

External Program Model Checking

Stefan Edelkamp, Shahid Jabbar,
Dino Midzic, Daniel Rikowski, and Damian Sulewski

Computer Science Department
University of Dortmund
Otto-Hahn Straße 14

Abstract. To analyze larger models for model checking, external algorithms have shown considerable success in the verification of communication protocols. This paper applies external model checking to software executables. The state in such a verification approach itself is very large, such that main memory hinders the analysis of larger state spaces and calls for I/O efficient exploration algorithms. We propose a general state expanding algorithm based on a search tree skeleton with outsourced states. External collapse traded time for space. Additionally, heuristics accelerate the search process, guide it towards the error and shorten the length of the counterexample. Different caching and exploration strategies are evaluated. We found a counterexample in a C++-program for the Dining Philosophers' problem with 300 philosophers in a successful exploration lasting for over 100 hours while consuming 0.5 gigabytes RAM and 19 gigabytes hard disk.

1 Introduction

Model checking is a formal verification method for state based systems, which has been successfully applied in various fields, including process engineering, hardware design and protocol verification. Recent applications of model checking technology deal with the verification of software implementations (rather than checking a formal specification). The advantage of this approach is manifold when compared to the modus operandi in the established software development cycle. For safety-critical software, the designers would normally write the system specification in a formal language like Z [35] and (manually) prove formal properties over that specification. When the development process gradually shifts towards the implementation phase, the specification must be completely rewritten in the actual programming language (usually C++). On the one hand, this implies an additional overhead, as the same program logic is merely reformulated in a different language. On the other hand, the re-writing is prone to errors and may falsify properties that hold in the formal specification.

Modern software model checkers rely on the extension or implementation of architectures capable of interpreting machine code. These architectures include virtual machines [38] and debuggers [26]. Such unabstracted software model checking does not suffer from any of the problems of the classical approach.

Neither the user is burdened with the task of building an error-prone model of the program, nor there is a need to develop a parser that translates (subsets of) the targeted programming language into the language of the model checker. Instead, any established compiler for the respective programming language can be used.

Given that the underlying virtual machine works correctly, we can assume that the model checker is capable of detecting all errors and that it will only report real errors. Also, the model checker can provide the user with an error trail on the source level. Not only does this facilitate to detect the error in the actual program, the user is also not required to be familiar with the specialized modeling languages, such as Promela. As its main disadvantage, unabstracted software model checking may expose a large state description, since a state must memorize the contents of the stack and all allocated memory regions. As a consequence, the generated states may quickly exceed the computer's available memory. Moreover, larger state descriptions slow down the exploration. Therefore, the most important topic in the development of an unabstracted software model checker is to devise techniques that can handle the potentially large states.

The list of techniques for state space compression is long (cf. [3]): *partial-order reduction* prunes the state space based on stuttering equivalences tracking commuting state transitions, *symmetry detection* exploits problem regularities on the state vector, *binary state encoding* allows to represent larger sets of states, *abstraction methods* analyze smaller state spaces inferred by suitable approximations, and *bit-state hashing* and *hash compaction* compress the state vector down to a few bits, while failing to disambiguate some of the synonyms. One important aspect (not mentioned in [3]) are *search heuristics* that guide the search process to the location of the error. Such directed model checking approaches have shown significant advances in the verification of communication protocols [7], selective mu-calculus [32], Java programs [10] and computer hardware [2].

But even with refined exploration techniques, model checking is bounded by the main memory resources. Several memory-limited model checking algorithms have been developed, e.g. [8, 12, 17] but still the core limitation hurdles the model checking of large programs. Infact, the use of virtual memory as a remedy to this problem can instead slow down the performance significantly. Since a general purpose virtual memory scheme, as is offered by many operating systems, does not know anything about the future memory accesses in a model checking algorithm, excessive page faults are inevitable.

External memory algorithms [31] use secondary storage devices, such as hard disk, to solve *large problems* - the problems that have a space requirement larger than the main memory available. They differ from the virtual memory scheme and are more informed about the future access to the data. Milestones for external model checking are [36, 21]. External search algorithms have also shown remarkable performance in the large-scale analysis of games [20], where external breadth-first search has fully explored a single-agent challenge using 1.4 terrabytes hard disk space. Recently, this form of large-scale exploration has been ported to enhance (directed) model checking. In [15] an external and

guided derivative of the explicit-state model checker SPIN has been introduced. As external model checking refers to the independent exploration of set of states the algorithms have been successfully parallelized [16], showing an almost linear speed-up. Based on the work of [33] this large-scale model checking algorithm has been extended to the validation of LTL properties [5].

For the purpose of this paper, we consider the external exploration in model checking unabstracted programs on the object-code level. As systems states for program executables are less accessible and highly dynamic, the algorithmic approach deviates considerably from previous work. The paper is structured as follows. First we introduce to program model checking on the object-code level and to the system states of a program. Next we turn to large-scale model checking and its limitation for the validation of programs. Then we propose the framework for external program model checking that we have developed. It consists of a compromise between main and external memory usage based on an annotated search tree skeleton. For the presentation of the approach we consider the distributed storage of states. Afterwards, we turn to the implementation of an external C++ model checker and present obtained time and space trade-offs in the experiments.

2 Object-Code Program Model Checking

The state of a computer program consists of static components, such as the global variables, as well as dynamic components, like the program stack and the pool of dynamically allocated memory.

Figure 1 shows the components that form the state of a concurrent program for object-code model checking. As model checking is particularly interesting for the verification of concurrent programs, the state description include all relevant information about an arbitrary number of running processes (threads). Threads may claim and release exclusive access to a resource by *locking* and *unlocking* it. Resources are usually single memory cells (variables) or whole blocks of memory. The state description include information about the location and size of dynamically allocated memory blocks, as well as the allocating thread. The memory is divided in three layers: The outer-most layer is the physical memory which is visible only to the model checker. The subset *VM-memory* is also visible to the virtual machine and contains information about the main thread, i.e., the thread containing the main method of the program to check. The program memory forms a subset of the VM-memory and contains regions that are dynamically allocated by the program. Before the next step of a thread can be executed, the content of machine registers and stacks must refer to the state immediately after the last execution of the same thread, or if it is new, directly after initialization. The *memory-pool* is used to manage dynamically allocated memory. The *lock-pool* stores information about locked resources.

An increased difficulty of program model checking the object code lies in the handling of untyped memory as opposed by Java, where all allocated memory can

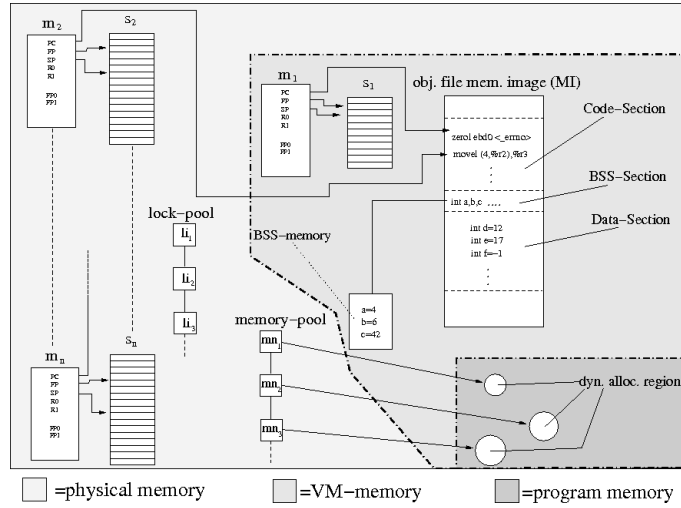


Fig. 1. System state of a program.

be attributed to instances of certain object types. Also, a standardized interface for multi-threading frequently does not exist.

3 External Model Checking

External model checking algorithms explicitly manage the memory hierarchy and can lead to substantial speedups compared to caching and pre-fetching heuristics of the underlying operating system, since they are more informed to predict and adjust future memory access.

External guided model checking refers to early results of externalizing the AI search algorithm A* [28] for optimal plan-finding in single-agent challenges. A* is a single-source shortest-path implicit graph algorithm with included estimate costs. When ran on virtual memory, A* becomes I/O bound due to excessive page faults. External A* [6] operates on a matrix of files (buckets) that is addressed by the generating path length and the heuristic estimate. Each access to a file is sequential and buffered. Duplicate elimination is delayed [19]. Based on External A*, in [15] an experimental explicit-state model checker for safety properties has been implemented on top of SPIN [13], parallelized [16] and extended to LTL properties [5]. As designed for error detection, all external algorithms operate

on-the-fly. The search for safety properties is shown to be I/O optimal¹, while for LTL properties the exact I/O complexity is still open².

Different to the approach presented below, the externality applied to traditional model checking is *strict*, so that main memory resources remain constant during the verification run. The set of states in the current bucket and the set of generated states are all contained on disk and streamed during bucket expansion. This allows to pause and resume the exploration at any given state. However, storing all states in program model checking individually on disk, challenges existing hardware resources.

Exploiting redundancies using a symbolic representation with decision diagrams has been successfully applied and externalized in the context of AI planning [4]. Each bucket in External A* is represented by and stored as a BDD. On the other hand, the highly dynamic structure of states in program model checking does not suggest the usage of BDDs. Therefore, we have relaxed the requirement of a constant main memory.

4 The Algorithm

The general state-expanding algorithm we propose is based on the idea of *mini-states*. Essentially, a mini-state is a pointer to a full system state residing on the secondary memory. A mini-state consists of the hash value of its corresponding state, a pointer to the state - in the form of a file pointer, and its predecessor information to reconstruct the solution path. Additional information includes its depth and its heuristic estimate to the target state. All in all, a mini-state has a constant size in contrast to a state that can change its size due to dynamic memory allocation.

For the sake of brevity, we restrict to properties that can be validated by looking at a state. Recall that in general state-expanding algorithms, full states have to be accessed either to get explored or to be referred to for duplicate detection.

4.1 Externalization

In a search algorithm a full state is only needed in two scenarios: expansion and duplicate detection. Exploiting the idea of mini-states, we propose to perform the search on a tree skeleton defined on the mini-states, while actual states reside on the secondary memory. A request for expansion now reads the state from the disk based on the file pointer stored in the corresponding mini-state. Once read,

¹ $O(\text{sort}(|E|)/p + \text{scan}(|V|))$ I/Os for an undirected state space graphs with consistent estimate, where $|V|$ and $|E|$ are the number of traversed states and transitions, p is the number of processors and where $\text{scan}(n)$ and $\text{sort}(n)$ are the I/O complexities to scan and sort n objects.

² The algorithms suggested amounts to $O(\text{sort}(|E|)/p + t \cdot \text{scan}(|V|))$ I/Os for a general state space graphs with consistent estimate, where t is the depth of the solution.

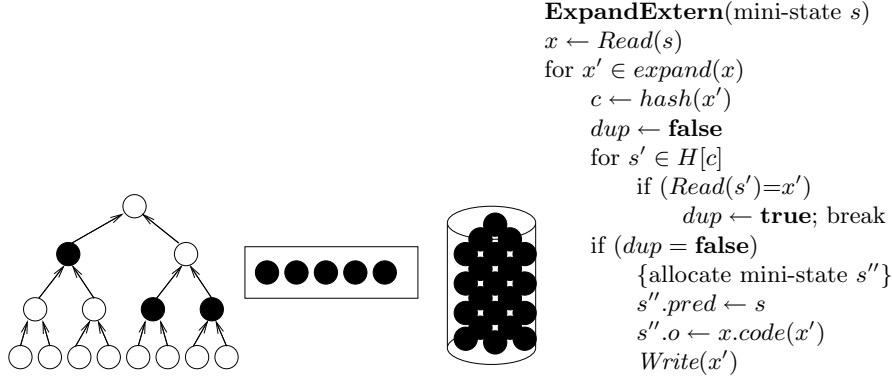


Fig. 2. General state expansion with external storage: Externalization of state in a search tree using a cache and an external state pool (left). Colored nodes state in the cache, hollow nodes illustrate mini-states without any representation in main memory. Pseudo-Code (right)

the state is expanded and its children are again saved in the form of mini-states in the internal memory and as full states on the secondary memory.

Duplicate detection is done based on a hash-table storing only the mini-states. The hash value of a mini-state is the hash value of the full state and is calculated when a state is generated.

In the worst case, we perform one I/O operation for every access to a state. To lessen the average number of I/O operations, we associate an internal cache data structure that allows to retrieve and store in main memory, a small set of states from secondary memory. Though this cache seems very much like virtual memory as offered by almost all operating systems, it can be configured to follow the best replacement strategy suited to the search algorithm. The cache principle is illustrated in Figure 2 (left).

The advantage of external state representation is that we can restore each state that we want from disk, even if it is not in main memory. To do so, we let each mini-state also refers to a file pointer location on disk. In case the efforts for state reconstruction become too large we can change to external states. Once read, the external state becomes full states in the search tree.

The pseudo-code for external search is based on completing mini-states from disk is given in Figure 2 (right). For a mini-state s , $s.o$ denotes the transition (e.g. the sequence of machine instructions), which transforms the predecessor $s.pred$ into s . Similarly, for a full state x , $x.code(x')$ denotes the operation which transforms x to its successor state x' . Note, that transition have a constant-sized representation, which is usually the program counter of a thread running in x . The notion $expand(x)$ refers to the expansion of a full state x and generating a

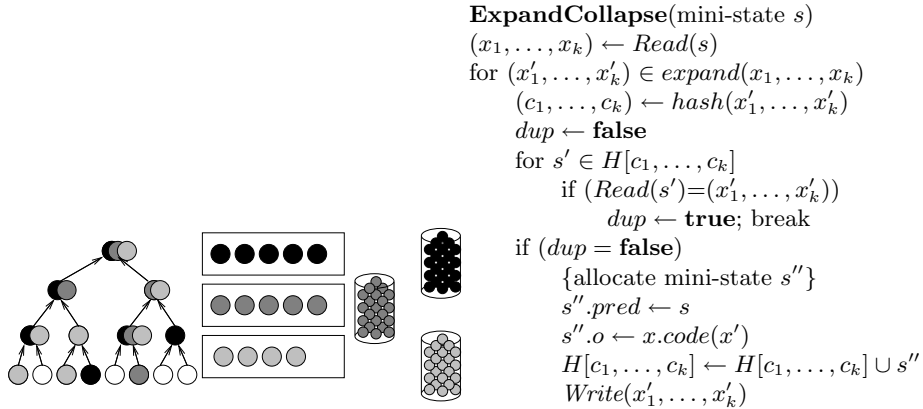


Fig. 3. General state expansion with external storage and collapse compression. Externalization of state in a search tree using caches and distributed state memorization. Colored nodes illustrate partial state information, hollow nodes illustrate mini-states without any information (left). Pseudo-code (right).

list of successors. The hash table H contains the mini-state representatives of all previously generated states.

4.2 External Collapse Compression

Either in main or on secondary memory, storing the entire state information individually is inefficient. A state consists of several parts: stack, memory pool, global variables, etc. A typical transition or a program statement changes only some of them, resulting in several states having common parts. Exploiting these redundancies can lead to a compression scheme where common parts are stored only once.

Collapse compression [14, 22] is a sophisticated approach to store states in an efficient way. Collapsing is based on the observation that although the number of distinct states can become very large, the number of distinct parts of the system are usually smaller. These parts of the state can be shared across all the visited states that are stored, instead of storing the complete encoding of state every time a new state is visited. So, different components are stored in separate hash tables. Each entry in one of the tables is given a unique number. A whole system state is identified by a vector of numbers that refer to corresponding components in the hash tables. This greatly reduces the storage needs for storing the set of already explored states. The principle of collapse compression is illustrated in Figure 3 (left).

Similar to the exposition in [30] we collapse the global store, the memory pool objects, and the threads. These are combined to individual hash addresses for a final encoding. As a hash function for the individual memory regions and stack frames we take $h(v_1, \dots, v_n) = \sum_{i=1}^n v_i \cdot |\Sigma|^i$ modulo a prime q , where Σ is

set of computer characters (usually encoded in form of a byte). The final value $hash(x_1, \dots, x_k)$ as denoted in the pseudo-code (Figure 3 (right)) is in fact a hash function on $h_1(x_1), \dots, h_k(x_k)$.

One advantage of this construction is that the hash function can be computed incrementally, based on the change in the state [25]. For example considering the stack frames, components are added or removed only at the beginning and the end, so that the resulting hash address can be computed incrementally in constant time:

$$h(v_1, \dots, v_n, v_{n+1}) \equiv \sum_{i=1}^{n+1} v_i \cdot |\Sigma|^i \equiv h(v_1, \dots, v_n) + v_{n+1} \cdot |\Sigma|^{n+1} \text{ mod } q$$

$$h(v_1, \dots, v_{n-1}) \equiv \sum_{i=1}^{n-1} v_i \cdot |\Sigma|^i \equiv h(v_1, \dots, v_n) - v_n \cdot |\Sigma|^n \text{ mod } q$$

For the memory pool, we use a balanced AVL [1] tree t with m inner nodes. We define a recursive hash function h' for node N in t as follows: If N is a leaf then $h'(N) = 0$. Otherwise, we set $h'(N)$ to $h'(N_l) + h(v(N)) \cdot |\Sigma|^{|N_l|} + h'(N_r) \cdot |\Sigma|^{|N_l|+|v(N)|} \text{ mod } q$. Here $|N|$ denotes the accumulated length of the vectors in a subtree with root N , and $v(N)$ stands for the subvector associated to N , while N_l , N_r denote the left and right subtrees of N respectively. If R is the root of t , then $h'(R)$ gives the same hash value as h applied to the concatenation of all subvectors in order. For incrementally hash the memory pool, we require logarithmic time.

External distributed state storage, external collapse compression for short, refers to the setting that different entities of state vectors are stored individually on disk. Stacks, memory regions, and individual storage units are maintained in different files, all buffered in cache data structures. This may increase the number of I/Os in case one state item is not contained in the cache but greatly reduces external resources. The pseudo code for external collapse compression is given in Figure 3 (right).

4.3 State Caching Strategies

Memory-limited search has a long tradition in Model Checking. For example, *state-space caching* [12] stores the states in DFS creating a cache of visited states until all main memory resources are exhausted. Many multiple redundant explorations are due to different interleavings of partial orderings of transitions. They have been tackled using different partial ordering methods such as *ample* [3], *persistent* or *sleep* sets [8].

Different to earlier approaches to external exploration we require caching strategies to retrieve/flush states into/from main memory. There are different caching strategies. Some known ones are *Last-In-First-Out (LIFO)*, *First-In-First-Out (FIFO)*, *Least-Recently-Used (LRU)*, *Least-Frequently-Used (LFU)*, *Flush-When-Full (FWF)*, etc. [37]. Different to the operating system we have

the advantage to adapt the caching strategy to the exploration algorithm. Another simple option is a *two-dimensional cache* as an array of fixed length l . The elements of the array are buckets and each bucket can contain k elements. The size of the cache is $m = kl$. Calculating the address for the entries is achieved by a usual hash function. The strategy to delete a node when exceeding the depth k we might consider is *first-in first-out*, such that the first inserted node in turn is replaced. A read failure is given, if the value is not represented within the stack anymore. It is easy to derive that the expected life time for an element is m . Therefore, let $p = 1/l$ be the probability, that a chosen element fits into a bucket and $q = (l - 1)/l$ be its dual probability. Then the random variable X denoting the number of assignments up to the k -th success is negative binomial distributed, such that $P(X = r) = \binom{r-1}{k-1} p^k q^{r-k}$ for $r \in \{k, k+1, \dots\}$. Therefore, $E(X) = k/p = kl$.

As the number of states to be expanded is also limited for AI search, different strategies have been proposed. IDA* [18] invokes a series of cost-bounded DFS searches, but shows exponential behavior in many state-space graphs. MREC [34] is an algorithm, that exploits the entire memory for exploration and reassigns space as needed, propagating cost values upwards. Node caching strategies like SNC [27] introduce randomness to the storage of nodes. The overall probability that a state is cached is $1 - (1 - p)^t$, where $p \in [0, 1]$ is a fixed parameter and t is the number of times that a state is revisited. For $p = 0$ the algorithm correlates with IDA* and for $p = 1$ it matches MREC.

We implemented separate caches, one for the data section, one for the binary section, one for the stack contents and one for the rest of the system state. All of the components can be individually flushed to and read from disk. For the data and binary section we incrementally check at construction time, whether a change has occurred, for the stack we check for redundancies at insertion time. In all three cases, the cache data structure is realized by using an AVL tree sorted by the individual hash addresses. The forth cache is a simple FIFO structure. If a state is generated, we first check by a hash comparison if it is new. If a hash conflict is determined the state is retrieved from the cache (or, if not present, from the hard disk). If the list exceeds a certain predefined value, all elements that are not yet residing on disk are flushed. The last state element is deleted making space for the next one to come.

For external storage we have decided to store system states in blocks of the same size. Each block correspond an own file. The number of elements to be stored should be adjusted that the file size is close to but does not exceed 2 GB, a common file size limit on current computers. If one block gets exhausted, a new file with a rising index file is created. Using number to index the files allows to keep the information overhead in the mini-state at the acceptable level of one integer value.

Philos.	Explored States	Path Length	RAM	Harddisk	Time
Original Implementation					
5	78,173	19	457MB	-	369s
Collapse Compression					
5	78,173	19	233MB	-	346s
Externalization					
6	642,982	22	195MB	568MB	90m
7	546,995	25	233MB	8,6GB	50h

Table 1. Exploration results for breadth-first search.

5 Experiments

We implemented external exploration on top of our tool StEAM [24]. StEAM is an experimental model checker for concurrent C++ programs³.

We draw experiments on a Linux System with 1.7 GHz CPU, 512 MB RAM and a IDE hard disk that was limited to 20 GB.

Our running case study are the Dining Philosophers problem that illustrate the scalability of the approach. With p we denote the number of philosophers. We compare different search methods, namely breadth-first, depth-first and best-first search and different memory saving strategies: incremental state storage as in original StEAM, collapse compression, and externalization. For secondary search all information of a state except the stacks are externalized. The cache sizes are fixed to 1,024 states.

First we consider breadth-first search. Our results are shown in 1. For the original implementation and internal collapse compression we show the largest instance that we could solve with respect to the available memory bound. The result in external search show that with internal collapse compression alone we cannot solve problems with more than 6 philosophers. For external search at $p = 8$ we arrived at the limit of our hard disk capacity, while the internal memory consumption suggest that we can scale higher. We observe that the step-optimal counterexample lies at depth $4 + 3p$, where p is the number of philosophers.

Depth-first search results are depicted in Table 2. The effect is that the exploration can solve by far larger instances. The unfortunate side-effect is that the counterexamples becomes quite lengthy. With the original implementation and internal collapse compression we approached to the limit of main memory with 20 and 25 philosophers, respectively. As we see, externalization allows to scale the problem size to at least twice as many philosophers. As the counterexample for $p = 50$ exceeds the length of ten thousands steps, we expect the burden for

³ The program model checker StEAM including the proposed externalization options is available at <http://sourceforge.net/projects/bugfinder> in Linux packages (.deb and .rpm). The web page also provides information on the tool from an developer and application programmer point of view. Compared to the implementation as provided by [24] the implmentation has been cleaned and documented with *doxygen*. For detecting memory leaks *valgrind* proved to been the most valuable resource.

Philo.	Explored States	Path Len.	RAM	Harddisk	Time
Original Implementation					
20	48,998	3,898	426MB	-	587s
Collapse Compression					
25	76,954	5,123	405MB	-	17m
Externalization					
50	304,929	10,938	343MB	3GB	4h

Table 2. Exploration results for depth-first search.

Philos.	Explored States	Path Len.	RAM	Harddisk	Time
Original Implementation					
50	9,465	353	239MB	-	10m
Collapse Compression					
50	9,465	353	211MB	-	10m
100	36,430	703	566MB	-	160m
Externalization					
150	80,895	1,053	256MB	2.3GB	14h
300	319,290	2,103	545MB	19GB	104h

Table 3. Exploration results for greedy best-first search using most-blocked heuristic.

the application programmer to trace the error to be too large and did not try larger instances.

The results for greedy best-first search are shown in Table 3. As a heuristic we chose the most-blocked heuristic that simply counts the number of process that cannot execute a transition due to existing locks.

As expected, directed search accelerates the exploration enormously while keeping the counterexample length at an acceptable level. With internal collapse compression we slightly exceeded the limit of main memory with 100 philosophers, at which the system started to swap.

Externalization allows to scale the problem size to 300 philosophers. We also measured the number of hard disk accesses. This exploration last for more than 4 days without exceeding the memory available. This indicates that our implementation is almost free of memory leaks. The number of write accesses is 319,291 and the number of read accesses is 7,280. Moreover, the number of items in the stack cache equals 1,203.

In all the above mentioned experiments we used a single file to store all the states. This policy turned out to work only for the domains where there was a good distribution of the hash values and the range of the hash function was large enough to result in few collisions. Such is the case with Dining Philosophers.

While experimenting on a c++ program for solving 8-puzzle instances, we observed that the single file based storage scheme resulted in an increase in time due to large number of I/Os. The reason turned out to be small range of the hash function that results in large number of collisions. Dividing the storage file

Instance	Explored States	Path Len.	RAM	Harddisk	Time	Hashcodes	Collisions
Original Implementation							
207165384	248,243	141	552MB	-	3,933s	2,906	21,538,404
267105384	437,400	150	861MB	-	6,933s	2,938	63,637,143
267150384	607,331	159	1125	-	8,961s	2,997	119,295,029
Collapse Compression							
207165384	248,243	141	294MB	-	3,798s	2,906	21,538,404
267105384	437,400	150	407MB	-	6,776s	2,938	63,637,143
267150384	607,331	159	509MB	-	9,490s	2,997	119,295,029
267154380	840,493	168	648MB	-	13,310s	2,985	222,463,964
267154308	1,405,027	177	974MB	-	21,713s	2,982	598,825,442
Single File Externalization							
207165384	248,243	141	173MB	64MB	4,503s	2,906	21,538,404
267105384	437,400	150	193MB	112MB	9,324s	2,938	63,637,143
267150384	607,331	159	212MB	156MB	14,574s	2,997	119,295,029
267154380	840,493	168	237MB	215MB	23,296s	2,985	222,463,964
Hash File Externalization							
207165384	248,243	141	173MB	71MB	4,422s	2,906	21,538,404
267105384	437,400	150	197MB	123MB	8,553s	2,938	63,637,143
267150384	607,331	159	216MB	167MB	12,955s	2,997	119,295,029
267154380	840,493	168	237MB	228MB	18,479s	2,985	222,463,964

Table 4. Comparison of state storage strategies on 8-Puzzle instances.

according to the hash values solved our problem. For each hashcode that was generated we assigned a new file, where states having that particular hashcode are saved. While resolving a hash conflict we suggest to read the file block-wise in a small internal memory cache. This resulted in an increase in time performance as compare to the single-file based storage where the algorithm has to perform several jumps in the file to resolve a conflict. In Table 4 we show a comparison of different state storage strategies while solving 8-puzzle instances. We compare the performance of this new storage scheme with the single-file based implementation. A gain in time is clearly observable. This new scheme though has to be dealt with care. For problems where the number of unique hashcodes is very large, the I/O performance of the algorithm can infact decrease because of accessing very small files. One should also take care that the number of files could grow more than the allowable limit of the operating system.

6 Conclusion

Tailoring a model checking engine to an existing virtual machine for model checking c++ is a challenging task, that was thought to be infeasible, e.g. by the creators of JPF [38].

With this work we have provide the first implementation of an external program model checker, which does not rely on abstract models. The main difference

to previous attempts in external model checking is that a skeleton of the search tree resides in main memory.

The savings obtained by external exploration are considerable and likely another important step towards practical applicability. Externalization positively combines with directed exploration for accelerated error detection.

Despite considerably long CPU times induced by hard disk access for external exploration in larger problem instances, the exploration efficiencies are still remarkable. To the authors knowledge (and even when equipped with a considerably small hardware resources of about 500 MB RAM and 20 GB hard disk), we could present the largest explorations in program model checking that have been achieved so far. Other program model checkers like VeriSoft are reported to solve the philosophers problems (including state-less search and partial order reduction with persistent and sleep sets) with at most 10 philosophers [8]. Moreover, even directed model checkers like HSF-SPIN that rely on an abstract model and that do not take the burden of exploring the object-code [23] show considerable work in solving larger philosophers problems as even the much simpler structured state vector appears considerably large. For externalizing HSF-SPIN [15], 2.29 GB were reported for $p = 100$ and 10.4 GB for $p = 150$ also using the most-blocked/active-process heuristic.

The approach for analyzing c++ based on model checking the object code is still experimental. While the tool and the virtual machine both compile on gcc 4.0, the compiler for the virtual machine still relies on gcc 2.95. For many input file this is not a limitation as recent developments of the compiler are mostly more restrictive on their inputs. However as the libraries are much different, we plan to extend the virtual machine to work on cross-compiled code such that our model checker can work on different processor models too.

We currently work on data and predicate abstraction [9] to be used in an abstract-refinement loop [11] or for constructing abstraction databases [29]. As abstractions convert a deterministic program into a non-deterministic one, we aim to use our external model checker to explore the abstract models to guide the search in the concrete program. Inspired by XSPIN, a GUI for SPIN model checker, we have also started to develop a GUI for the model checking tool in form of a plugin for Eclipse. So far the frontend can be used to select the parameters of and call the model checker StEAM on the developed sources in the know c++ environment CDT. Moreover, we are able to display the counterexample trail and exploration statistics of the model checker in an XML browser.

References

1. G. M. Adelson-Velskii and E. M. Landis. An algorithm for the organization of information. *Doklady Akademii Nauk SSSR*, 146:263–266, 1962.
2. R. Bloem, K. Ravi, and F. Somenzi. Symbolic guided search for CTL model checking. In *DAC*, pages 29–34, 2000.
3. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, 1999.
4. S. Edelkamp. External symbolic heuristic search with pattern databases. In *ICAPS*, pages 51–60, 2005.

5. S. Edelkamp and S. Jabbar. Large-scale directed LTL model checking. In *SPIN*, 2006.
6. S. Edelkamp, S. Jabbar, and S. Schrödl. External A*. In *KI*, pages 226–240, 2004.
7. S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *STTT*, 5(2-3):247–267, 2004.
8. P. Godefroid. Software model checking: The VeriSoft approach. *Formal Methods in System Design*, 26(2):77–101, 2005.
9. S. Graf and H. Saidi. Construction of abstract state graphs of infinite systems with PVS. In *CAV*, pages 72–83, 1997.
10. A. Groce and W. Visser. Model checking Java programs using structural heuristics. In *ISSTA*, pages 12–21, 2002.
11. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software Verification with BLAST. In *SPIN*, pages 235–239, 2003.
12. G. Holzmann. Tracing protocols. *AT&T Technical Journal*, 64(12):2413–2434, 1985.
13. G. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.
14. G. J. Holzmann. State compression in spin. In *Third Spin Workshop*, Twente University, The Netherlands, 1997.
15. S. Jabbar and S. Edelkamp. I/O efficient directed model checking. In *VMCAI*, pages 313–329, 2005.
16. S. Jabbar and S. Edelkamp. Parallel external directed model checking with linear I/O. In *VMCAI*, 2006.
17. C. Jard and Th. Jeron. Bounded-memory algorithms for verification on-the-fly. In *CAV*, 1991.
18. R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.
19. R. E. Korf. Breadth-first frontier search with delayed duplicate detection. In *IJCAI-workshop: Model Checking and Artificial Intelligence (MoChArt)*, pages 87–92, 2003.
20. R. E. Korf and P. Schultze. Large-scale parallel breadth-first search. In *AAAI*, pages 1380–1386, 2005.
21. L. M. Kristensen and T. Mailund. Path finding with the sweep-line method using external storage. In *ICFEM*, pages 319–337, 2003.
22. F. Lerda and W. Visser. Addressing dynamic issues of program model checking. In *SPIN*, pages 80–102.
23. A. Lluch-Lafuente. *Heuristic Search in the verification of Communication Protocols*. PhD thesis, Computer Science Institute, Freiburg University, 2003.
24. T. Mehler. *Challenges and Applications of Assembly-Level Software Model Checking*. PhD thesis, University of Dortmund, 2006.
25. T. Mehler and S. Edelkamp. Dynamic incremental hashing in program model checking. *ENTCS*, 2005.
26. E. Mercer and M. Jones. Model checking machine code with the GNU debugger. In *SPIN*, pages 251–265, 2005.
27. T. Minura and T. Ishida. Stochastic node caching for memory-bounded search. In *National Conference on Artificial Intelligence (AAAI)*, pages 450–456, 1998.
28. J. Pearl. *Heuristics*. Addison-Wesley, 1985.
29. K. Qian and A. Nymeyer. Guided invariant model checking based on abstraction and symbolic pattern databases. In *TACAS*, pages 497–511, 2004.
30. Robby, M. B. Dwyer, J. Hatcliff, and R. Iosif. Space-reduction strategies for model checking dynamic software. *ENTCS*, 89(3), 2003.

31. P. Sanders, U. Meyer, and J. F. Sibeyn. *Algorithms for Memory Hierarchies*. Springer, 2002.
32. A. Santone. Heuristic search + local model checking in selective mu-calculus. *IEEE Transactions on Software Engineering*, 29(6):510–523, 2003.
33. V. Schuppan and A. Biere. From distribution memory cycle detection to parallel model checking. *STTT*, 5(2–3):185–204, 2004.
34. A. K. Sen and A. Bagchi. Fast recursive formulations for best-first search that allow controlled use of memory. In *IJCAI*, pages 297–302, 1989.
35. J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 1992.
36. U. Stern and D. Dill. Using magnetic disk instead of main memory in the murphi verifier. In *CAV*, pages 172–183, 1998.
37. A. S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, 1992.
38. W. Visser, K. Havelund, G. Brat, and S. Park. Model Checking Programs. In *ICSE*, pages 3–12, 2000.

7 Appendix: c++-Sources Dining Philosophers

Philosopher.h

```
#ifndef PHILOSOPHER
#define PHILOSOPHER

#include "IVMThread.h"

class Philosopher : public IVMThread {

private:
    static int id_counter;
    short * leftfork;
    short * rightfork;

public:
    Philosopher();
    Philosopher(short * leftfork, short * rightfork);
    virtual void start();
    virtual void run();
    virtual void die();
};
#endif
```

Philosopher.cc

```
#include "icvm_verify.h"
#include "IVMThread.h"
#include "Philosopher.h"

class IVMThread;
extern int g[10];

Philosopher::Philosopher(short * lf, short * rf) :
    IVMThread::IVMThread() {
    leftfork=lf; rightfork=rf;
}

Philosopher::Philosopher() {}

void Philosopher::start() {
    run();
}

void Philosopher::run() {
    int i;
```

```

    while(1) {
        VLOCK(leftfork);
        VLOCK(rightfork);
        VUNLOCK(rightfork);
        VUNLOCK(leftfork);
    }
}

void Philosopher::die() {}
int Philosopher::id_counter;

philosophers.c

#include <stdlib.h>
#include <assert.h>
#include "Philosopher.h"

class Philosopher;

Philosopher ** p;
short forks[255];

void initThreads (int n) {
    p=(Philosopher **) malloc(n*sizeof(Philosopher *));
    for(int i=0;i<n;i++) {
        p[i]=new Philosopher(&forks[i], &forks[(i+1) % n]);
        p[i]->start();
    }
}

int main(int argc, char ** argv)
{
    int n;
    BEGINATOMIC;
    if(argc<2) {
        fprintf(stdout, "-----Missing Parameter!-----\n");
        exit(0);
    }
    n=atoi(argv[1]);
    initThreads(n);
    ENDATOMIC;
    return 1;
}

```