Techniques for Memory-Efficient Model Checking of C and C++ Code*

Petr Ročkai, Vladimír Štill, and Jiří Barnat

Faculty of Informatics, Masaryk University
Brno, Czech Republic
{xrockai,xstill,barnat}@fi.muni.cz

Abstract. We present an overview of techniques that, in combination, lead to a memory-efficient implementation of a model checker for LLVM bitcode, suitable for verification of realistic C and C++ programs.
As a central component, we present the design of a tree compression scheme and evaluate the implementation in context of explicit-state safety, LTL and untimed-LTL (for timed automata) model checking. Our design is characterised by dynamic, multi-way adaptive partitioning of state vectors for efficient storage in a tree-compressed hash table, representing the closed set in the model checking algorithm. To complement the tree compression technique, we present a special-purpose memory allocation algorithm with very compact memory layout and negligible performance penalty.

1 Introduction

Model checking is an important verification technique with wide applicability in software development. The older generation of model checking tools primarily targeted special-purpose "modelling" languages, and as such are suitable for stratified, long-term development processes. In those cases, the role of the model checker was towards the early stages, especially in high-level design. However, the trend in the software industry is towards much more tightly integrated development cycles, where all activities are coupled as closely as possible to coding and early deployment. In those scenarios, it would be impractical to add a long and drawn-out process of modelling design elements that are to be programmed (coded) in the implementation language at almost the same time. It is those concerns that motivate the current work on model checking code directly. Additionally, such tight integration of programming and model checking has other benefits: it becomes possible to use the model checker to verify implementation-level properties this way (as contrasted with design-level properties). As such, a sufficiently powerful model checker has the capacity to enter the programmer's

* This work has been partially supported by the Czech Science Foundation grant No. 15-08772S.

† The final publication is available at Springer via `http://dx.doi.org/10.1007/978-3-319-22969-0_19`

toolkit alongside interactive symbolic debuggers (like `gdb`) and runtime analysis tools (like `valgrind`).

While it is quite obvious that those are all worthwhile goals, model checking of executable code presents substantial challenges. In the case of explicit-state model checking, the approach used by the DIVINE model checker [1], those challenges derive from the large number of distinct states reachable through execution of programs. This is most pertinent to multi-threaded programs, where model checking happens to be also most useful. Besides the size of the state space, the primary challenge in verifying a program directly lies in the interpretation of the source code. In DIVINE, this challenge was quite successfully resolved by using a standard C/C++ compiler with an LLVM backend, and by interpreting the resulting bitcode instead of the (much more complicated) original source code. Besides simplifying the implementation of the model checker, this also removes large portion of the complicated C++ compiler from the trusted code base.

The remaining challenges, stemming from large state spaces, are hence twofold: the time required to explore the state space, and the memory required to store it. Some techniques attack both problems at once: reduction techniques that vastly reduce the number of reachable states are one such approach [7]. In this regard, DIVINE employs a very successful τ+reduction [8] which removes many thread interleavings and compresses state chains down to a single transition, without compromising the soundness of model checking. Some approaches target one of those problems specifically: one such is parallelisation, which exclusively aims at reducing the time required for a verification run to complete. This is an important goal because a verification tool that can be used interactively is more valuable than a batch one, where the user needs to wait overnight (or for a week) to obtain the result. In this regard, DIVINE employs parallelism extensively and achieves decent speed-ups through its use.

Finally, despite extensive state space reduction, the state spaces obtained from C (and especially C++) programs are very large, memory-wise. And while parallelism gives us an acceptably fast algorithm, it is easy to run out of available memory. Of course, there is always room for optimisations: the LLVM interpreter embedded in DIVINE is currently the main speed bottleneck, and as such is subject to ongoing optimisation effort. Nonetheless, even in its current incarnation, on most computers, DIVINE will run out of memory very quickly. As such, techniques that reduce memory use are of prime importance, even if they have a modest negative impact on speed.

1.1 Reducing Memory Use

There are a few elements in an explicit-state model checker where large amounts of (fast, random-access) memory are required. Usually, by far the most extensive is the representation of the closed set, although the open set (usually a queue in a parallel model checker) can become quite large as well. The representation of the program being model checked is usually small and of constant size throughout the computation, as is the code of the model checker itself. Hence, for all but very small models, the memory requirements of the model checker are dominated

by the open and closed sets, which are composed of state vectors and often some ancillary per-state data of the model checking algorithm. Besides the state vectors themselves, the fact they are organised in a data structure (a hash table, a queue or similar) causes memory overhead of its own. While with "plain" LLVM-based model checking the state vectors are very large (often many kilobytes), and as such, eclipse the memory requirements of all the data structures that hold them, we will see the importance of memory efficiency of those data structures rise in prominence when the amount of memory occupied by a single state vector shrinks considerably.

One important technique that can contribute to memory efficiency of explicit-state model checking is lossless compression. Several methods of lossless compression – including methods based on state vector decomposition – were introduced over the time as discussed in Section 1.2. In our work we present an extension of existing state vector decomposition methods that is particularly well suited for real-world application of model checking of C and C++ code through LLVM bitcode – it supports dynamically sized states, has no need for preallocation of fixed-size closed set and supports parallel model checking. We show in our experiments that for verification of real-world programs with DIVINE, the method we describe constitutes enabling technology. That is, we show that it is possible to verify programs where verification without compression would require terabytes of RAM.

1.2 Related Work

The oldest and simplest lossless compression method was to use a generic data compression algorithm (Huffman coding, arithmetic coding, etc.) to compress individual state vectors before storing them into memory [5, 3]. These approaches only minimally exploit the redundancy *between* different states, which is usually much higher than the redundancy *within* a single state vector.

In this respect, a better method has been proposed in [4], where the state vector is decomposed and each slice of the vector is hashed separately and only indices to those slices are saved as a state. This exploits the fact that many state vectors contain parts that are identical between different states and also much longer than a single pointer – hence, storing a pointer to a separately hashed slice is more memory-efficient than storing the duplicated area repeatedly. While this idea is in a way a specialisation of otherwise very generic and well-known dictionary-based compression (as employed by the commonly used LZ77 [10] algorithm), it has some special properties that make it more interesting for model checking: namely, the construction of the "dictionary" makes it easy and efficient to hash the compressed states and compare them for equality – neither of those steps needs to decompress states already stored.

The one-level scheme proposed in [4] has been improved upon by [2], making it fully recursive. It also removes the requirement that the compression algorithm knows specifics about the state vector layout. This recursive approach has been further adapted for parallel model checking in [6]. One downside of this

implementation is a requirement for a fixed-size, pre-allocated hash table with fixed size slots.

We use a similar scheme, but we re-introduce *optional* state vector layout awareness into the compressor, we use generic n-ary trees instead of binary, we use resizing hash tables in the implementation and we focus on dynamically sized states which naturally occur in LLVM-based programs which include memory allocation.

2 Tree Compression

Depending on the verification task, the storage size of a single vertex (state) can be fairly large. This is especially true of more complicated model checking inputs, like timed automata or LLVM[1]. In those cases, it makes sense to consider compression schemes for states and/or the entire state space. In DIVINE, we have implemented the latter [9], using a scheme similar to *collapse* [4]. Since our hash table is resizeable to facilitate better resource use, we cannot directly use some of the improvements that rely on fixed-size hash tables [6]. On the other hand, since the hash table we use can accommodate variable-size keys, we are not limited to fixed-layout trees and can use content-aware state decomposition like in the original *collapse* approach (but unlike original collapse, we can decompose the state recursively, which is useful with more complex state vectors, like those arising from LLVM inputs). The decomposition tree structure is illustrated in Figure 1.

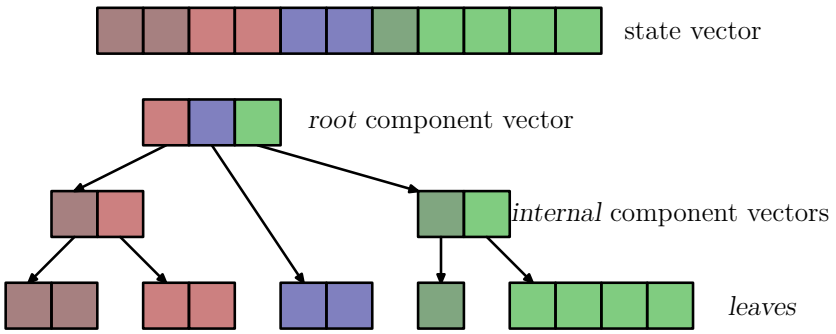

Fig. 1. A decomposition of a state into a component tree. The leaves represent fragments of the original state vector.

Our approach uses three hash tables that are adaptively resized as needed. One holds root elements – one root element corresponds to each visited state

[1] In theory, nothing about LLVM per se causes states to be large; in practice, however, inputs that are expressed in terms of LLVM have a tendency to have much richer state than more traditional formalisms, like DVE or ProMeLa.

1:1. These root elements are represented as component vectors, where each component is represented as a separate object in memory. Those components are de-duplicated using a *leaf table* – a state fragment that is identical in multiple different states is only stored once, and the *root* table refers to the de-duplicated instances of those objects. To facilitate recursive decomposition, we also maintain a third table, *internal*, for internal nodes of the state decomposition tree. The *internal* nodes have the same structure as *root* nodes (a vector of pointers), but they do not correspond to complete states and the *internal* table is not consulted by the model checking algorithm when looking up vertices during search.

The component vectors contain a flag to decide whether a particular component is another component vector or a state fragment, as otherwise they are not distinguishable – both are stored as raw byte arrays in memory, without distinct headers. Clearly, reconstructing a state vector from a component vector is easily done by walking the decomposition tree and copying leaf node content to a buffer from left to right. In theory, storing the size of the entire state in the *root* component vector could improve efficiency by making the reconstruction work in a single pass, copying fragments into a pre-allocated buffer. In practice however, the decomposition trees are small and the requisite pointers are retained in fast CPU cache on the first pass (when the buffer size is computed), making the savings from a single-pass algorithm small. Moreover, the extra memory overhead of storing another integer along with each state is far from negligible.

The trade-off inherent in tree-based compression schemes is visible in Figures 1 and 2. Compare the number of squares (memory cells) in these two pictures. The original state vector occupies 11 cells, its decomposition uses 18 cells. However, adding another similar state (state B in Figure 2) increases the memory use only by 9 cells in the compressed variant, while it would add another 11 cells without compression. The state vectors illustrated here are extremely small; real-world LLVM states typically occupy thousands of memory cells and bigger states naturally favour compression. On the other hand, a realistic implementation introduces slightly more memory overhead than the idealised picture show here.

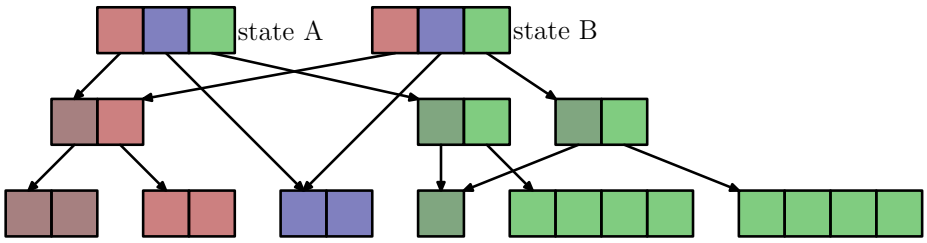

Fig. 2. A de-duplicated pair of states. The layers are analogous to Figure 1. States A and B differ only in the light green component.

2.1 Splitting State Vectors

The fact that both the component vectors in the internal nodes and the state vector fragments stored in the leaf table are of possibly variable size (and making them fixed-size would not improve compactness, thanks to the memory allocator design described in Section 3), we gain the capability to decide on how to split state vectors dynamically. This capability can be used to align boundaries of both leaf fragments as well as their groupings with logical divisions of the state vector. The working hypothesis is that this would improve compression ratio, since changes between state vectors that are neighbours in the state space have a tendency to be localised within the state vector. By correctly aligning the split points for the purposes of compression, we expect the changes between a pair of related state vectors to be localised to the smallest possible subtree. Moreover, the size of a decomposition tree has an impact on performance: if we can identify large contiguous chunks of the state vector that change only rarely, if at all, we can reduce the size of the decomposition trees and thus improve the overall speed of verification. On the other hand, if those larger chunks in fact do change, this has adverse effect on compression ratio. Therefore, finding a good way to split the state vectors is a balancing act: smaller leaf fragments and more balanced trees lead to better compression, but incur higher performance penalty. Of course, leaf fragment size cannot be reduced arbitrarily: to achieve compression, a leaf must be strictly larger than a single pointer (8 bytes), since the reference in the parent node is represented using a pointer.

2.2 Interactions

The tree compression methods interacts with other components of the model checker. First, the memory allocation regime is an important aspect: how big a pointer to a node is, for example, is quite important from the perspective of compression ratio. With 32-bit pointers, compared to 64-bit, we could expect nearly twice the memory efficiency. However, that would also limit the number of nodes in the compression tree to about 4 billion: considering that on realistic x86-class hardware, exploring and storing 40 billion states is possible, and even if we neglect the requirement to also store internal and leaf nodes of the tree, 32-bit pointers are clearly insufficient.

Another aspect to consider is how the requirements of parallel exploration affect the compression method. In shared memory, DIVINE offers two exploration modes, *shared* and *partitioned*. On modern hardware, the *shared* mode is usually faster, especially with higher thread counts. In the context of compression, it offers another important advantage: since it uses a single hash table which is shared by all the workers, tree compression is very efficient. Since all states are stored in the same (compressed) hash table, all redundancy can be exploited for compression. With the *partitioned* scheme, on the other hand, each state is statically assigned to a particular worker thread, and each thread maintains a private hash table. This hash table is slightly more efficient (because access to it does not need to be thread-safe), but this advantage is usually outweighed by

more costly communication between the threads which need to exchange states based on the partitioning. The effect on compression is even more pronounced, though: since each thread stores – and compresses – state vectors privately, a large fraction of the leaf and internal nodes will be duplicated. This happens whenever two state vectors share a subtree, but are assigned to different worker nodes. This subtree would only be stored once in the *shared* scheme, but twice in the *partitioned* scheme.

On the other hand, DIVINE also offers a *distributed-memory* mode, using MPI for communication. This mode necessarily works just like the *partitioned* mode in shared memory: each machine in the cluster has a private hash table and compression is performed locally within that hash table. This means the compression will be less efficient in distributed-memory situations, nonetheless substantial savings are still possible.

Finally, besides the closed set stored in the hash table (or hash tables in partitioned and distributed modes), a model checker needs to maintain an open set. In parallel algorithms, both for checking safety (reachability) and for LTL model checking (OWCTY), this is often a queue. Since the compression method we use is lossless, the state vector can be reconstructed from its compressed form and it is possible to also compress the open set, in addition to the closed set.

3 Memory Allocation

Memory allocation is an extremely frequent operation in an explicit-state model checker. Moreover, the memory pool that threads allocate from is a shared resource, requiring certain amount of synchronisation. One way to side-step this issue is to statically pre-allocate as many resources as possible – this is the approach taken by, most prominently, the model checker SPIN. The main downside of this approach is that the tool either has to "guess" resource use very well ahead of time, or rely on the user to provide guidance. In all but very simple scenarios, the former is very hard to get right – models vary wildly from one to another in which parts of the model checker they stress. Some require very long queues or deep stacks, even when the overall size of the state space is comparatively small. Others only need a very small queue but the state space is huge, and almost all memory needs to be allocated towards the closed set. Some models have few big states, requiring few slots in the hash tables, but need a lot of memory for storing the states themselves.[2]

However, there is a more important limitation, namely with regard to multitasking: users expect to be able to execute multiple instances of a program at the same time, especially if the verification runs are well below the limits of the computer they are using. Static resource allocation in such cases becomes a chore – especially so if multiple users are involved on shared hardware. In most cases, we aim at interactive use: batch scheduling is only suitable for very

[2] The LTSmin model checker avoids this particular resource split by storing state vectors decomposed, each fixed-size chunk stored inline in the large pre-allocated hash table.

large instances, where the entire computer (or a cluster) is tied up in a single verification task. Meanwhile, a large SMP system can easily serve many tasks and many users interactively – but this means that tasks should only consume resources that they actually need, so that resource conflicts are minimised. This is very hard to achieve if memory needs to be pre-allocated at a time when the size of the state space is not yet known.

To address those issues, DIVINE uses dynamic allocation for all resources, achieving optimal hardware utilisation when multitasking. There are, however, multiple challenges associated with this flexibility, especially when dealing with parallel algorithms.[3]

3.1 Allocation Profile

When designing a custom memory allocator, the first thing to ask is what is the allocation profile of our target application. Are object sizes similar, or distributed across a wide spectrum? Are there many small allocations, or few big allocations? Is memory retained for a long time, or a short time? Is memory deallocated often?

We can answer most of those questions for DIVINE: for one, there is a tendency to see many objects of similar size. This is most visible in models with fixed-size states (this is actually the case with majority of input languages in DIVINE: most traditional modelling languages require all state variables to be explicitly declared and do not provide dynamic variables). It is also true, to a smaller extent, with variable-size state vectors: many states will differ in content but not the size of the state vector. For LLVM, state size changes when a thread is created, a function is entered or left and when a new thread is created or when heap memory is allocated. All these operations are comparatively rare, so we can expect many states of any given size to appear over time. This is even more pronounced when compression enters the picture, since the fragments have more uniform sizes than the entire state vectors. This favours a design where objects of a particular size are grouped into bigger blocks, reducing overheads in the parent allocator (both time and memory overhead).

This type of layout also offers the opportunity to store exact object size as allocator metadata, once per block of objects. When state vectors (or their fragments) are of variable length, their length needs to be stored somewhere: if each state vector stores its own length, this either adds 4 bytes of overhead per state (or, when using 2 bytes, causes the rest of the vector to be stored unaligned which incurs a large performance penalty). Both are far from optimal. If the size is stored once per block, a single 4-byte word can be used to keep the size for hundreds of objects, saving considerable amounts of memory. It does mean that the allocator needs to be able to find block metadata from a pointer, to read the

[3] Intra-process parallelism can be very useful even when multiple verification instances are involved. A 64-core system can easily accommodate 4 verification tasks running on 16 cores each, splitting memory between those 4 tasks as needed. If memory becomes scarce, some of the processes can be suspended and swapped out to disk and later, when other tasks have finished, resumed again.

object size associated with the pointer. This particular optimisation also cancels out the extra overhead from adaptive, recursive state splitting employed in our compression scheme. For root and internal nodes, the size of the node (obtained through the allocator) can be used to easily compute the number of children. Likewise, the size of a particular leaf fragment can be cheaply extracted from the allocator metadata.

Second, there are two main classes of objects during state space exploration: the first class contains state vectors that are part of the closed set, and will be reclaimed at the end of the verification run, but not earlier. The second class contains newly generated successor states that may or may not be duplicates of states in the closed set – some of those will go on to be added to the closed set (which may require their re-allocation if compression is enabled) while others will be deallocated when they are found to be duplicates. In other words, some objects are short-lived, and some are very long lived – however, there are few, if any, "in-between" objects. This split would favour a generational allocator – especially since we often know ahead of time whether a particular object will be short- or long-lived (at least in the case where compression comes into play – in other circumstances, the distinction is less clearly cut).

Since compression is such an important ingredient, its requirements need to be considered in the design of a good memory allocator. The considerations laid out above lead to a design where memory is allocated in blocks of same-sized objects. For a number of reasons, it is impractical to reclaim blocks that have been already claimed for a particular object size for another object size (here, parallel access is the main reason that an efficient solution is not known to exist). However, when compression is in use, the state vectors that are allocated during successor generation (into the open set) only exist for a very short time, since they are immediately moved into the compressed state store. Consequently, if the same allocator was to be used for those ephemeral state vectors, a substantial amount of memory would be claimed but unused. While the amount of memory so wasted is only proportional to the number of different state vector sizes (and as such not very large), it can add up to many megabytes. More importantly, this overhead appears in each thread separately and is therefore also proportional to the number of execution threads. So while raw speed is not affected much by a generational approach, memory efficiency can be jeopardised. With those considerations in mind, when state compression is enabled, ephemeral memory is obtained from a simple, special-purpose allocator.

3.2 Pointer Representation

There are two basic options on how to represent pointers: either use raw machine pointers, or use an indirection scheme. The former has a clear advantage in terms of access speed: dereferencing a raw machine pointer is as fast as it gets – any other representation will incur additional costs. On the other hand, most contemporary platforms use pointers that are 64 bits wide – for realistic memory sizes, this constitutes substantial overhead. Current CPUs can physically address at most 48-bit memory addresses, while the rest of the pointer representation

is unused – that is 16 bits of memory lost for every pointer. Moreover, there are plenty of places in DIVINE where extra bits packed inside pointers can save considerable amount of memory: the hash tables, for example, can use (some of) those 16 bits to store a small part of the hash value to avoid full object comparisons and speed up lookups at no extra memory expense. Quite importantly, the compression algorithm can use a few of those bits for type-tagging pointers, making it free, in terms of memory use, to distinguish state vector fragments from state component vectors (cf. Section 2).

Moreover, a custom pointer representation enables the allocator to easily find the block header for any given pointer, making it possible to obtain object sizes from pointers to those objects. As explained in previous section, this can save considerable memory in some cases.

The main downside is that the pointer dereference operation needs to consult a lookup table to reconstruct the raw machine pointer. The lookup tables can be represented in such a way that this can be implemented using a single addition instruction, followed by a memory fetch from the lookup table, followed by another addition instruction. Since the lookup tables are relatively small, we can hope that they will always be readily available from fast CPU cache. Maybe more importantly, there will only be very few very hot cache lines in those lookup tables. In our informal testing, the slowdown from this indirection was in single-digits percent range, while the memory savings were quite substantial. Based on this, we have decided to use indirect pointers for storing states and state fragments.

3.3 Implementation

The considerations laid out in previous sections give us a fairly good guidance on how to implement an efficient allocator for use in DIVINE. Our implementation uses a custom pointer type, which is translated to machine pointers on demand, at the cost of an extra memory fetch (which is expected to be served from cache, since the indirection table is usually very hot) and a couple of addition instructions. All data structures in the hot paths of the allocator (object allocation and deallocation) are thread-local and expensive thread synchronisation only happens in special circumstances, usually after some threshold is exceeded: either per-thread freelists have grown too big, or they have become empty; or when all freelists are empty and no pre-allocated memory is available, in which case it needs to be obtained from the operating system.

The shared data structures: indirection tables and lists of shared freelist, are implemented as standard lock-free data structures. Since they are only accessed comparatively rarely, no special precautions need to be taken to make access to them more efficient – the indirection table is almost entirely read-only – it is only written when a new block is allocated. Additionally, a shared counter is maintained to assign blocks to threads (threads claim 16 blocks at once to minimise contention on this counter; the blocks are only allocated when they are needed though).

Table 1. Scaling of pthread_rwlock LLVM model with and without compression and with different splitters.

Configuration	W=1		W=2		W=4		W=8	
	time	scale	time	scale	time	scale	time	scale
no comp.+eph alloc.	7581	1	3785	2.00	1985	3.82	1009	7.51
tree+none+generic	11094	1	6052	1.83	3000	3.70	1499	7.40
tree+old+generic	11625	1	6230	1.87	3074	3.78	1559	7.46
tree+eph+generic	11332	1	5693	1.99	2981	3.80	1523	7.44
tree+eph+hybrid	11258	1	5677	1.98	2973	3.79	1518	7.42
tree+eph+obj-mono	11227	1	5727	1.96	2972	3.78	1519	7.39
tree+eph+obj-rec	11265	1	5743	1.96	3006	3.75	1540	7.31

4 Measurements

We implemented the aforementioned scheme in DIVINE and evaluated it using several large C and C++ models translated into LLVM. We also verified general usability of this scheme by benchmarking a few UPPAAL Timed automata models. All the models can be found in DIVINE source distribution. In this section we will give a detailed analysis of our results.

To measure memory requirements, we used DIVINE's simple statistics output which allows us to track memory allocation during a verification run. We measured resident memory usage, either for DIVINE as a whole or divided by number of states explored; either way, the number in statistics is adjusted by subtracting resident memory used before the model is loaded and before the verification algorithm starts – this allows us to easily compare numbers between different configurations of DIVINE, but still includes all the overheads of the algorithm, such as overhead of thread-local data in a multi-threaded setting. Memory measurements were performed on several computers in a way no memory swapping could have occurred.

For time measurements, we take wall time from DIVINE's report. This time includes the initialisation of the algorithm and the time required to load the model. Time measurements were performed on server with two Intel Xeon E5-2630v2 CPUs at 2.60GHz with 128GB of memory.

Besides the detailed measurements presented in the following sections, we have also measured (using the same set of models) that on average, verification with compression generates states at 77% of the speed of uncompressed algorithm in case of single threaded run, and 73% for 8 workers. We have also measured the scaling behaviour of various configurations of compression and memory allocation schemes. The results of those measurements are summarised in Table 1.

4.1 Allocation schemes

Table 2 shows how memory requirements of DIVINE with tree compression vary based on the allocation scheme used and the number of worker threads. In this case we have considered three variations of allocation scheme:

Table 2. Memory use of LLVM models with compression depending on memory allocator and number of workers.

Name	Average state memory (B)														
	W=1			W=2			W=4			W=8			W=16		
	n/a	old	eph	n/a	old	eph	n/a	old	eph	n/a	old	eph	n/a	old	eph
pt_rwlock	105	90	**88**	106	93	**89**	106	96	**90**	106	104	**90**	109	121	**94**
pt_barrier	60	**45**	**45**	65	**53**	**53**	64	53	**52**	63	54	**52**	63	**53**	**53**
collision	252	232	**229**	253	237	**229**	253	245	**229**	257	261	**235**	265	296	**246**
elevator2	105	**81**	**81**	106	**82**	**82**	106	**82**	**82**	106	**82**	**82**	107	84	**83**
lead-uni_basic	55	**45**	**45**	56	47	**45**	55	48	**45**	56	52	**46**	57	59	**48**
lead-uni_peterson	66	57	**56**	67	59	**56**	67	61	**56**	67	67	**58**	69	79	**60**
hashset-2-4-2	243	202	**191**	244	213	**191**	244	232	**192**	246	270	**194**	250	340	**198**
	W=40														
hashset-3-1	67	77	**47**												

n/a direct allocator, which uses raw machine pointers, and allocates them using general purpose allocator (TBB `malloc`); this scheme stores the size of each entry directly in the memory of the entry, which increases its overhead;

old indirection allocator from Section 3.3 without ephemeral memory optimisation;

eph indirection allocator from Section 3.3 with ephemeral memory optimisation.

It can be clearly seen that indirection allocator with ephemeral memory optimisation is the best option, providing best memory efficiency among the considered options. While the indirection allocator without ephemeral memory optimisation provides comparable efficiency in single-threaded verification, it quickly loses to the optimised version as number of workers increase; this is caused by thread-local overhead of the allocator when allocating short-lived blocks of different sizes. Furthermore, for sufficient number of workers, overhead of the per-thread structures of this allocator can outweigh per-state overheads of the naive solution. These measurements show the importance of an efficient memory allocation scheme for multi-threaded verification, which was further emphasised on `hashset-3-1` model with 630 millions of states, which was verified using 40 worker threads: here, the naive solution has 43 % overhead over our allocator with ephemeral storage, while the allocator without ephemeral storage has 64 % overhead over ephemeral storage allocator. This shows that efficient parallel allocator is a necessary part of memory-efficient parallel verification.

4.2 Compression efficiency

Tables 3 and 4 list overall memory usage and memory usage per state, respectively, including memory usage for various state-vector splitting strategies:

none Verification without compression. For large models (where more than 320 GB RAM was required to finish verification) this value is a lower bound based on average state size and the number of states as reported by a run

Table 3. Total resident memory used for LLVM models, without and with compression with different splitters.

Name	# of states	memory usage (GB) compression none	generic	hybrid	obj-mono	obj-rec	ratio best	ratio worst
pt_rwlock	10.7 M	67.9	**0.88**	0.93	0.92	0.94	77.2	72.2
pt_barrier	128.5 M	> 825.4	**5.48**	9.00	8.98	9.27	150.5	89.0
collision	3.0 M	47.6	0.64	**0.63**	0.64	0.64	75.3	74.1
elevator2	33.0 M	> 342.8	2.50	1.93	**1.90**	1.90	180.3	137.4
lead-uni_basic	19.2 M	232.0	**0.81**	1.30	1.30	1.30	288.1	178.3
lead-uni_peterson	12.2 M	146.4	**0.64**	1.03	1.03	1.03	229.6	142.2
hashset-2-4-2	6.7 M	133.3	1.20	**1.15**	1.15	1.16	116.1	111.1
hashset-3-1	626.9 M	> 15109.8	**27.51**	31.96	31.55	31.44	549.1	472.7

Table 4. Total resident memory used for LLVM models, without and with compression with different splitters.

Name	# of states	average state memory (B) compression none	generic	hybrid	obj-mono	obj-rec	ratio best	ratio worst
pt_rwlock	10.7 M	6807	**88**	92	91	94	77.2	72.2
pt_barrier	128.5 M	> 6900	**45**	75	75	77	150.5	89.0
collision	3.0 M	17119	229	**227**	231	229	75.3	74.1
elevator2	33.0 M	> 11130	81	62	**61**	61	180.3	137.4
lead-uni_basic	19.2 M	12966	**45**	72	72	72	288.1	178.3
lead-uni_peterson	12.1 M	12926	**56**	90	90	90	229.6	142.2
hashset-2-4-2	6.7 M	21283	191	**183**	184	184	116.1	111.1
hashset-3-1	626.9 M	> 25879	**47**	54	54	53	549.1	472.7

with compression. This bound therefore does not include any overheads of the verification algorithm.

generic Compression with a generic splitter which decomposes a state vector into a balanced binary tree with fixed-sized leaves.

hybrid Compression with a splitter that decomposes a state vector according to the top-level structure of the state vector. The splitter is aware of global symbols, heap, and thread stacks. These chunks are further split in a generic way.

obj-mono An extension of the hybrid approach which further decomposes the state vector, respecting boundaries of smaller objects (individual variables, stack frames and so on). This splitter does not decompose any large individual objects.

obj-rec An extension of the *obj-mono* approach that also allow for decomposition of large objects (> 40 bytes) in a binary fashion.

From the aforementioned tables, the following conclusions can be drawn: tree compression offers excellent savings for LLVM models, providing up to several orders of magnitude decrease in memory requirements. This enables verification

Table 5. Total resident memory used for Timed Automata models, without and with compression.

Name	# of states	memory usage (GB) compression			average state memory (B) compression			ratio	
		none	custom	generic	none	custom	generic	best	worst
fischer9_ltsm	0.56 M	0.86	**0.11**	0.13	1656	**212**	249	7.8	6.6
fischer9	0.56 M	0.86	**0.11**	0.13	1656	**211**	249	7.8	6.6
fischer10	2.5 M	4.40	**0.26**	**0.26**	1892	**113**	**113**	16.6	16.6
fischer11	11.1 M	23.2	**1.15**	1.40	2243	**110**	135	20.2	16.6
fischer12	48.8 M	> 119	**4.23**	**4.23**	> 2618	**93**	**93**	28.0	28.0
train-gate9	6.5 M	3.26	**0.91**	1.03	535	**149**	169	3.6	3.2
train-gate10	65.4 M	36.8	**5.94**	11.14	604	**97**	182	6.2	3.3

of models which would be otherwise intractable on any realistic hardware[4]. Furthermore, with the exception of `hashset-3-1`, all of the measured compressed state-spaces can be efficiently verified using a high-end laptop. This is a significant improvement over a dedicated multi-socket computer for verification of the same models that would be needed otherwise (without compression).

Even more significant is the observation that memory requirements per state decrease as the number of states increases, and that they seem to converge to approximately the same number independent of state vector size: even though `hashset-3-1` has almost 4 times larger state vector then `pthread_barrier`, its states are compressed into almost the same size.

Finally, we observe that the effect of advanced splitting algorithms on memory efficiency is mostly negative for LLVM models, even though the achieved compression ratios are still very good in those cases.

Table 5 shows compression results for UPPAAL Timed automata models, using a custom and a generic state vector splitter. The generic version is modelling-language-agnostic and therefore the same as in case of LLVM models. The custom splitter uses a technique similar to the hybrid approach in LLVM. For UPPAAL models, the achieved compression ratios are much lower, but still a significant reduction is obtained. Furthermore, we can see that in this case a custom splitter can significantly improve compression ratio.

5 Conclusions

We have presented a scheme for compressing state vectors in an explicit-state model checker geared towards verification of C and C++ programs. The main contribution of our work is a very efficient scheme for allocating memory and its novel combination with a tree compression scheme. Our approach builds on

[4] If we extrapolate from the biggest model, `hashset-3-1`, we can estimate maximum tractable state space size to be over 40 billion vertices considering high-end server with 2TB of RAM, this could result in around 950 TB of raw uncompressed state space.

earlier solutions but mitigates many of their limitations. The presented scheme is very flexible and offers excellent compression ratios (up to 500×) at a very modest performance penalty. Our tool, building on the presented approach, is realistically capable of exploring on the order of tens of billions of states using commercial, off-the-shelf hardware. Moreover, this number discounts the savings from τ+reduction which alone offers a 50-1000× saving (depending on the model, larger state spaces usually benefit more), together approaching the equivalent of 10^{12} unreduced, uncompressed states (or, considering an average state size of 12 kilobytes, the equivalent of 10000 terabytes of memory).

This represents a considerable improvement in our ability to verify real-world code. With the addition of sufficient parallelism into the mix, very realistic programs can be model-checked in reasonable time and memory using explicit-state techniques. Just as importantly, those advances benefit not only verification of big problem instances on big hardware, but also considerably expands what can be verified using your laptop. In the course of development of DIVINE itself, we increasingly rely on model checking the source code of its components to ensure their correctness. We are quite happy to report that this approach to software development is quickly becoming viable.

References

1. J. Barnat, L. Brim, V. Havel, J. Havlíček, J. Kriho, M. Lenčo, P. Ročkai, V. Štill, and J. Weiser. DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs. In *Computer Aided Verification (CAV 2013)*, volume 8044 of *LNCS*, pages 863–868. Springer, 2013.
2. S. Blom, B. Lisser, J. van de Pol, and M. Weber. A Database Approach to Distributed State Space Generation. *Electronic Notes in Theoretical Computer Science*, 198(1):17–32, 2008.
3. J. Geldenhuys, P. de Villiers, and J. Rushby. Runtime Efficient State Compaction in SPIN. In *Theoretical and Practical Aspects of SPIN Model Checking*, volume 1680 of *LNCS*, pages 12–21. Springer, 1999.
4. G. J. Holzmann. State Compression in SPIN: Recursive Indexing And Compression Training Runs. In *The International SPIN Workshop*, 1997.
5. G. J. Holzmann, P. Godefroid, and D. Pirottin. Coverage Preserving Reduction Strategies for Reachability Analysis. In *PSTV*, pages 349–363, 1992.
6. A. Laarman, J. van de Pol, and M. Weber. Parallel Recursive State Compression for Free. In *SPIN*, pages 38–56, 2011.
7. D. Peled. Ten Years of Partial Order Reduction. In *Proceedings of the 10th International Conference on Computer Aided Verification*, pages 17–28. Springer-Verlag, 1998.
8. P. Ročkai, J. Barnat, and L. Brim. Improved State Space Reductions for LTL Model Checking of C & C++ Programs. In *NASA Formal Methods (NFM 2013)*, volume 7871 of *LNCS*, pages 1–15. Springer, 2013.
9. V. Štill. State Space Compression for the DiVinE Model Checker, 2013. Bachelor's thesis, Faculty of Informatics, Masaryk University Brno.
10. J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *Information Theory, IEEE Transactions on*, 23(3):337–343, May 1977.