceed the maximum capacity provided. It is clear that bit-packing is a slight rephrasing of the former. The set of numbers is represented by a model's variables and their respective bit size requirements, and the bins are represented by the fields which will contain all the packed fields (referred to as the owner fields).

The bins and the allocation algorithm mentioned above are necessary to ensure that no variable gets broken up across a bin boundary. We want to avoid the case where a variable of 9 bits has the lower 5 bits in one bin, and the upper 4 bits in another, for example. This would have adverse effects on performance due to the need for additional bit shifts and masks to acquire the value that is requested.

There are various approximation algorithms for solving the bin-packing problem in polynomial time [13]. Within these algorithms two distinct classes can be formed, offline and online algorithms. The former relies on the full set of numbers being provided ahead of time, and the latter is based on items coming in one at a time. Examples of such algorithms are First Fit Decreasing (FFD) and Best Fit Decreasing (BFD) for the former, and First Fit (FF) and Best Fit (BF) for the latter. The primary difference between the two classes is that offline can provide closer to optimal results due to some slight pre-processing of the input [13].

Based on the fact that MCSTA must compile complete models, where all variables are already known, it is clear that an offline variant is possible. We chose BFD as the algorithm for use within

MCSTA, chiefly for its slightly closer to optimal results in certain practical situations compared to FFD [13]. Both of these algorithms can run in $O(n \log n)$ time, where n is the length of the input list. BFD works by sorting the input numbers in decreasing order, and subsequently assigning each field to the fullest bin in which it fits. If no such bin exists then a new one is created instead.

An implementation as defined for the bit-packing problem is listed in Algorithm 1. The binary search tree keeps track of the remaining capacities of the bins. Here the *MAX* is set to 32 to limit the size of each bin, such that they can fit in a 32 bit integer. This maximum was deliberately chosen to keep the hash code calculations – which must return a 32 bit integer – mentioned in Section 4 cheap. At present the hash code would be calculated by a simple `ADD` and `MUL`, whereas a 64 bit integer would involve two of each, in addition to a bit shift and mask.

The result that is received from Algorithm 1 contains various bins with varying amounts of fullness. These bins are assigned the closest fitting primitive type for saving in MCSTA's `State` struct. For example, were a bin to contain 14 bits worth of fields it would be assigned a 16-bit integer.

5.1.2 Alignment. A consequence of the chosen *MAX* of 32 bits is that the alignment of the generated `State` starts to matter. Once we include a 4 byte integer the alignment of the `State` becomes 4 bytes. This means that if we were to create an array of `States` they would all need to be a multiple of 4 bytes in size to ensure aligned access to the integer used within [18]. This can have some important effects on the final struct size due to padding. As an example, say we have packed all variables of a model into 2 bins, of 32 and 6 bits, respectively. In ideal circumstances the total struct size would be 5 bytes, but since the alignment of the struct will be 4 bytes an additional 3 bytes of padding will be inserted by the compiler to maintain the alignment.

An obvious solution to the above would be to eschew the usage of any bin size larger than a single byte. This would ensure the alignment of a struct would always remain 1 byte, and thus no padding would ever be needed. This does, however, bring with it certain complications when it comes to variables larger than a single byte. A variable with a minimum of 9 bits would need additional bit shifts and masks to acquire the value, reducing performance and increasing complexity of implementation. Instead, we opted to delve into making use of the above padding to provide performance improvements for certain benchmarks as outlined below.

5.1.3 Performance Improvement. The result of running the bin-packing algorithm listed in Algorithm 1 is a close to optimal, and in most cases truly optimal, bin-packing of all variables. This packing will ensure the smallest memory footprint possible, but can come at a significant performance penalty in certain situations when ran for real-world models. For an exact overview of possible performance penalties refer to Section 5.3.1.

However, as there is usually some padding bytes involved, an algorithm can be devised to unpack certain variables from the owner fields to improve performance, whilst still maintaining a significantly shrunk state size. As there is only limited padding bytes available this unpacking would need to be based on a certain heuristic to estimate the relative benefit of unpacking a certain field over another. This heuristic could range from more advanced analysis of the abstract syntax tree to determine the likelihood of access, to more simple estimates based on a field's bit length. This research followed the latter strategy, where the rationale flows forth from the intuitive idea that fields with more bits allocated to them are more likely to be updated frequently during state-space exploration.

Algorithm 2: Bit-packing Performance Optimisation

```
Input  : fields ← All variables of a model
Output : packedFields ← All packed fields with bin info
packed ← BinPacking(fields)
filtered ← ∅
alignedSize ← packed.AlignedSize;
wasteBits ← alignedSize * 8 − packed.BitsUsed
Sort location variables to top of packed, followed by bit
  length descending.
foreach field ∈ packed in order do
    if field.WasteBits ≤ wasteBits then
        wasteBits ← wasteBits − field.WasteBits
        filtered.insert(field)
    end
end
while true do
    /* Bit-pack the symmetric difference        */
    packed ← BinPacking(fields△filtered)
    if filtered.ByteSize + packed.ByteSize > alignedSize
     then
        filtered.removeLast()
    else
        return packed
    end
end
```

In addition to the fields explicitly declared in a model MCSTA will also create so called *Location* variables to keep track of a particular system instance's program counter. As the program progresses this is constantly accessed, and in turn these variables are most likely to provide performance improvements when unpacked. We therefore decided to unpack these variables first, followed by user declared fields in decreasing order of their bit length.

The full packing optimisation algorithm is outlined in Algorithm 2. The core idea is to calculate the number of 'wasted bits' – bits which are still unused due to padding or larger packed field allocations – and selectively unpack fields in the order described prior in this section. Individual field waste bits in turn refer to the amount of bits that would remain unused if their primitive type representations were used instead. For example, in the case of a `Boolean` this would be 7 bits. Afterwards, bin-packing has to be ran again, as the remaining fields might no longer pack as well as before, thereby increasing the state size compared to the previously established optimal. So long as that is the case filtered fields will be added back into the packed fields list. Eventual resolution is ensured, as worst-case all filtered fields will be unfiltered.

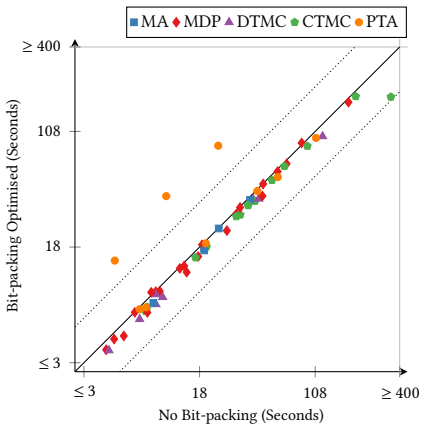


Fig. 4. Exploration time (seconds), grouped by model type

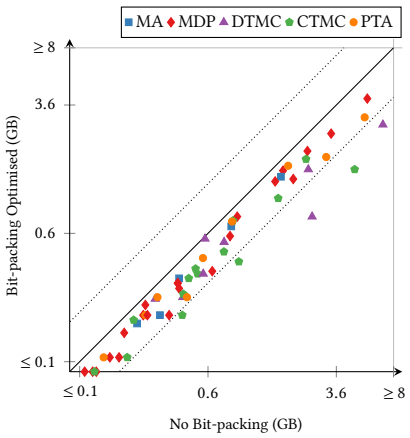


Fig. 5. Memory usage (GB) of hash table, grouped by model type

5.2 Experimental Design

Benchmarking is the core of the strategy for evaluating the efficacy of bit-packing. The benchmarks measure run-time, and memory usage at the end of the state-space exploration step. The benchmark suite as defined in Section 3 was used. All instances were ran three times, and the results of variable fields (exploration time) averaged to mitigate thermal noise or other processes interrupting the benchmark.

An additional comparison is made between the unoptimised bit-packing when only using Algorithm 1, and the optimised bit-packing when using Algorithm 2. The evaluation of the efficacy of the latter is based on run-time improvements, as memory usage remains the same due to sticking within the allocated state size.

5.3 Results, Discussion, Conclusion

In this section the results of the benchmarks are listed. See Table 1 for a tabular overview of the results. Note that some benchmarks with three instances had the smallest excluded to fit the table. In the table every model and the parameters used to run them are listed. The total amount of states for that particular instance, alongside the respective state sizes (**s. size**) is displayed. Exploration time is denoted as **time**; in the case of bit-packing there are two times listed, the optimised algorithm (**time (o)**), and the unoptimised algorithm (**time (d)**). Lastly, both total memory consumption (**mem**), and hash-table memory consumption (**h-mem**) are listed as well. The latter is where bit-packing shows its effects most clearly. Note that there is no separate entries for the optimised and unoptimised bit-packing memory consumption, as both are the same.

For comparing run-time and memory usage scatterplots with a logarithmic scale are used. See Figure 4 and Figure 5. Each mark represents one instance. The dotted lines around the main diagonal line indicate a 2 fold change when comparing the x and y axis. When a mark lands in one particular half of the graph (separated by the solid diagonal line) it indicates that the value that we are plotting is larger for that axis. For example, in Figure 4 one can see three `PTA` instances in the bit-packing half of the graph. This shows that bit-packing was slower compared to no bit-packing for those cases.

Figure 4 gives an overview of the run-time performance of the optimised bit-packing compared to no bit-packing. On this sample of benchmarks a minor decrease in run-time is noted for most instances in favour of bit-packing. A notable exception is `WLAN-LARGE` (visible as `PTA` in Figure 4) where a 3.5 fold increase in run-time is visible instead. The exact reason why this model performs so poorly is difficult to ascertain, as it does not accesses a great amount of state variables. However, the behaviour is consistent even across several executions. Upon manual evaluation of the three outlier instances it is noted that the execution time of bit-packing is gradually reduced as more and more fields are unpacked. This indicates that there does not seem to be one critical variable bottle-necking execution, and it is instead the access pattern of the data that causes the slowdown.

We hypothesise that the run-time improvements that are noted in other benchmarks are primarily caused by the shortened hash-code calculation. The owner fields can be directly used in the hash code calculation, without having to `ADD` and `MUL` each individual sub-field like in the non bit-packing cases. Additionally, the smaller state sizes may improve cache utilisation. Since the hash-table is accessed in a largely random order – minimising potential cache hits – the effect would be minor, however. Lastly, less memory bandwidth would be needed due to the smaller state sizes, which might have an impact if the model checking was bandwidth constrained.

In contrast, the memory footprint of all instances is consistently smaller through the use of bit-packing, as can be seen in Figure 5 or Figure 6. In almost all cases the state instance size was reduced by 50% or more, in the exceptional case 80%. This provides roughly proportional gains when it comes to memory usage of the hash table, where all the states are collected. However, while the hash table saw a significant decrease in memory usage across the board, this does not translate to a great decrease in total memory consumption in Table 1. It is clear that most of the memory space is occupied by other objects besides state instances. Be that as it may, benchmarks with large states still see notable improvements as total state count increases. This is best illustrated by the `EGL` benchmark, where a decrease of 39% (10.66 → 6.5 GB) in total memory usage is noted.

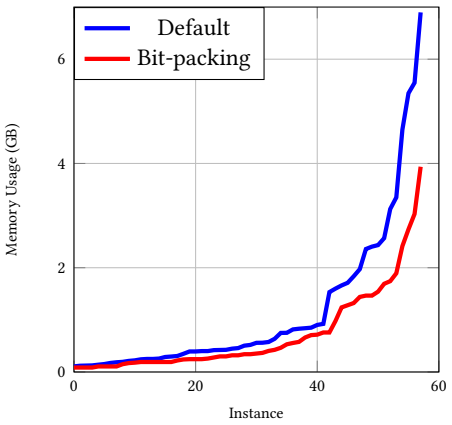


Fig. 6. Memory Usage (GB) of hash table, incrementing order

Based on the results of this sample of benchmarks we can conclude that bit-packing can decrease memory usage in models. Run-time can also see a slight decrease depending on the model, but in turn can also see explosive increases for some pathological cases. These exceptions warrant caution for using bit-packing by default.

5.3.1 Discussion Optimised Algorithm. Visible in Table 1 are three cases of performance regressions when looking at unoptimised bit-packing, compared to no bit-packing. `PACMAN`, `WLAN-LARGE`, and `WLAN-DL` saw regressions – with the former two being the most severe. The optimised bit-packing contains regressions for the same instances, but their severity was reduced. `PACMAN` saw a 50% decrease in run-time on its largest instance when compared to the unoptimised bit-packing (96.8 ⟶ 47.8 seconds). `WLAN-DL` saw a 32% decrease in run-time (13.1 ⟶ 8.9 seconds) on its largest instance.

The exception to the prior two is `WLAN-LARGE`, where no notable decrease in run-time was noted for its largest instance. The reason for this discrepancy is easily explained, however. The instances for parameters 3 and 4 are tightly packed. No fields could be filtered out following the rules in subsubsection 5.1.3. The compiled `State` was in essence no different from the unoptimised version. The smallest instance did, however, have several fields which could be unpacked, netting a 25% decrease in run-time.

6 CONCLUSION

Explicit model checking can be an important tool for verifying system behaviour. Complex models can, however run into practical limits with regard to memory and run-time. This paper evaluated the potential for improvement in two key areas, state hashing and bit-packing. In Section 4 we have shown that the current hash function used for MCSTA performs adequately, and extracting additional performance improvements from either the hash-table or hash function could prove to be a difficult task.

Section 5 evaluated the potential of bit-packing a model's state to shrink memory usage and potentially run-time as well. The findings of this section indicate that there is always a slight decrease in memory usage, and usually a decrease in run-time as well. This comes with the important caveat that the run-time can increase two fold in some few instances as well. The addition of a value based unpacking algorithm allowed several of these exceptional instances to reduce their run-time back to near non-bitpacked performance. This was not a panacea, however, as some instances still retain their worst case behaviour.

Future work then naturally flows forth from the above. Further evaluation of what value to assign each variable of a state instance for potential unpacking can have important consequences for run-time performance, as shown by the optimisation algorithm in Section 5.1.3. Additionally, observing the run-time consequences of allowing a more loose compression ratio to favour performance could prove an interesting next step. For example, where currently an instance's state might optimally shrink from 36 to 12 bytes, what are the performance implications if instead it was allowed to unpack fields up to 20 bytes, and how best to determine that optimal ratio.

Alternatively, one could also examine the unpacking algorithm itself. At the moment it is a rather naive implementation, but the problem is closely related to the NP-complete Knapsack problem. This is similar to the bin-packing problem. It differs in that one would need to pack only a subset of the most high-value fields into a single bin, instead of having to pack all variables into as few bins as possible. In this case the `State` instance would be the knapsack, with its max size being the aligned size. A new, optimised, solution could be created to solve this more efficiently using the solutions which already exist for the Knapsack problem.

REFERENCES

[1] C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen. 2003. Model-checking algorithms for continuous-time Markov chains. *IEEE Transactions on Software Engineering* 29, 6 (2003), 524–541. https://doi.org/10.1109/TSE.2003.1205180
[2] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of Model Checking (Representation and Mind Series).* The MIT Press. 1–16 pages.
[3] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem (Eds.). 2018. *Handbook of Model Checking.* Springer.
[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2001. *Introduction to Algorithms* (2nd ed.). The MIT Press.
[5] Christian Eisentraut, Holger Hermanns, and Lijun Zhang. 2010. On Probabilistic Automata in Continuous Time. In *2010 25th Annual IEEE Symposium on Logic in Computer Science.* 342–351. https://doi.org/10.1109/LICS.2010.41
[6] Cindy Eisner and Doron Peled. 2002. Comparing Symbolic and Explicit Model Checking of a Software System. 230–239. https://doi.org/10.1007/3-540-46017-9_18
[7] Michael R. Garey and David S. Johnson. 1990. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman Co., USA.
[8] Arnd Hartmanns. 2018. QCOMP Bounded Exponential Backoff Benchmarks. https://qcomp.org/benchmarks/#beb [Online; Accessed 28-April-2022].
[9] Arnd Hartmanns and Holger Hermanns. 2014. The Modest Toolset: An Integrated Environment for Quantitative Modelling and Verification. In *Tools and Algorithms for the Construction and Analysis of Systems*, Erika Ábrahám and Klaus Havelund (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 593–598.
[10] Arnd Hartmanns and Holger Hermanns. 2015. Explicit Model Checking of Very Large MDP Using Partitioning and Secondary Storage. In *Automated Technology for Verification and Analysis - 13th International Symposium, ATVA 2015, Shanghai, China, October 12-15, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9364)*, Bernd Finkbeiner, Geguang Pu, and Lijun Zhang (Eds.). Springer, 131–147. https://doi.org/10.1007/978-3-319-24953-7_10
[11] Arnd Hartmanns and Holger Hermanns. 2019. A Modest Markov Automata Tutorial. In *Reasoning Web. Explainable Artificial Intelligence - 15th International Summer School 2019, Bolzano, Italy, September 20-24, 2019, Tutorial Lectures (Lecture Notes in Computer Science, Vol. 11810)*, Markus Krötzsch and Daria Stepanova (Eds.). Springer, 250–276. https://doi.org/10.1007/978-3-030-31423-1_8
[12] Arnd Hartmanns, Michaela Klauck, David Parker, Tim Quatmann, and Enno Ruijters. 2019. 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 (Lecture Notes in Computer Science, Vol. 11427)*, Tomás Vojnar and Lijun Zhang (Eds.). Springer, 344–350. https://doi.org/10.1007/978-3-030-17462-0_20
[13] David S. Johnson. 1973. *Near-optimal Bin Packing Algorithms.* Ph. D. Dissertation. Massachusetts Institute of Technology.

[14] Brian W. Kernighan and Dennis M. Ritchie. 1978. *The C Programming Language*. Prentice Hall.
[15] B. J. McKenzie, R. Harries, and T. Bell. 1990. Selecting a hashing algorithm. *Software: Practice and Experience* 20, 2 (1990), 209–224. https://doi.org/10.1002/spe.4380200207 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380200207
[16] Gethin Norman, David Parker, and Jeremy Sproston. 2013. Model checking for probabilistic timed automata. *Formal Methods in System Design* 43 (10 2013), 164–190. https://doi.org/10.1007/s10703-012-0177-x
[17] Vassil Roussev, Golden G. Richard, and Lodovico Marziale. 2007. Multi-resolution similarity hashing. *Digital Investigation* 4 (2007), 105–113. https://doi.org/10.1016/j.diin.2007.06.011
[18] Sergey Slotin. 2021. Alignment and Packing. https://en.algorithmica.org/hpc/cpu-cache/alignment [Online; Accessed 08-June-2022].
[19] Frits Vaandrager. 2011. A First Introduction to Uppaal. https://www.mbsd.cs.ru.nl/publications/papers/fvaan/uppaaltutorial.pdf [Online; Accessed 05-May-2022].

model			no bit-packing				**bit-packing**				
name	params	states (millions)	s. size (bytes)	time (sec)	mem (GB)	h-mem (GB)	s. size (bytes)	time (o) (sec)	time (d) (sec)	mem (GB)	h-mem (GB)
BEB	4, 8, 8	36.6	20	27.5	3.46	1.53	12	23.2	25.1	3.17	1.24
	4, 8, 9	63.5	20	47.7	5.74	2.40	12	39.7	41.3	5.22	1.89
BRP	64, 12, 32, 256	56.8	36	60.5	6.68	3.13	12	53.3	62.7	5.30	1.74
	64, 16, 32, 256	95.5	36	109.5	11.31	5.35	12	97.7	107.2	8.99	3.03
CABINETS	3, 2, 1	4.4	22	19.3	0.53	0.22	12	17.8	21.0	0.48	0.17
CLUSTER	700, 2000, 20	17.7	20	31.8	3.25	0.75	4	28.9	33.1	2.97	0.46
	1024, 2000, 20	37.8	20	67.0	6.95	1.60	4	63.0	71.8	6.33	0.98
CONSENSUS	6, 4	2.4	19	9.7	0.52	0.12	8	9.1	11.1	0.49	0.09
	6, 6	3.5	19	14.2	0.74	0.15	8	13.4	16.9	0.70	0.11
COUPON	9, 4, 5	21.1	14	9.1	1.85	0.75	4	7.4	8.9	1.63	0.53
CROWDS	6, 20	10.3	33	10.2	1.40	0.56	12	8.3	9.5	1.18	0.34
	7, 20	45.4	33	45.0	6.16	2.43	12	38.1	43.0	5.19	1.47
CSMA	3, 6	84.9	19	87.5	8.95	3.35	8	90.3	86.0	8.02	2.41
	4, 4	133.3	24	180.8	15.02	5.55	12	169.9	171.3	13.41	3.94
DPM	6, 6, 5	10.5	15	8.8	1.34	0.40	8	7.6	8.6	1.26	0.32
	6, 7, 5	23.4	15	19.4	2.90	0.83	8	17.1	19.0	2.74	0.66
	6, 8, 5	47.3	15	39.4	5.84	1.66	8	37.4	41.8	5.51	1.32
EAJS	5, 250, 11	3.0	21	4.2	0.38	0.13	8	3.7	4.4	0.34	0.09
	6, 300, 13	3.7	24	4.8	0.48	0.17	8	4.3	4.9	0.41	0.11
ECHORING	512	4.4	46	6.6	0.61	0.35	16	6.5	7.8	0.46	0.19
	1024	8.8	46	13.3	1.16	0.64	16	12.9	15.8	0.88	0.35
EGL	10, 1	24.1	87	41.3	3.96	2.57	12	37.8	38.6	2.15	0.76
	10, 2	66.1	87	121.1	10.66	6.90	24	100.1	119.0	6.50	2.74
FIREWIRE	1, 36, 300	23.5	18	31.8	3.43	0.90	12	30.0	35.6	3.28	0.76
	1, 36, 400	44.6	18	60.3	6.49	1.71	12	58.0	68.9	6.22	1.44
FIREWIRE-PTA	48, 7500	21.7	18	19.8	2.02	0.84	12	19.1	19.2	1.89	0.71
	60, 7500	48.9	18	44.0	4.47	1.84	12	42.9	43.9	4.18	1.54
FMS	9	11.1	24	38.1	3.15	0.50	12	34.5	35.1	3.01	0.37
	11	54.7	24	202.2	16.11	2.36	12	185.8	185.1	15.44	1.69
KANBAN	5	2.5	20	7.8	0.77	0.12	8	6.7	8.3	0.73	0.09
	6	11.3	20	38.3	3.51	0.46	8	34.3	40.5	3.37	0.32
MAPK_CASCADE	7, 30	4.0	29	20.0	1.40	0.19	8	18.1	21.2	1.31	0.11
	8, 30	10.3	29	55.3	3.77	0.52	12	50.7	57.3	3.59	0.34
NAND	60, 2	9.4	9	4.4	0.75	0.29	4	3.6	4.2	0.71	0.24
	60, 4	18.8	9	9.1	1.51	0.58	8	8.6	9.0	1.49	0.56
PACMAN	40	2.8	15	18.7	0.25	0.11	8	18.7	37.5	0.23	0.09
	50	5.0	15	33.7	0.44	0.19	8	33.2	67.6	0.40	0.15
	60	7.3	15	48.2	0.60	0.24	8	47.8	96.8	0.55	0.19
PHILOSOPHERS	20, 1	45.2	81	349.0	19.78	4.65	12	184.5	210.4	16.60	1.47
POLLING	18, 16	7.1	39	42.5	2.26	0.42	8	36.7	42.5	2.04	0.19
	19, 16	14.9	41	95.7	5.00	0.92	8	86.0	96.3	4.48	0.40
SF	2, 4	4.8	38	24.2	0.54	0.31	16	24.0	26.7	0.42	0.19
TANDEM	2047, 1000, 2	8.4	8	8.0	1.12	0.21	4	7.1	7.7	1.09	0.18
	4095, 1000, 2	33.6	8	33.8	4.49	0.85	4	29.7	31.7	4.36	0.72
TIREWORLD	35	9.2	21	17.6	1.70	0.40	8	15.5	15.8	1.58	0.28
	45	46.1	23	69.4	6.64	1.97	8	65.4	63.7	5.95	1.28
WLAN-DL	4, 80	7.0	17	5.6	0.69	0.26	8	4.5	5.9	0.63	0.19
	5, 80	9.6	17	8.0	1.01	0.39	8	6.5	7.7	0.91	0.30
	6, 80	10.1	17	8.5	1.05	0.39	8	8.9	13.1	0.95	0.30
WLAN-LARGE	2	3.3	16	4.8	0.45	0.14	8	14.6	19.5	0.41	0.11
	3	7.3	16	10.7	0.91	0.25	8	39.7	43.1	0.85	0.19
	4	16.7	16	24.0	2.00	0.56	8	86.6	93.8	1.87	0.43

Table 1. Benchmark results for bit-packing including state size, exploration time, memory, and hash-table memory used. Bit-packing has optimised (o) and non-optimised (d) results listed for exploration time.