Science of Computer Programming 76 (2011) 136–157

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Flash memory efficient LTL model checking☆

S. Edelkamp [a], D. Sulewski [a,*], J. Barnat [b], L. Brim [b], P. Šimeček [b]

[a] Universität Bremen, Germany

[b] Masaryk University Brno, Czech Republic

ARTICLE INFO

Article history:
Received 22 July 2009
Accepted 9 March 2010
Available online 17 June 2010

Keywords:
Model checking
External memory algorithms
Algorithm engineering

ABSTRACT

As the capacity and speed of flash memories in form of solid state disks grow, they are becoming a practical alternative for standard magnetic drives. Currently, most solid-state disks are based on NAND technology and much faster than magnetic disks in random reads, while in random writes they are generally not.

So far, large-scale LTL model checking algorithms have been designed to employ external memory optimized for magnetic disks. We propose algorithms optimized for flash memory access. In contrast to approaches relying on the delayed detection of duplicate states, in this work, we design and exploit appropriate hash functions to re-invent immediate duplicate detection.

For flash memory efficient *on-the-fly* LTL model checking, which aims at finding any counter-example to the specified LTL property, we study hash functions adapted to the two-level hierarchy of RAM and flash memory. For flash memory efficient *off-line* LTL model checking, which aims at generating a minimal counterexample and scans the entire state space at least once, we analyze the effect of outsourcing a memory-based perfect hash function from RAM to flash memory.

Since the characteristics of flash memories are different to magnetic hard disks, the existing I/O complexity model is no longer sufficient. Therefore, we provide an extended model for the computation of the I/O complexity adapted to flash memories that has a better fit to the observed behavior of our algorithms.

1. Introduction

Model checking real-life industrial systems is a memory demanding and computational intensive task. Utilizing the increase of computational resources available for the verification process is indispensable to handle these complex systems. The three major approaches to gain more computational power include the usage of parallel computers, clusters of workstations and the usage of external memory devices, as well as their combination.

In this work, we study two model checking problems: reachability analysis and LTL model checking. The aim of reachability analysis is to find an erroneous state of a system or prove that all states satisfy criteria given in the propositional logic. *Linear temporal logic* (LTL) [55] is a favorite logic for specification of temporal properties of systems. It can express the behavior of whole paths in contrast to properties of single states. The LTL model checking problem can be reduced to the problem of *accepting cycle detection* in the state space of a finite automaton [19], i.e., checking for the presence of a cycle containing at least one accepting state.

☆ This work has been partially supported by the Grant Agency of Czech Republic grant No. 201/06/1338, the Academy of Sciences grant No. 1ET408050503, and DFG grant No. ED 74/8-1.

* Corresponding author.
E-mail address: Damian.Sulewski@TU-Dortmund.de (D. Sulewski).

0167-6423/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2010.03.005

Table 1
Rough classification of flash media devices in form of solid state disks with respect to RAM and magnetic hard disks.

Characteristic	RAM	Solid state disk	Magnetic disk
Volatile	Yes	No	No
Shock resistant	Yes	Yes	No
Storage capacity	Small	Medium	Large
Energy consumption	High	Medium	High
Price	High	Medium	Very cheap
Random reads	Very fast	Fast	Slow
Random writes	Very fast	Slow	Slow
Sequential reads	Very fast	Fast	Fast
Sequential writes	Very fast	Fast	Fast
Throughput	Large	Medium	Small

We focus on external memory devices, for which the access to stored information is orders of magnitude slower then the access to information stored in the main memory. Therefore, the complexity of algorithms is measured in the number of I/O operations [1]. Our goal is to develop algorithms that reduce this number.

1.1. The advent and rise of solid state disks

I/O efficient algorithms reflect the physical properties of external memory devices, i.e., they are designed to minimize expensive random accesses to data in favor of their block processing. However, similar to all the PC components, the external memory devices are continuously developed as well and their properties are improving over time. Recently, flash memory based external memory devices became widely used as the so called *solid state disks* (SSDs).

An SSD is a drive that is electrically, mechanically and software compatible with a conventional (magnetic) hard disk drive, but its storage medium is not magnetic (like a hard disk) or optical (like a DVD) but consists of solid state semiconductor chips. SSDs built upon DRAM chips are a kind of *random access memory* (RAM) extension with similar speed and price. We rather focus on flash-based (NAND EEPROM) drives, because they offer an interesting compromise between speed and price — recently, the price for very fast SSDs has fallen below $3/GB [42], while DRAM typically costs $20/GB or more.

SSDs provide faster access time than magnetic disks, because the data can be randomly accessed and does not rely on a read/write head synchronizing with a rotating disk. For the current qualitative comparison of RAM, magnetic disks and SSDs see Table 1, which is derived from media tested in [3]. Note that SSDs outperform magnetic disks especially in random reading. In other parameters the difference is not significant. The reason why random writing is slower than reading comes from architectural principles of flash memories: in order to write even a single byte, it is necessary to copy an entire block (hundreds of kB) into a cache, modify the data, and write the whole block, while reading can be performed immediately. Thus, the difference between random reading and writing is fundamental for flash-based SSDs and it cannot be changed without replacing the technology completely.

Unlike its magnetic counterpart, an SSD does not rely on physical movements of the head(s) to access the data, so that their access time is much shorter. For example, the speed of random reads for a solid state disk built with NAND flash memory lies roughly at the geometric mean of the speeds of RAM and magnetic *hard disk drive* (HDD) [51]. The only factor limiting solid state disks from being massively spread is the cost of the device if expressed per stored bit. This cost is still significantly higher for SSDs than for magnetic disks. However, it is definitely subject to change in the future.

The progress within the last two years confirms that SSD transfer rates have risen and access times have decreased, but random writing has stayed substantially slower than reading for all SSD models.[1]

1.2. External memory LTL model checking

In model checking, a graph is usually *implicit*, i.e., it is defined by initial states and a successor function. This means that graph algorithms have to generate the graph as they traverse it. E.g., when a graph traversal algorithm needs to proceed to an immediate successor s' of a state s, it computes state s' from the state vector of s using a state generation function. To prevent re-visiting of already explored states, all states that have been processed are stored in the memory. Hence, whenever a state is generated, it is first checked against the set of stored states to learn whether it is a new state or has been visited before. In the context of I/O efficient algorithms, this check is referred to as *duplicate detection*.

Disk-based algorithms have been studied in the context of formal verification, model checking [19] in particular, as one of the techniques to fight the well-known *state explosion problem* for more than a decade [61]. In this article we focus on

[1] Recently, SSD manufacturers have begun to face slow random writes by large DRAM caches [41]. To attract customers, they provide biased write access times in specifications of their drives, which are unrealistically improved by orders of magnitude by the caches under optimistic testing scenarios. More realistic write access times can be estimated from another specification parameter called *Write IOPS* (input/output operations per second).

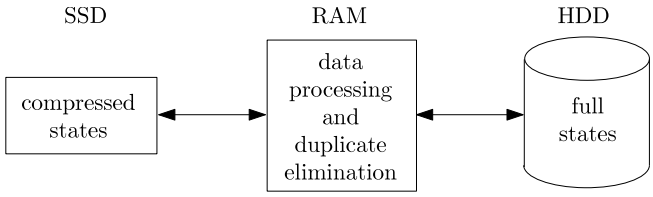

Fig. 1. Memory architecture with solid-state and hard disk.

enumerative LTL model checking, which is the standard option for analyzing software systems. *External memory LTL model checking* refers to the use of external storage for analyzing state spaces that are too large to be kept in RAM. One of our main research goals is to consider a simple question that comes up with the advent of solid state disks. Namely, if it is meaningful to design new I/O algorithms for LTL model checking that take advantage of the fast random reads of a solid state disk, or if it is satisfactory to apply existing I/O efficient LTL model checking algorithms for SSDs.

In *semi-external LTL model checking* invented by Edelkamp et al. [29], an algorithm is allowed to allocate a constant number of bits per state in RAM. It achieves a different time/space trade-off compared with external model checking. The approach utilizes the fact that, given a *perfect hash function* for a set of all states, Courcoubetis et al.'s *nested depth-first search* (NDFS) *on-the-fly* model checking algorithm [20] stores only 2 bits per state in RAM. The overhead for practical perfect hash function construction algorithms [14,15,11] is to generate the entire state space first. The RAM requirements are a small constant of bits per state.

In contrast to on-the-fly model checking, semi-external model checking is naturally an *off-line* method, because all states have to be generated before the search for a counterexample starts. Semi-external versions of *double depth-first search* (DDFS) also presented in [20] show advances especially for state spaces containing no counterexamples [29]. Moreover, a combination with an iteratively deepening bounded search is proposed in order to find short counterexamples before the entire state space is generated.

The memory hierarchy that we mainly assume for our model checking algorithms is displayed in Fig. 1. Full state information is stored on a magnetic disk, while compressed information is made available on the SSD, the information on the HDD is mapped to the SSD. *Immediate duplicate detection* (IDD), meaning, the algorithm checks whether a state has already been visited immediately after generating it, is made possible using a bit-vector hash-table, possibly together with a memory-based hash function in the RAM. For collision resolution, if required, the SSD will be queried to compare the stored representation with the generated one.

1.3. Contributions and structure

This article merges and extends the work of two precursing conference papers. The first one [5] answers the question if flash memory can help in model checking and considers on-the-fly LTL model checking using solid state disks. The second one [31] considers flash-efficient LTL model checking with minimal counterexamples. Besides providing more background material the article contains the following research results.

- In order to analyze the performance of the algorithms on flash devices, we contribute a complexity model that is able to express the fact that SSDs have fast random read operations while random writes are still slow. We slightly extend the traditional I/O model for magnetic disks of Aggarwal and Vitter [1] with a parameter p that represents a penalty factor for random write operations. We consider this extension to be sufficient and simple enough, allowing us to derive I/O complexities of our algorithms and predict their practical performance.
- To answer the question of immediate duplicate detection for on-the-fly model checking, we design several techniques to implement a flash memory efficient graph traversal procedure. Namely, we discuss several variants of a hashing mechanism that is used, e.g., by the *nested depth-first search algorithm* [20,39] to efficiently identify already generated states during the graph traversal. We also report on a preliminary experimental comparison of newly suggested flash memory efficient techniques with standard I/O efficient ones, and discuss the impact of possible technology improvements that may emerge in the future.
- We implement and refine the RAM-efficient (and not previously implemented) algorithm of Gastin and Moro [34] to find counterexamples of minimal length. Compared to another external memory optimal counterexample generating algorithm [25], our proposal serves a stronger optimality criterion and has a better worst-case I/O performance. We additionally show that the number of bits per state in RAM can be reduced by moving the hash function to an SSD. Let c_{PHF} be the number of bits required (per stored state) to represent the hash function. If the hash function is stored on the flash memory, the RAM requirements for detecting duplicates can be reduced from $1 + c_{PHF}$ bits per state to 1.

This article is organized as follows.

- Section 2 introduces the broadly accepted external memory model of Aggarwal and Vitter. To derive the I/O efficiencies of flash memory devices we modify it, introducing a penalty parameter for random writing.

- In Section 3 we recall I/O efficient techniques used for enumerative external memory LTL model-checking. We distinguish between reachability analysis and LTL model checking, pointing out the main aspects of both.
- In Section 4 we consider model checking with SSDs and study several flash memory efficient hashing techniques that prefer sequential write and random read operations. We then show how the modifications can be used to design a flash memory efficient nested depth-first search algorithm for on-the-fly model checking.
- Section 5 addresses semi-external LTL model checking using perfect hash functions. It first recalls the definition of semi-externality that has been adapted to enumerative graph search in model checking. Then, after introducing the basics and recent advances in perfect hashing, it reviews the improvements of nested and double depth-first search obtained with this approach. As an intermediate step, we exploit flash memory by exporting the perfect hash function that may have become too large for the RAM to the SSD.
- In Section 6 we implement the internal sparse-memory minimal counterexample LTL model checking algorithm of Gastin and Moro [34], adapt it to run semi-externally and run it on flash memory. Their pseudo-code algorithm contained minor bugs, which were removed in our presentation. Moreover, we employ a 1-level bucket implementation of the priority queue.[2]
- Section 7 reports on our experimental evaluation of I/O efficient techniques optimized for both magnetic and solid state disks. We start with our improvements for on-the-fly model checking comparing the different strategies we have suggested. We then provide experiments to study the impact of flash memory semi-externally searching minimal counterexamples.
- In Section 8 we conclude the article, discuss what impact possible predicted technological improvements may have, and indicate future research avenues.

2. Towards a complexity model for flash memory devices

Existing two-level memory hierarchy models fail to realize the full potential of flash-based storage devices. In this section we propose a modified version of the standard I/O model, which reflects the capabilities of solid state disk storage devices and still remains tractable. Our experiments will show that the theoretical analysis of algorithms on our models maps to the empirical behavior of algorithms when using solid-state disks as external memory.

2.1. Standard I/O complexity model

A widely accepted model for the analysis of the complexity of I/O algorithms is the model of Aggarwal and Vitter [1], where the complexity of an algorithm is measured in the number of external I/O operations. This is motivated by the fact that a single I/O operation is slower by approximately six orders of magnitude than a computation step performed in the main memory [63]. Therefore, an algorithm that does not perform the optimal amount of work but has a lower I/O complexity may be faster in practice compared to an algorithm that performs the optimal amount of work but has a higher I/O complexity.

The complexity of algorithms in the model of Aggarwal and Vitter is parametrized by M, B, and D, where

- M denotes the number of items that fit into the internal memory,
- B denotes the number of items that can be transferred in a single I/O operation, i.e., the block size and
- D denotes the number of blocks that can be transferred in parallel, i.e., the number of independent parallel disks available.

An item is an abstract measurement, it can resolve, e.g., to bytes or state vectors.

The abbreviations $sort(n)$ and $scan(n)$ stand for $O(N/(DB)\log_{M/B}(N/B))$, the known I/O complexity of sorting n items on external memory and $O(N/(DB))$, the known I/O complexity of scanning n items in external-memory, respectively.

2.2. Modified I/O complexity model

For solid state disks, the model by Aggarwal and Vitter is no longer valid, since it does not cover the different access times for random read and write operations. For solid state disks we propose to extend the model of Aggarwal and Vitter with a penalty factor p for random write operations.

The computational model we propose shares similarities to the *general flash model* in [2], in which different access times for reading and writing are proposed, but our extension with one additional parameter for writing penalty (instead of two block size parameters) is more conservative and easier to apply. In contrast to [17], where the authors describe the process of automating part of a hand-crafted Z model of NAND flash memory, using the Z/Eves theorem prover, we aim at developing a general and simple I/O complexity model. In order to reflect a gap between random reading and writing, our model adds only one parameter p denoting a penalty for each random write operation. When analyzing the I/O complexity of an algorithm, it is only required to distinguish random write operations and multiply their count by p.

[2] We omitted to repeat their insightful correctness proofs, so that the presentation of the algorithm appears quite compact. We refer the reader to the original article for a more detailed treatment.

The *unit-cost flash model* also contributed in [17] is the general flash model augmented with the assumption of an equal access time per element for reading and writing. This simplifies the model considerably, since it becomes significantly easier to adapt external-memory results. Compared to our model, the authors require two parameters (block size for reading and block size for writing), which both depend on the particular hardware used. It appears easier to transfer the single penalty parameter p between different devices.

Our model is also general in the sense that for $p = 1$ we get back the original complexity model of Aggarwal and Vitter. The value of p depends on a specific device, but, for simplicity, we assume that $p < N/B$, where N is the size of an input — thus, if one random write follows a sequence of N/B write operations, no element with a penalty factor is inferred in the asymptotic I/O complexity, as the random writing operation is amortized for the previous writing of blocks. The assumption $p < N/B$ is reasonable, since for typical p values N would have to be very small to break this assumption and such an input would surely fit in the RAM easily.

3. External memory model checking with magnetic disks

Due to the huge number of states, their large size, and the increasing speed of generating them, the memory demands while analyzing systems rise rapidly. In order to release memory, states stored in the set of visited states have to be fully or partially flushed to the external memory. Under these circumstances a check, whether or not a state has been visited, may involve an I/O operation as not only the states stored in internal memory, but also the states stored on the external memory device must be considered. This, however, renders a standard graph traversal algorithm inefficient as the I/O operation is slower by orders of magnitudes compared to a single or several reads from the internal memory.

The portfolio of disk-based search techniques [26] in verification includes multi-threaded C++ programs [27], parallel I/O efficient state space generation [9] and cluster-based I/O efficient model checking [8].

3.1. External memory reachability analysis

Reachability analysis amounts to searching the entire state space in an arbitrary order until an erroneous state is found. Therefore, the main task is to traverse a graph representing the state space. The core technique that gave birth to I/O efficient graph algorithms is the so-called *delayed duplicate detection* (DDD) [44,45,52,61], whose idea is to postpone the individual checks against the set of visited states and perform them together in a group thus amortizing the cost of I/O operations.

The worst-case I/O complexity for external memory breadth-first (and its directed variant external memory A* [37,28]) search in a graph $G = (V, E)$ is $O(l \cdot scan(|V|) + sort(|E|))$, where l is the length of the largest back edge in the breadth-first search graph. More precisely, the locality l (for the breadth-first search graph) is defined as

$$l = \max\{layer(s) - layer(s') + 1 \mid s, s' \in V; (s, s') \in E\},$$

where $layer(s)$ denotes the depth of s in the breadth-first search graph.

There are other techniques that have significant impact on the performance of an I/O efficient graph traversal algorithm. For example, it is possible to perform hash compaction or compression of states to be stored, which results in a smaller amount of data to be transferred between external and internal memory. Another quite successful improvement builds upon using a Bloom filter maintained in the main memory in order to reduce unnecessary I/O operations. Also simple partitioning of states stored on external memory may have an impact on the performance of an I/O efficient graph traversal procedure. For more details on these techniques we kindly refer the reader to [36]. Other delayed duplicate detection techniques are layered duplicate detection by Lamborn and Hansen [46] and dynamic delayed duplicate detection by Evangelista [32].

As mentioned above, an important aspect of an I/O efficient algorithm is that the data stored on external memory is accessed in blocks. While the clever implementation techniques aim at reducing the number of I/O operations, or reducing the amount of data being transferred, there is also the possibility to improve the performance of an I/O efficient algorithm by simple improving the performance of an I/O operation. For example, by connecting two identical external memory devices into a mirror RAID array we can almost double the bandwidth for reading the block of data from the external memory device. Note that this approach basically improves bandwidth only while it does not influence the latency, i.e., the time needed to read the first bit.

Similarly, it is possible to reduce time needed for solving the problem if instead of the serial I/O efficient algorithm working over a single external device a parallel I/O efficient algorithm is used utilizing multiple processors and external memory devices [9]. This is, however, possible only if the algorithm involved allows parallel processing, which is, e.g., the case for breadth-first search, but is not the case for depth-first search.

3.2. External memory LTL model checking

For accepting cycle detection there is the space- and time-optimal NDFS algorithm [39]. Unfortunately, the algorithm becomes rather inefficient, as soon as states to be stored cannot be maintained in the main memory [4,25]. Moreover, counterexamples are usually not of minimal length.

Three different I/O efficient algorithms for solving the LTL model checking problem have been published [25,7,10]. The first I/O efficient optimal solution for the LTL model checking problem [25] builds on the reduction of *liveness-to-safety* property conversion [58] that originally has been designed for symbolic model checking [48]. Then, safety properties can be verified using simple reachability analysis (e.g., in breadth-first order). The algorithm was further improved by using the directed A* search and parallelism. Since the reduction to the reachability relation testing may result in an up to quadratic increase in the space complexity, this algorithm should be rather viewed as a tool for bug hunting.

A new I/O efficient algorithm for LTL model checking was given in [7]. The algorithm avoids the expensive increase in the space complexity, but does not work on-the-fly, which means that the full state space must be generated and stored on the external memory device before it is checked for the presence of an accepting cycle. This disadvantage makes the algorithm quite inefficient in the cases where an error can be discovered quickly using some on-the-fly algorithm. Finally, the algorithm given in [10] is both on-the-fly and linear in the space requirements with respect to the size of the state space.

Another I/O efficient algorithm for accepting cycle detection is *one-way-catch-them-young* (OWCTY) [18]. This algorithm generates the whole state space and then iteratively prunes parts that do not lead to any accepting cycle.

Later on, an external on-the-fly LTL model checking algorithm based on the *maximal-accepting-predecessors* (MAP) algorithm [16] has been developed [10]. It additionally avoids scans of previous layers for duplicate detection, especially in large search depths.

4. External memory model checking with solid state disks

In this section, we investigate whether relatively fast random read operations allow one to design substantially faster I/O efficient algorithms than those previously optimized for magnetic disks. First of all, we tried to dispose of DDD, since it is the main source of I/O operations in classical I/O efficient graph algorithms. Instead, by exploiting fast random reads we implement IDD. Thus, we devised and evaluated three strategies implementing IDD with use of SSDs and compared the fastest one to DDD-based algorithms. All three strategies implement IDD by hashing to a list or a table of visited states stored on SSD.

First, we study direct access to the solid state disk without exploiting RAM usage. This implies both random read and random write operations. The implementation serves as a reference, and can be scaled to any implicit search with a visited state space that fits on the solid state disk.

Next, we compress the state in internal memory to include the address on external memory only. In this case states are written sequentially to the external memory in the order of generation. For resolving hash synonyms, random reads are needed for the lookup of states. Even though linear probing shows performance deficiencies for internal hashing, for block-wise strategies, it is the apparent candidate.

The third option supports flushing the internal hash table to the external device, once it becomes full. In this case, full state vectors are stored internally. For large amounts of external memory and small vector sizes, large state spaces can be looked at. Usually the exploration process is suspended while flushing the internal hash table. We observe different trade-offs for the amount of randomness for external readings and writings, which mainly depend on increasing the locality of the access.

4.1. General considerations on hashing for hierarchical memory

The general setting (see Fig. 2) is a hierarchical memory structure with a hash table H_b that is kept on the SSD and a hash table H_f that is kept in RAM. As said, SSDs prefer sequential writes and sequential reads, but can cope with an acceptable number of random reads. Both hash tables store state information, but the internal hash table is smaller, since it is located in RAM — thus it can serve only as a kind of buffer or cache for the external table.

Collision detection and disambiguation can yield an additional computational burden, especially on the external hash table as they usually lead to repetitive random reads, which are much slower than in RAM. As chaining requires an overhead for storing and following links, we are left with open addressing and adequate probing strategies.

For resolving collisions in RAM, strategies like quadratic probing and double hashing, which lead to a better distribution, usually yield a smaller number of (state comparison) steps for searching (as well as inserting) elements. In an external memory setting this is no longer the case. As linear probing finds elements through sequential scanning, it is more I/O efficient. This fact remains true for external hashing on SSDs as the seek time is substantial.

The efficiency analysis of linear probing goes back to Knuth [43]. For a load factor of α a successful search requires about $1/2\,(1 + 1/(1-\alpha))$ accesses on average, while an unsuccessful search requires about $LP_\alpha = 1/2\left(1 + 1/(1-\alpha)^2\right)$ accesses on average. This means that for a hash table that is filled up to $\alpha = 50\%$ we expect less than three states to look at on the average, which easily fit into one block. Given that random access is slower than sequential, this implies that unless the hash table becomes filled, linear probing with one I/O per lookup per node is an appropriate candidate for SSD-based hashing.

4.2. Mapping: first layered memory hashing strategy

The simplest method to apply SSDs in graph search is to store each node at its external hash address in a file, and – if occupied – to apply conflict resolution on disk. Due to their large seek times, this option is clearly infeasible for HDDs, but it does apply to some extent on SSDs. Nonetheless, besides extensive use of random writes that operate block-wise and are

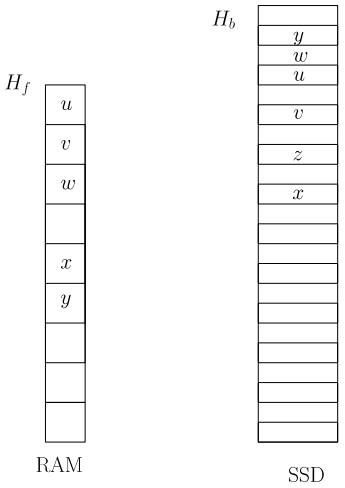

Fig. 2. Internal and external memory, such as RAM and SSD.

expected to be slow, one problem of the approach is the initialization time, incurred by erasing all existing data stored in external memory.

Hence, we apply a refinement to speed-up the search. With one additional large bit-vector kept in RAM, we denote, whether or not a position has been already occupied. This reduces initialization time to clearing all bits in the main memory. Moreover, this saves lookup time in case of hashing a new state with an unused table entry. Viewed differently, one can think of a Bloom filter [13] with conflict resolution on disk. Fig. 3(left) illustrates this approach. The bit-vector *occupied* memorizes whether the address on the SSD is in use or not.

With a full state vector of several bytes to be stored in the external memory, investing one bit per state in RAM is often acceptable. Hence, the limit for this exploration strategy is the number of states that can be stored on the solid state disk, which we assume to be sufficiently large.

For analyzing the approach, let n be the number of nodes and e be the number of edges in the state space graph that are looked at. Moreover, we generally assume an $O(n + e)$ graph search algorithm like NDFS.

- Without an occupied vector e lookup and n insert operations are required. Let B be the size of a block (amount of data retrieved or written with one I/O operation) and $|s|$ be the length of a state. As long as $LP_\alpha \cdot |s| \leq B$, at most two[3] blocks are read for each lookup.[4] For $LP_\alpha \cdot |s| > B$ no additional random read access is necessary. After the lookup, an insert operation results in one random write. This results in an I/O complexity of $O(e + pn)$.
- Using the occupied vector, the number of read operations reduces from e to n, assuming that no collisions take place yielding an I/O complexity of $O(pn)$.

As the main bottleneck of the approach is random writing to the external memory, we can additionally employ a write buffer in the form of an internal hash table in RAM as another refinement. Due to numerous insert operations, the internal hash table will become filled, and then has to be flushed to the external memory, which incurs writes and subsequent reads. One option that we call *merging* is to sort the internal hash table wrt. the external hash function before flushing.[5] Now have a sequential write (due to the linear probing strategy), such that the total worst-case I/O time for flushing is bounded by the number of flushes times the efforts for sequential writes. Fig. 3(right) illustrates the approach. As we are able to exploit sequential data processing, updating the external hash table corresponds to a scan of the data (see Fig. 4). Blocks are read into the RAM and merged with the internal information and then flushed back to SSD.

4.3. Compressing: second layered memory hashing strategy

State compression is a common option in LTL model checking. There are lossless compression strategies like FSM compaction [40], as well as lossy compression strategies like bit-state hashing [38] or hash compaction [60]. For the sake of completeness, in this article we avoid lossy compression methods as they imply partial state space coverage.

Instead we store all state vectors in a file on the external storage device, and substitute the state vector by its relative file pointer position. For an external hash table of size m this requires $\lceil \log m \rceil$ bits per entry, summing up to $m\lceil \log m \rceil$ bits in total. Fig. 5 illustrates the approach with arrows denoting the position on external memory. An additional bit-vector *occupied* is no longer needed.

[3] When linear probing arrives at the end of the table, an additional seek to the start of the file is needed.

[4] In our system: $B = 4096$ bytes, and $|s| \approx 40$ bytes.

[5] If the sequence is partially sorted, this might call for adaptive sorting algorithms.

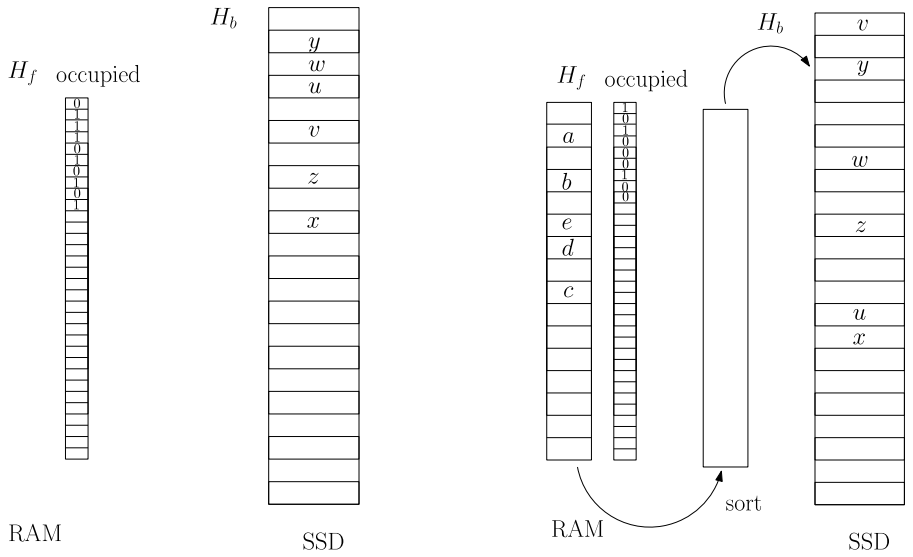

Fig. 3. External hashing without and with merging.

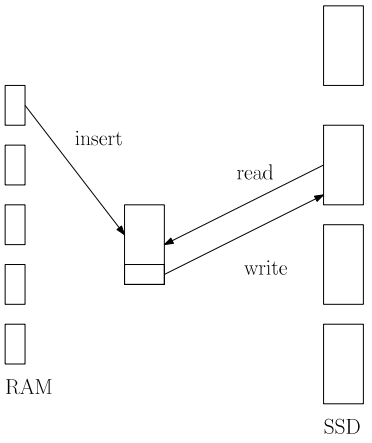

Fig. 4. Updating tables in hashing with linear probing while merging.

This strategy also results in e look-ups and n insert operations. Since the ordering of states on the SSD does not necessarily correlate with the order in main memory, the lookup of states due to linear probing induces multiple random reads. Hence, the amount of individual blocks which have to be read is bounded by $LP_\alpha \cdot e$. In contrast, all insert operations are performed sequentially, utilizing a cache of B bytes in memory. Subsequently this approach performs $O(LP_\alpha \cdot e)$ random reads on the SSD. As long as $LP_\alpha < 2$ this approach performs fewer random read operations than mapping. As sequential writing of n states of s bytes requires $n|s|/B$ I/Os, the total flash-memory I/O complexity is $O(LP_\alpha \cdot e + n|s|/B)$.

Note that *Cuckoo hashing* [53], where 2 hash tables are used and the keys are distributed between them, can reduce the number of look-ups to at most 2.

4.4. Flushing: third layered memory hashing strategy

The above approaches either require significant time to write data according to h_b, or request significant sizes of RAM. There are further trade-offs that we will consider next.

One first solution that we call *padding* is to append the entire internal hash table as it is to the existing data on the external table. Hence, the external hash function can be roughly characterized as $h_b(s) = i \cdot m' + h_f(s)$, where i denotes the current number of flushes and s the state to be hashed.

Writing is sequential and the conflict resolution strategy is inherited from internal memory. Reading a state for answering membership requires up to i many table look-ups. Conflict resolution may lead to an even worse performance. For a moderate number of states that exceed RAM resources only by a very small factor, however, the average performance is expected to be good. Since all states can reside in the main memory, no access to the external memory is needed.

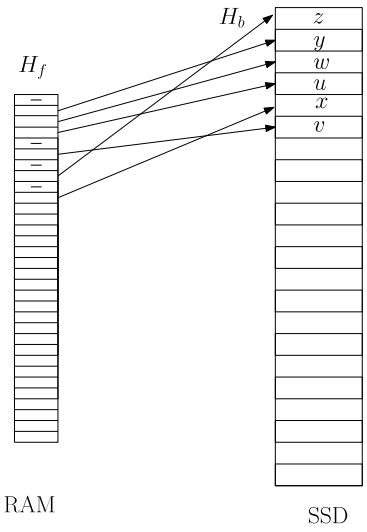

Fig. 5. State compressing.

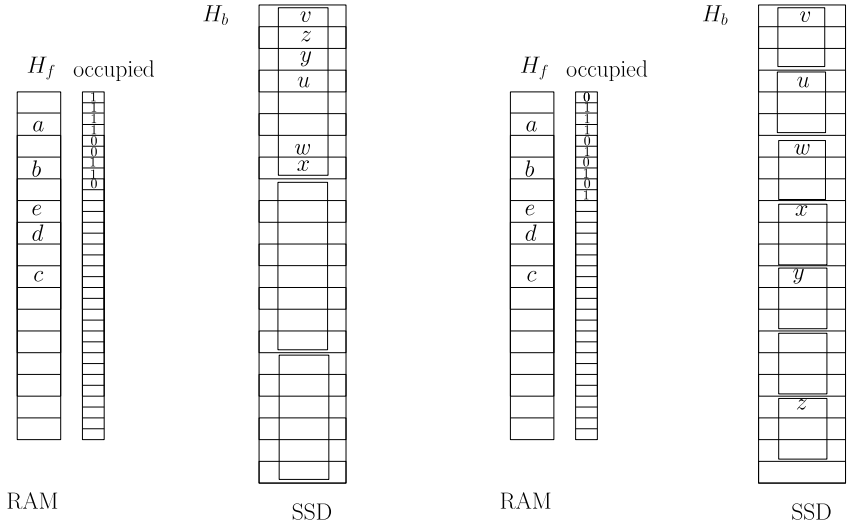

Fig. 6. Padding and slicing.

We can safely assume that the load factor α is small enough, so that the extra amount of work due to linear probing is sufficiently small by using block accesses. Again e look-ups and n insert operations are performed. Let e_i be the number of successors generated in stage i, $i \in \{0, \ldots, r-1\}$. For stage 0 no access to the external table is needed. For stage i, $i > 0$, at most $O(i \cdot e_i)$ blocks have to be read. Together with the sequential write of n elements (in r rounds) this results in a complexity of $O(n|s|/B + rp + \sum_{0 \le i < r} i \cdot e_i)$ I/Os.

An illustration is provided in Fig. 6(left). The entire internal hash table has been flushed once, while the maximum number of flushes is set to 3.

The obvious alternative is to *slice* the external hash table such that $h_b(s)$ becomes $h_f(s) \cdot r + i$. An illustration is provided in Fig. 6(right).

The disadvantage of processing the entire external hash table during flushing is compensated by the fact that the probing sequences in the hash tables can now be searched concurrently. For the lookup we use a Boolean vector of size i that monitors if an individual probing sequence has terminated with an empty bucket. If all probing sequences fail, the query itself has failed.

4.5. Strategy comparison and outcomes

Various implementations of graph traversal on the SSD have been suggested. It is apparent that some of them are less I/O efficient but have lower demands on the internal memory (mapping and flushing strategies), while others allocate more RAM but perform much fewer I/O operations in the ordinary case (compress strategy).

Table 2
Overview of hash functions of different hash strategies applied to on-the-fly LTL model checking.

	Mapping	Compressing	Padding	Slicing
$h_f(s)$	–	$h(s) \bmod m$	$h(s) \bmod m$	$h(s) \bmod m$
$h_b(s)$	$h(s) \bmod m$	$h(s) \bmod m$	$i \cdot m' + h(s) \bmod m'$	$(h(s) \bmod m) \cdot r + i$

Table 3
Space complexity in bits for different levels in the memory hierarchy for different hash strategies for on-the-fly LTL model checking.

	Mapping	Compressing	Padding	Slicing
RAM	$2m$	$m + m\lceil \log m \rceil$	$2m + m' \times \|s\|$	$2m + m' \times \|s\|$
SSD	$m \cdot \|s\|$	$m \times \|s\|$	$m \times \|s\|$	$m \times \|s\|$
HDD	$\max_i \|Open_i\| \times \|s\|$	$\max_i \|Open_i\| \times \|s\|$	$\max_i \|Open_i\| \times \|s\|$	$\max_i \|Open_i\| \times \|s\|$

With the above hashing schemes, we arrive at full flexibility in applying immediate duplicate detection in DFS. Table 2 summarizes the functions applied for the different hashing strategies. As parameters we have $m = 2^b$ and $m' = 2^f$. Moreover, h is the hash function, e.g., as found in the *DIstributed VerIficatioN Environment* (DiVinE) [6], m is the size of the external hash table (in the number of elements),

Table 3 compares the amount of memory required for the different hashing strategies; $|s|$ denotes the state vector size (measured in bits) and $Open_i$ denotes the set of states in the search frontier in iteration i.

Note that there are refinements to NDFS [59], which are faster, but need more bits.

5. Semi-external LTL model checking

Having a collision-free hash function on a set of all states and limited information per state (e.g., one flag for monitoring if a state has been visited), a graph algorithm stores in RAM only a constant number of bits per state. To formalize a class of such graph algorithms Edelkamp et al. [29] define *semi-external* graph algorithms as follows:

> A graph algorithm $\mathcal{A}$ is called *c-bit semi-external* for $c \in \mathbb{R}^+$, if there is a constant $c_0 > 0$ such that for each implicit graph $G = (V, E)$ the internal memory requirements of $\mathcal{A}$ are at most $c_0 \cdot v_{max} + c \cdot |V|$ bits, where v_{max} is a maximal bit-length for representing a vertex from V.

Including v_{max} in the complexity is necessary, since this value varies for different state spaces. Note that c_0 is the same for all state spaces, therefore it is not possible to hide arbitrarily large costs in $c_0 \cdot v_{max}$.

To show that a semi-external graph search is possible, next we present insights into perfect hashing.

5.1. Practical perfect hashing

Perfect hashing [14] is a space-efficient way of associating unique identifiers to states. It yields constant random access time in the worst-case. Perfect hash functions are injective mappings, while a bijective mapping from a set of states is called a *minimal perfect* hash function. To construct a perfect hash function for a model checking instance according to [15], it is necessary to generate the entire state space graph G first. Thus, a hash function construction has the same I/O complexity as the algorithm for I/O efficient reachability analysis (implemented by I/O efficient BFS). Hence, hash function construction pays off only if it can accelerate algorithms with worse I/O complexity than that of I/O efficient reachability analysis — e.g., for external memory LTL model checking we know only algorithms slower than reachability analysis.

In essence, perfect hashing is an injective mapping of some state set S to the index range $\{1, \ldots, m\}$ (or, equivalently, to $\{0, \ldots, m-1\}$) with $|S| \leq m$. For minimum perfect hashing we additionally have $m = |S|$, implying that the mapping is one-to-one. For the construction of a (minimum) perfect hash function the set S has to be known.

Let U with $|U| = u$ be the universe, of which a set of states $S \subseteq U$ with $|S| = n$ is chosen. Hashing relates to storing and finding elements of the set S in T with $|T| = m$ in order to implement an efficient dictionary data structure. Unfortunately, collisions in the form of two different elements mapped to the same address are frequent: the probability of no collision $m!/(m^n(m-n)!)$ is small even for n/m being large. Usually collisions are not avoided but resolved.

Perfect hashing, however, is the construction of a collision-free (injective) mapping of some set $S \subseteq U$ to the index range of T and usually leads to $O(1)$ worst-case access time. The construction of a minimum perfect hash function is a major theoretical result in computer science and goes back to work of Fredman et al. [33]. Assuming U with $|U| = u$ to be a set of numbers they have shown that for the (universal[6]) class of hash functions

$$\mathcal{H}(k, u, s) = \{h(x) = (k \cdot x \bmod u) \bmod s \mid k \in \{0, \ldots, u-1\}\}$$

[6] A class of hash functions $\mathcal{H} = \{h \mid h : U \to [0 \ldots m-1]\}$ is *universal* if for any h in $\mathcal{H}$ we have $|\{h \in \mathcal{H} \mid h(x) = h(y)\}| = |\mathcal{H}|/m$, so that the probably of collision $P(h(x) = h(y)) = 1/m$ for all x, y.

and buckets

$$B_i = \{x \in S \mid h \in \mathcal{H}(k, u, s) : h(x) = i\}$$

there exists a k in U and $s \geq n$ with $\binom{|B_i|}{2} < n^2/s$. This central result implies that the number of collisions at least for a certain $h \in \mathcal{H}$ can be bounded. As a consequence, the authors derive that for buckets of quadratic size, i.e., for $s = O(n^2)$, there exists a k in U such that for all i we have $|B_i| \leq 1$. Hence, at most one element is stored, yielding a perfect hash. Moreover, using this results we can find at least one k in U with $\sum_i |B_i|^2 = O(n)$.[7] Therefore, in a two-level hash approach, where a first hash function h maps the elements to a first array and a second hash function h_i that hashes the elements to arrays of quadratic size (with a sum that is still linear in n), the authors could guarantee an overall perfect hash function using $O(n)$ space while providing $O(1)$ lookup time.

Since the construction refers to static sets, Dietzfelbinger et al. [24] generalized Fredman, Komlòs and Szemerédi's result to a dynamic setting by exploiting properties of a certain class of polynomial hash functions.

For minimizing the space requirements measured in the number of bits needed for storing the perfect hash function, Mehlhorn [49] showed that at least $\Omega(n + \log\log u)$ bits are needed. Schmidt and Siegel [57] closed the gap by showing that a minimum perfect hash function with $O(1)$ worst case access with $O(n + \log\log u)$ bits does exist. No construction algorithm was given. Fortunately, Hagerup and Tholey [35] could construct a minimal perfect hashing (with $O(1)$ access time), using $n \log e + \log\log u + O(n(\log\log n)^2/\log n + \log\log\log u)$ bits[8] that, nonetheless, was hardly practical.

Exploiting random (hyper)graph theory, Majewski et al. [47] established $O(1)$ worst-case perfect hashing with $O(n \log n)$ bits. This work is the basis for minimum perfect hashing with $O(n)$ bits as proposed by Botelho et al. [14,15] that we used for our experiments. The actual compression relies on deeper results in hypergraph theory, whose exposition exceeds the scope of this article. Conceptually, it refers to the construction of k hash functions and tables through randomly constructing k-partite graphs. An element that is perfectly hashed has to be present in one of the k tables, meaning that the randomly drawn graph has to be k-partite and acyclic with high probability. For $k = 2$ two hash functions map elements to two tables, so that a bipartite graph has to be constructed. Then an element is either in one table or the other, so that the lookup operates in constant time. The crucial observation is that with little extra space in each table the randomly drawn bipartite graph is acyclic with high probability.

Space-efficient perfect hashing according to [14,15] has been implemented recently.[9] Roughly speaking, it builds on a partition with buckets of at most $n = 256$ elements each, using a simple first-level hash function that guarantees 128 bucket elements on the average, and no more than 256. For each of the buckets, we now have two individual hash tables on which a bipartite graph is built.[10] The two hash functions of each bucket can be stored compactly in $m = 2cn$ bits, with $c \approx 1.05$. If the number of elements in the bucket addressed by a first-level hash function is 256, then $m \approx 530$ bits have to be stored to evaluate the perfect hash function correctly. In flash-memory model checking we take advantage of the fact that a global state lookup in a hash function stored on disk (the external construction of the perfect hashing refers to [15]) requires 1 seek, then reading a sequence of bits representing a bucket. If the number of bits is smaller than the block size, there is no additional overhead.

The currently best space performance for (minimum) perfect hashing is achieved in the *hash, displace & compress* method by Belazzougui et al. [12], which follows the *hash & displace* method that goes back to Tarjan and Yao [62]. The algorithm yields $O(n)$ construction and $O(1)$ membership queries, with a space performance close to the information-theoretical optimum, e.g., for $m = n$ it yields about 2.07 bits per key and for $m = 1.23n$ it yields about 1.4 bits per key.

Let us briefly study how close to the optimum these space performances are. On page 129, Mehlhorn [50] states that the lower bound for the number of bits for storing a perfect hash function is

$$L = \log \frac{\binom{u}{n}}{\left(\frac{u}{m}\right)^n \binom{m}{n}} = \sum_{0 \leq i < n} \log(1 - 1/u) - \sum_{0 \leq i < n} \log(1 - 1/m). \tag{1}$$

The numerator denotes the number of subsets S in U, while the denominator denotes the number of sets any h is perfect on. Thus, we have that $L/\log e = \sum_{0 \leq i < n} \ln(1 - 1/u) - \sum_{0 \leq i < n} \ln(1 - 1/m)$. By using lower and upper bounds on $\sum_{0 \leq i < n} \ln(1 - 1/k)$ with $k = m$ and $k = u$ we induce $L/\log e > -(u - n)\ln((u - n)/u) + (m - n + 1)\ln((m - n + 1)/m)$ as follows.

We have

$$k \cdot \int_{1-\frac{i+1}{k}}^{1-\frac{i}{k}} \ln x \, dx \leq \ln(1 - 1/k) \leq k \cdot \int_{1-\frac{i}{k}}^{1-\frac{i-1}{k}} \ln x \, dx$$

7 This result was strengthened, so that for at least half of all possible k in U we have $\sum_i |B_i|^2 = O(n)$.

8 Value e stands for Euler's number, so that $\log e = 1.44269504$.

9 See http://cmph.sourceforge.net.

10 The two-table construction shares similarities with cuckoo hashing.

such that

$$k \cdot \sum_{0 \le i < n} \int_{1-\frac{i+1}{k}}^{1-\frac{i}{k}} \ln x \, dx \le \sum_{0 \le i < n} \ln(1 - 1/k) \le k \cdot \sum_{0 \le i < n} \int_{1-\frac{i}{k}}^{1-\frac{i-1}{k}} \ln x \, dx$$

and

$$k \cdot \int_{1-\frac{n}{k}}^{1} \ln x \, dx \le \sum_{0 \le i < n} \ln(1 - 1/k) \le k \cdot \int_{1-\frac{n-1}{k}}^{1} \ln x \, dx.$$

By using

$$k \cdot \int_{1-\frac{t}{k}}^{1} \ln x \, dx = k \cdot [(\ln x - 1) \cdot x]_{1-\frac{t}{k}}^{1} = -(k-t)\ln(1-t/k) - t$$

for $t = n$ and $k = u$ or $t = n-1$ and $k = m$ we get the bound $L/\log e > -(u-n)\ln((u-n)/u) + (m-n+1)\ln((m-n+1)/m)$ stated above.

Given that u/n is large, for $n = m$ we have $\ln(1 - n/u) \approx -n/u$ so that $(u-n)\ln((u-n)/u) + (m-n+1)\ln((m-n+1)/m) > -(u-n)\cdot(-n/u) - \ln n = n - n^2/u - \ln n$, so that for minimum perfect hashing with $m = n$ we derive a lower bound L of about $n \log e \approx 1.44n$ and for $m = 1.23n$ this gives a value of approximately $0.89n$ bits.

Preserving the ordering in the input, i.e., $h(x) < h(y)$ for all keys $x < y$, can be harder. Given that keys can be in any order, order-preserving hashing requires $\Omega(n \log n)$ bits. If the keys are known to be in lexicographic order, Belazzougui et al. [11] have shown (and implemented[11]) an $O(n \log\log\log u)$ space and $O(\log\log u)$ search time, as well as an $O(n \log\log u)$ space and $O(1)$ search time algorithm.

An alternative perfect hashing approach applicable to symbolic state spaces was proposed by Dietzfelbinger and Edelkamp in [23]. The authors present linear time invertible perfect hash functions based on precomputed SAT-count values in a BDD.

5.2. Semi-external LTL model checking with solid state disks

Although no algorithm for I/O-efficient DFS on implicit graphs is known, by using perfect hash functions, Edelkamp et al. [29] show that semi-external DFS is feasible. In their approach

- first, an external memory BFS [52] generates all states on disk (the external step).
- then, a minimal perfect hash function (residing in RAM) is constructed on the basis of all these states.
- finally, the actual verification is performed using this perfect hash function to address a table of Boolean values implementing a set of visited states.

The external step allocates at most $c_0 \cdot v_{max}$ bits for proper c_0, the second step allocates another c bits per state because of the hash function representation and the actual verification needs one additional bit per state to implement the set of visited states. Therefore, such algorithm for DFS is $(c+1)$-bit semi-external.

Since semi-external DFS is only conceptually important, it is relatively easy to implement semi-external versions of NDFS for searching for accepting cycles in a given graph. It essentially performs two interleaved DFSs — one searching for accepting states and one searching for a path from each discovered accepting state to itself. Both DFSs have their own sets of visited states. Thus, such an implementation of NDFS is $(c+2)$-bit semi-external. To save one bit per state, Edelkamp, Sanders and Šimeček propose DDFS, the variant of NDFS that runs both DFSs separately. The algorithm performs the first DFS to find all accepting states. The second DFS explores the state space seeded by these states. Seeds are taken in the depth-first post-order given by the first DFS. Both DFSs share the same space for storing the set of visited states. Therefore, such an implementation of DDFS consumes the same amount of memory as one DFS. Consequently, DDFS can be implemented by a $(c+1)$-bit semi-external algorithm.

The I/O complexity of both, DFS and DDFS, is dominated by an initial state space generation, i.e., $O(l \cdot scan(|V|) + sort(|E|))$.

Beside introducing the notion of c-bit semi-external algorithms, Edelkamp et al. shows that semi-external LTL model checking outperforms ordinary external algorithms especially on large state spaces. Moreover, it has a better I/O complexity than ordinary external algorithms — its asymptotic I/O complexity is the same as that of external reachability analysis.

The key idea to improve the RAM efficiency of semi-external memory algorithms is rather simple. Instead of the hash function being maintained completely in RAM, it is stored (partially or completely) on SSD. The visited bit array that denotes whether or not a state has been seen before remains in RAM and consumes one bit per state.

[11] See http://Sux4j.dsi.unimi.it.

Procedure SSD-LTL-Model-Check
Input: Initial State s, Successor Generating Function *succ*

Vars: *StateSpace* : State vectors on HDD
h : Perfect hash function, $c_{PHF} \times |StateSpace|$ on SSD
visited : internal bit-array [1..|*StateSpace*|]
StateSpace := *External-BFS*(s, *succ*);
h := *Construct-PHF*(*StateSpace*);
Semi-External-LTL-Model-Check(s, *succ*, h, *visited*);

Fig. 7. Flash-efficient semi-external model checking (wrapper).

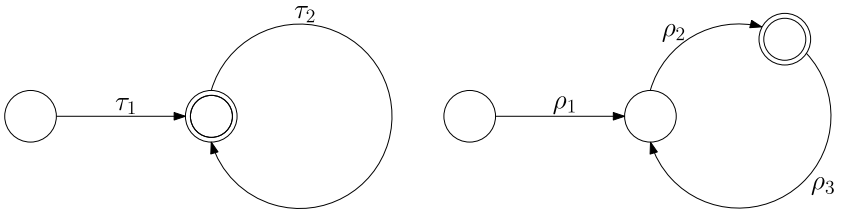

Fig. 8. Two types of lasso-shaped LTL counterexamples (left — seed is accepting, right — seed may not be accepting).

Note that *static* perfect hashing, as approached in this article (in contrast to *dynamic* perfect hashing[12] [5]) is flash-efficient. Most perfect hashing algorithms [33] can be made dynamic [24], but here we have the additional limitation of slow random writes (as discussed in Section 4), so that rehashing has to be sequential. In other words, internal and external hash functions have to be compatible.

Our static setting distinguishes three phases: state space generation, perfect hash function construction and search. The external memory construction process in [15] is streamed and includes sorting the input data according to some first-level hash function. Therefore, it can be executed efficiently on the hard disk. In our experiments with a key set provided as a file, the perfect hash function was constructed also in form of a file, reading the keys from disk and writing the generated perfect hash function to it. (Or from hard disk to solid state disk, or from flash media card to solid state disk.)

Fig. 7 shows a generic implementation of an extended LTL model checking algorithm with integrated flash memory for storing and accessing the perfect hash function. For implementing *Semi-External-LTL-Model-Check*, different options are available. For the example of semi-external DDFS (possibly combined with an iterative deepening startegy), one bit per state in the vector *visited* is sufficient. Hence, in Fig. 7 we have allocated 1 bit for each reachable state to support early duplicate detection in RAM.

Exploiting flash memory, the semi-external minimal counterexample algorithm described above can be made 1-bit semi-external, if the BFS depth is attached to the state in the perfect hash function on the solid state disk. Therefore, the number of bits required at each state on the solid state disk is enlarged by the logarithm of the maximum BFS layer. Assuming that this value is smaller than 256, which was the case in our experiments, one byte per state is sufficient.

6. Finding minimal counterexamples semi-externally

Counterexamples to LTL properties are infinite paths in (finite) state spaces of Büchi automata where some accepting states occur infinitely often. If such a counterexample exists, then there is also at least one so called lasso-shaped counterexample (see Fig. 8) consisting of an infinite loop containing an accepting state and a path to that loop. A state in which the path meets the loop is called a *seed*. There are two possible definitions of the minimality of counterexamples. The first one requires an accepting state being a seed (left), while the second one allows an accepting state to occur at an arbitrary position in the loop (right).

Depth-first-search based algorithms (usually) produce non-minimal counterexamples. Many other algorithms, like OWCTY and MAP [18,16], also do not guarantee minimality of the counterexamples produced. As we are not aware of an internal linear-time algorithm for finding minimal counterexamples, we cannot expect a number of I/Os linear to the efforts of scanning the search space.

The external memory LTL model checking algorithm of Edelkamp and Jabbar [25] produces a minimal-length lasso-shaped counterexample of the first type. Its worst case I/O complexity is defined by searching the product graph with $|Accept| \times |V|$ nodes where *Accept* is a set of accepting states.

For this article, we consider a stronger optimality criterion for which the counterexample $\rho_1\rho_2\rho_3$ is minimal among all lasso-shaped counterexamples (of the second type). It is obvious that the length is at most as large as the above. We produce

[12] Where, each time RAM becomes sparse, the internal function, which stores the states in the RAM, has to be moved and merged with the external hash function, using external storage.

Procedure Minimal-Counterexample-Search
Input: Initial State s, Successor Generating Function *succ*

Var: ϵ_s : max depth of the BFS starting at s
StateSpace : state vectors incl. BFS level (external)
Accept : accepting state vectors incl. BFS level (external)
h : perfect hash function $c_{PHF} \times |StateSpace|$ (external)
visited : internal bit-array $[1..|StateSpace|]$
depth : $|StateSpace| \times \lceil \log(\epsilon_s + 1) \rceil$ (internal or external)
G : file for state vectors on external memory
PQ : internal dynamic array of files for state vectors
$(StateSpace, \epsilon_s) := External\text{-}BFS(s, succ)$;
$h := Construct\text{-}PHF(StateSpace)$;
$visited := (0..0)$; $G := \emptyset$;
$(depth, Accept) := BFS\text{-}distance(s, succ, G)$;
$opt := \infty$;
for each $(r \in Accept \wedge depth(r) < opt)$ **do**
  $visited := (0..0)$; $G := \emptyset$; $PQ := BFS\text{-}PQ(r, succ, G)$;
  $visited := (0..0)$; $G := \emptyset$; $(t, n) := Prio\text{-}min(r, succ, PQ, G)$;
  if $(n < opt)$ **then** $s_1 := t$; $s_2 := r$; $opt := n$;
$visited := (0..0)$; $G := \emptyset$; $\rho_1 := Bounded\text{-}DFS(s, s_1, succ, G, depth(s_1))$;
$visited := (0..0)$; $G := \emptyset$; $dist(s_1, s_2) := BFS(s_1, s_2, succ, G)$;
$visited := (0..0)$; $G := \emptyset$; $\rho_2 := Bounded\text{-}DFS(s_1, s_2, succ, G, dist(s_1, s_2))$;
$visited := (0..0)$; $G := \emptyset$; $\rho_3 := Bounded\text{-}DFS(s_2, s_1, succ, G, opt - depth(s_1) - dist(s_1, s_2))$;

Fig. 9. Constructing a minimal counterexample semi-externally by traversing the state space with breadth-first searches.

optimal counterexamples with a worst-case I/O complexity of roughly $|Accept|$ times the one of semi-external BFS with internal duplicate detection. As we will see, the algorithm's I/O complexity shows a considerable improvement to [25]. More importantly, the worst-case space consumption is linear in the model size and matches the ones of OWCTY and MAP.

The following implementation, calling a semi-external BFS $O(|Accept|)$ times, adapts the algorithm of Gastin and Moro [34]; an internal-memory algorithm that finds optimal counterexamples space-efficiently.[13] Moreover, the algorithm progresses only along forward edges, which appears to be a necessity, as in LTL model checking reversing transitions can be cumbersome.

6.1. The Gastin/Moro algorithm adapted to flash memory

Fig. 9 provides the pseudo-code for searching a minimal counterexample with a combination of solid state and hard disks. The construction of the priority queue[14] is shown in Fig. 10, while the synchronized traversal is shown in Fig. 11.

The main algorithm consists of three phases, corresponding to the three concatenated sub-paths of the counterexample $\rho_1\rho_2\rho_3$, path ρ_1 to the cycle seed (phase 1), path ρ_2 from the seed to the accepting state (phase 3) and path ρ_3 back from the accepting state to the seed (phase 2). Phase 1 of the algorithm executes a plain BFS, that comes for free while constructing the perfect hash function, even though our implementation performs another semi-external BFS for it. Phase 2 and 3 start a BFS from each accepting state, incrementally improving a bound *opt* for the length of the minimal counterexample. Phase 3 invokes a BFS driven by an ordering obtained by adding the BFS distances from phases 1 and 2. States in this phase are ordered with respect to $|\rho_1|+|\rho_3|$ and stored in a 1-level bucket priority queue, originally invented by Dial [22].[15] If duplicate elimination is internal, states can be processed in sequence. Hence, all three phases allow streaming and can be implemented I/O efficiently.

Files are organized in form of queues, but they do not support the *Dequeue* operation, for which deleting and moving the content of the file would be needed. Therefore, instead of calling *Enqueue* and *Dequeue* our algorithms are rewritten based on the operations *Append* and *Next*. As a consequence that files do not run empty, and to keep the data structures on the external device as simple as possible, we had to adapt the implementation. The counters l and l' maintain the actual sizes of the currently active and the next BFS layer (at the end of a layer, l' counting the unique successors of the next layer is set to 0). Value q denotes the current head position in the file. It is incremented after reading an element. When the file is scanned completely, elements are removed and $q = 0$ is set.

13 In [34], the algorithm was not implemented. Hence, our presentation eliminates some minor bugs.
14 The notation aligns with [34], proofs of correctness and optimality are inherited.
15 In [34], a heap was used, which is less efficient.

Procedure BFS-PQ
Input: Accepting state r, file for state vectors G, successor generating function *succ*
Output: Dynamic array for state vector files PQ (external)

Vars: $n := 0$; $l := 1$; $l' := 0$; $q := 0$; $loop := false$;
if $(depth(r) < opt)$ **then**
  $PQ.Append[depth(r)](r)$; $G.Append(r)$;
  $visited[h(r)] := true$;
while $(q \neq |G| \wedge n < opt)$ **do**
  $u := G.Next()$; $q := q + 1$; $l := l - 1$;
  for each $(v \in succ(u))$ **do**
    if $(visited[h(v)] = false)$ **then**
      $visited[h(v)] := true$;
      if $(depth(v) + n + 1 < opt)$ **then** $PQ[depth(v) + n + 1].Append(v)$;
      $G.Append(v)$; $l' := l' + 1$;
      $loop := loop \vee (v = r)$;
      if $(loop \wedge depth(v) + n + 1 < opt)$ **then**
        $opt := depth(v) + n + 1$;
  if $(l = 0)$ **then** $l := l'$; $l' := 0$; $n := n + 1$;
if $(loop)$ **then return** PQ **else return** $\emptyset$;

Fig. 10. Constructing a 1-level bucket file-based priority queue.

Procedure Prio-min
Input: Accepting state r, file for state vectors G, successor generating function *succ*, dynamic array of state vector files PQ
Output: Pair (u, t) of state u and lasso length t

Vars: $l := 0$; $l' := 0$; $q := 0$;
  $n := \min\{i \mid PQ[i] \neq \emptyset\}$;
  $p := \sum_i |PQ[i]|$;
while $((p \neq 0 \vee q \neq |G|) \wedge (n + 1 \neq opt))$ **do**
  for each $(u \in PQ[i] \wedge visited[h(u)] = false)$ **do**
    $G.Append(u, u)$; $visited[h(u)] := true$;
    $p := p - 1$; $l := l + 1$;
  while $(l \neq 0)$ **do**
    $(u, u') := G.Next()$; $q := q + 2$;
    for each $(v' \in succ(u'))$ **do**
      if $(v' = r)$ **then return** $(u, n + 1)$;
      if $(visited[h(v')] = false)$ **then**
        $visited[h(v')] = true$; $G.Append(u, v')$;
        $l' := l' + 1$;
    $l := l - 1$;
  $l := l'$; $l' := 0$; $n := n + 1$;
return $(\perp, \infty)$;

Fig. 11. Synchronized traversal driven by the 1-level bucket priority queue on disk.

With the constant access time, perfect hashing speeds up all graph traversals to mere scanning. It provides duplicate detection and fast access to the BFS depth values (wrt. the initial state) that have been associated with each state. Finally, solution extraction[16] reconstructs the three minimal counterexample sub-paths ρ_1, ρ_2, and ρ_3 between two given states using bounded DFS.[17]

Counterexample reconstruction based on bounded DFS can also be implemented semi-externally.[18] Let *opt* be the length of the optimal counterexample and *dist* the length of the shortest path between two states. It is not difficult to see that for start state s, seed state s_1, and accepting state s_2 we have $|\rho_1| = dist(s, s_1)$, $|\rho_2| = dist(s_1, s_2)$ (to be computed with BFS), and $|\rho_3| = opt - dist(s, s_1) - dist(s_1, s_2)$.

16 This code fragment is slightly different to [34].
17 In difference to [34] we avoid overwriting the *depth* value. The DFS depth is determined by the stack size, such that, once the threshold is known, no additional pruning information is needed.
18 The reconstruction is slightly different to [34] as we use the knowledge on $|\rho_1|$ and *opt* to avoid BFS calls.

The implementation already includes performance improvements mentioned in [34], while constructing the priority queue. For example, accepting states without loops or over-sized loops are neglected.

The maximal size for the priority queue is bounded by the maximum depth ϵ_s of the BFS starting at s, plus the diameter of the search space $diam = \max_{s_1,s_2} dist(s_1, s_2)$. As the latter is not known in advance, we use dynamic vectors for storing the priority queue.

6.2. Analyzing the algorithm

In the presented semi-external minimal counterexample algorithm, beside delayed duplicate detection for constructing the perfect hash function, all duplicates are caught in RAM. For sorting-based initial external breadth-first search, we arrive at an overall I/O complexity of

$$O(sort(|E|) + (l + |Accept|) \cdot scan(|V|)),$$

where l is the locality of the search graph $G = (V, E)$. In contrast, the algorithm in [25] applies at most $O(l \cdot scan(|Accept||V|) + sort(|Accept||E|))$ I/Os.

If the perfect hash function is outsourced to the flash memory, the semi-external LTL model checking algorithm described above improves from being $(1 + c_{PHF})$-bit to 1-bit semi-external.

We implemented two externalizations of the perfect hash function.

- In the first one, for each bucket we only flushed the information on the m bits (which leaves about 184 bits remaining in the RAM).
- In the second externalization, we flushed all information except the file pointer to the bucket (which reduces the number of bits per bucket to 64).

In all our implementations with access to the perfect hash function on flash memory, we store some information of the bucket in RAM, but bypass the lower bound of 1.44 bits per state [24]. It is rather simple to extend our implementations to also externalize the remaining bits to the disk,[19] such that we can arrive at a 1-bit semi-external algorithm. Such a 1-bit semi-external algorithm allows using almost all the available RAM for visited bits. The number of I/Os does not change.

Storing the hash function after generating the state space externally requires $scan(|V|)$ I/Os. During a (double) depth-first search, for each state space edge we pose a query to the hash function, such that $O(|E|)$ I/Os are needed. As the hash function for the on-the-fly variant is computed for each BFS level, the complexity can increase.

For minimum counterexample generation we have the following situation. If allocating $1 + c_{PHF} + \lceil \log(\epsilon_s + 1) \rceil$ bits per node exceeds RAM, flash memory can help. Outsourcing the perfect hash function together with the BFS-level takes $O(scan(|V|))$ I/Os, while the total I/O complexity for the look-ups for duplicate detection is bounded by $|\text{Accept}| \cdot |E|$ I/Os. Outsourcing the array *depth* to the solid state disk by enlarging the disk representation of the perfect hash function does not yield additional I/Os, as the access to one compressed state (with depth value included), is still below the block size. (In this case, the pseudo-code changes from the access $depth(v)$ to $depth[h(v)]$.)

7. Experimental evaluation

All algorithms have been implemented extending the *DIstributed VerIficatioN Environment* (DiVinE) [6]. They include only a part of the library deployed with DiVinE, namely state generation and internal storage. For the implementation of the external memory container and for efficient sorting and scanning we have used STXXL (Standard Template Library for Extra Large Data Sets) [21]. The following models have been taken from the BEEM library [54] (where more detailed information can be found):

- *Rether* is a software-based, real-time Ethernet protocol developed at SUNY.
- *MCS* is Mellor-Crummey and Scott list-based queue lock using a fetch-and-store and compare-and-swap algorithm.
- *Train-gate* is a simple controller of a train gate converted from an UPPAAL demo.
- *Lifts* is a controller of a distributed system for lifting trucks.
- *Szymanski* is a three-bit linear wait algorithm for mutual exclusion.

All models are parameterized, so their state spaces differ for different settings. For optimal counterexamples we took the ones for which the minimal lengths were not known.

For the experiments we used a Desktop PC with AMD Athlon CPU (32 bit) a SATA HDD of 280 GB with 13.8 ms seek time and about 61.5 MB/s for sequential reading and a 32 GB SATA solid state disk produced by HAMA, which has 0.14 ms seek time and scales to about 93 MB/s for sequential access.

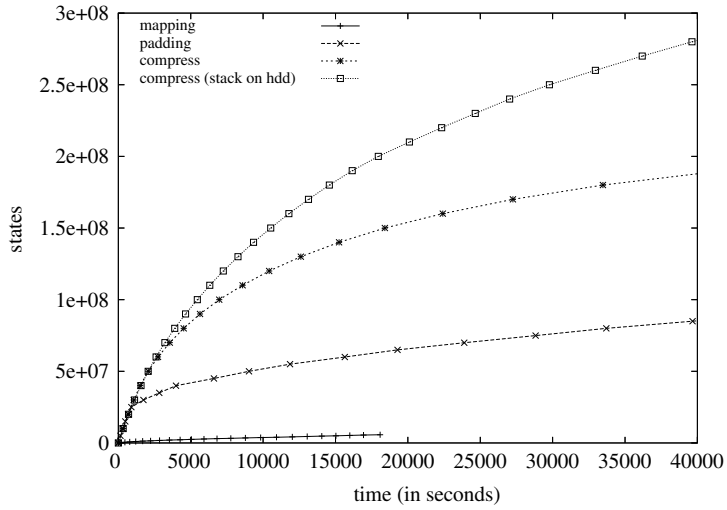

Fig. 12. Comparing the three strategies *Mapping*, *Padding*, and *Compress* on the Rether-4 model.

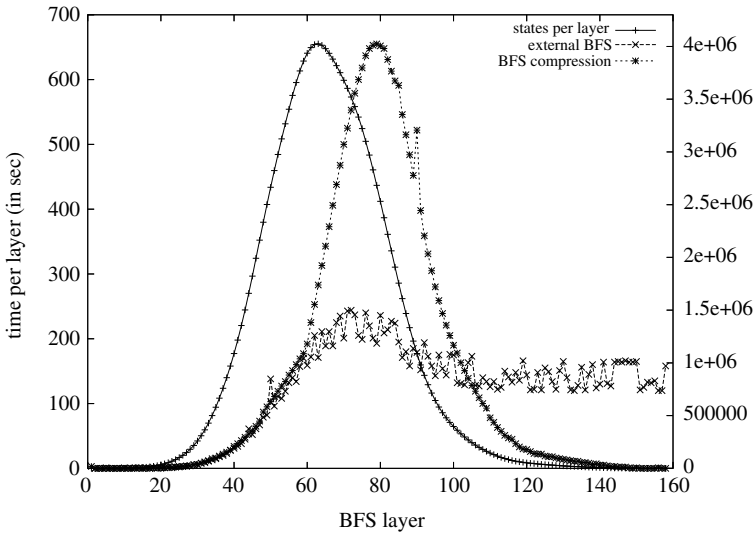

Fig. 13. Comparing cBFS to eBFS on the Szymanski 5 property 4 model (the right axis, together with the *states per layer* plot shows the size of each layer; the remaining curves show the time per layer for the different approaches).

7.1. On-the-fly model checking

To confirm the theoretical results we check the Rether-4 protocol from the BEEM library. The plot shown in Fig. 12 illustrates nested depth-first search runs with different immediate duplicate detection strategies. All experiments, aside from the *mapping* strategy, were stopped after 40,000 s (mapping was stopped after 1800 s due to its obvious lack of performance). The mapping strategy is the worst one because of numerous random writes.

The *compress* strategy performs best, due to its low I/O complexity without inducing any penalties for random writes. The difference between *compress* and *compress (stack on HDD)* is the location of the stack file. In the first case, it was located on the SSD, in the second it was on a separate HDD. We observe that having the stacks stored on a second disk drive gives another speed-up of about 30% for the state space traversal.

The motivation to use SSDs was to exploit fast random access to them. Now, we compare new algorithms designed for SSDs to traditional I/O efficient algorithms, which we run on SSDs too. To get a fair picture about both approaches, we perform a reachability analysis in breadth-first order. As a novel approach we run BFS with immediate duplicate detection and compression strategy (cBFS). As a traditional approach we run a standard external BFS (eBFS) with delayed duplicate detection after each level.

[19] By using a sparse representation of the buckets. A drawback of writing the uncompressed representation of the buckets on disk is that the file size increases by 2 (from 128 on the average to 256).

First, the state space of the Szymanski (5) model with property 4 was generated using both approaches. The plot in Fig. 13 demonstrates the dependency in expanding speed between the cBFS and the BFS layer size, while the expanding time per layer remains almost the same for eBFS. This is due to the fact that in delayed duplicate detection the time of level generation is mostly determined by the size of the visited states set, which is completely passed for each BFS layer. Thus, in a large search depth immediate duplicate detection saves much time compared to delayed duplicate detection.

The next three models were used for testing:

	Number of states	Size in GB
MCS	$120 \cdot 10^6$	4.8
Train-Gate	$50 \cdot 10^6$	3.2
Rether-2	$31 \cdot 10^6$	2.8

It is apparent that the results strongly depend on the structure of a state space. Moreover, as shown in [7], the I/O complexity of eBFS is highly dependent on the number of BFS layers, while the I/O complexity of cBFS is not. This can be demonstrated on the model Rether-2, with 552 BFS layers (see Fig. 14). While eBFS performs poorly on this model, cBFS finishes in several minutes. The new approach can also benefit from a small number of back edges and various heuristics helping to recognize duplicates with no reading from the disk. This is the case for the Train-Gate model, where the amount of random reads was only 30 million, even though the state space has 50 million states, due to the fact that duplicates were typically found in internal buffers (only 8 MB large) before flushing to disk. The model MCS is an example, where eBFS performs better — the state space has a relatively small number of BFS levels (90).

From the I/O complexities of both algorithms and from our measurements it follows that eBFS has to slow down the exploration faster than cBFS with an increasing portion of the state space explored. Thus, cBFS can often outperform it from some BFS level due to its linearity in I/O complexity. The moment, when cBFS outperforms eBFS depends to a great extent on numerous platform and input specific factors: state space structure (number of BFS layers, portion of back edges), bandwidth, access time, file system, implementation (we did not implement the heuristics from [10] or [36]). Even though it is not easy to predict, whether or from which point of exploration cBFS outperforms eBFS, the main impact of behavior of both algorithms is that there can be a threshold, from which cBFS outperforms eBFS on a given input and so algorithms for SSDs like cBFS are practical.

7.2. Minimal counterexamples

Next, we evaluate the efficiency of the minimum counterexample generation, storing the perfect hash function in the RAM. To save time, we generate the state space on SSD, if possible. We observed a speed-up of about 2, compared to the hard disk.

Our first case study is Lifts(7) with property 4. The state space consists of 7496,336 states and 20,080,824 transitions and is generated in 262 layers. Its generation time amounts to about 1122 s on the SSD. The perfect hash function is first split into 58 parts, and then finalized in 66 s. The first BFS that initialized the depth array and flushes the set of accepting states required 187 s. The number of accepting states is 2947,840. Our minimum counterexample algorithm implementation first finds a counterexample with seed depth 81 and lasso length 117 (found within 10 s), which is then improved to seed depth 87 and cycle length 2. Proving this to be optimal yields a total run-time of 4035 s, with the CPU operating at 86%. According to [29], the non-optimal semi-external double DFS approach takes about 1920 s to generate one counterexample. A factor of 2.1 as a trade-off for optimality is acceptable, as optimality is an assertion about all paths.

For the Szymanski (4) model with property 2 the state space consists of 2256,741 states and 12,610,593 transitions and is generated in 110 layers. Its generation took 511 s on the SSD. The hash function is split into 17 parts. The first BFS took 96 s and generated 1128,317 accepting states. The counterexample lengths found are 31 and 19. The last one is optimal. The total run-time is 2084 s. According to [29], semi-external double DFS takes about 600 s to generate a counterexample and is thus faster by a factor of about 3.4.

Last, but not least, we look at the externalization of the hash tables to the SSD. As we have not yet externalized the *depth* array, we applied LTL model checking with the double DFS implementation of [29]. Table 4 shows our results. First, the space consumption of the data structures for the models is reported, then we compare the time–space trade-off in three different externalizations. The first one stores the perfect hash function in the RAM, and thus matches the implementation of [29]. The other approaches externalize the hash function via direct I/O on either hard or solid state disk. Note that all experiments have a static storage offset of 238.89 MB due to the inclusion of the DiVinE model checker and STXXL.

The value *States* indicates the complexity of each model as stored on the hard disk. We have not used the number of states here to highlight the compression ratio between the size of the state space and the size of the perfect hash function *h*.

The columns *h* and *Visits* show that the size of *h* is proportional to the size of the *visited* array. Since *visited* is |*StateSpace*| bits long and *h* contains a representation of each state, this is what one might have expected. Note that the Szymanski protocol needs 4.95 bits per state, while the Lift protocol takes 6.34 bits per state on the average, as the implementation of Botelho and Ziviani [15] is not capable of creating a perfect hash function for this protocol using 4.95 bits per state.

For memory comparison between [29] and our extension to it, we report the RAM usage for *h*. The experiments show that it drops by a factor of 14 for Szymanski and even by a factor of 20 for the Lifts protocol. As mentioned above, it is possible to

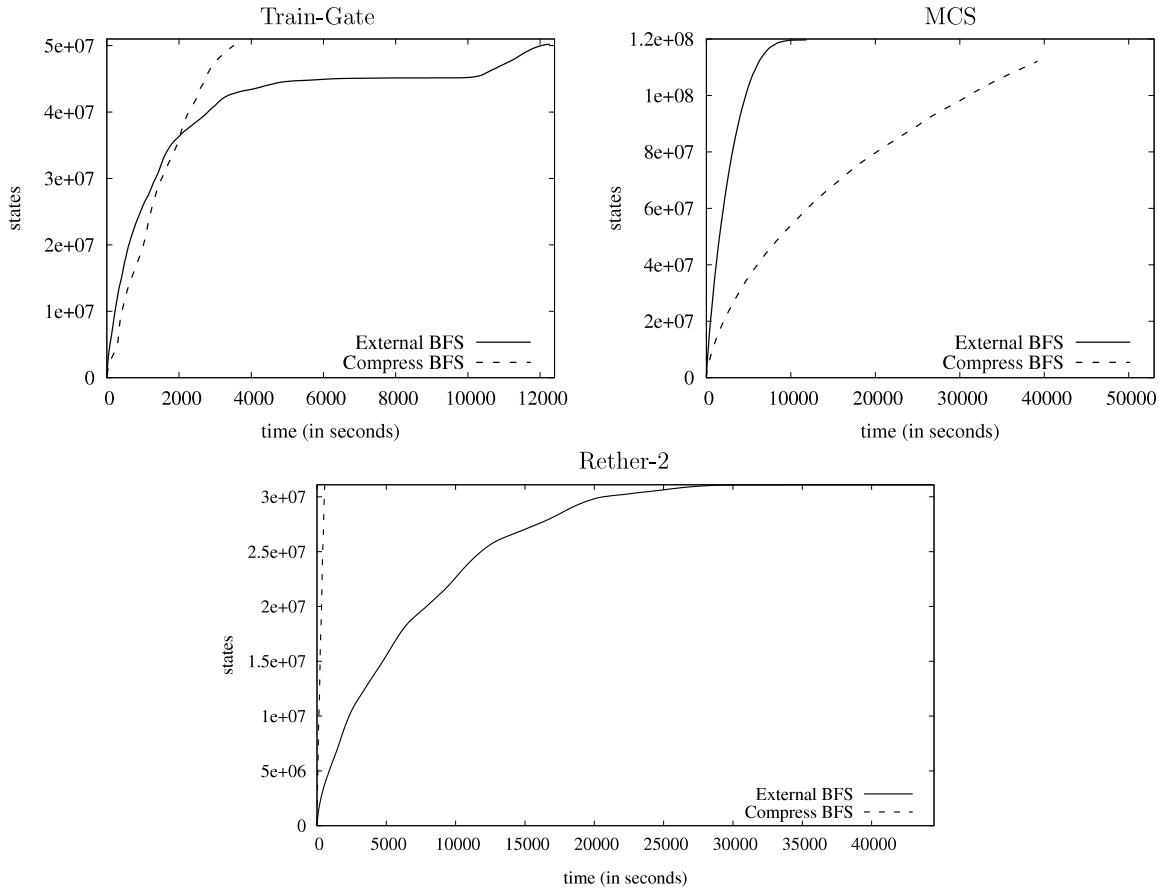

Fig. 14. Comparison of eBFS and cBFS.

Table 4
Flash performance on double depth-first search (on models with invalid temporal properties, times are given in *mm:ss*).

	Space consumption			*h* in RAM		*h* on external device		
Model	*States*	*h*	*Visits*	RAM	Time	RAM	SSD	HDD
Szy.(2),P3	15 MB	3848 kB	778 kB	2877 kB	0:06	2 kB	0:50	0:37
Szy(3),P3	65 MB	133 MB	275 kB	990 kB	2:58	6892 kB	39:02	26:11
Lift(7),P4	351 MB	567 MB	915 kB	455 MB	4:27	220 kB	68:56	48:17
Lift(8),P2	1559 MB	2521 MB	4071 kB	2024 MB	19:44	990 kB	377:22	o.o.t

externalize *h* completely, using more space on the external device, without an increase in I/O: the RAM usage is due to the fact that file pointers are stored to every bucket in the *h* file[20] and could be omitted by imposing a constant bucket length in the file.

We see that outsourcing *h* on the external medium needs more time for the search and that for small experiments, where *h* fits into the hard disk cache, the externalization on hard disk is faster. Lifts(8), P2 is one experiment, where *h* is larger than 16 MB and does not fit into the hard disk cache. The experiment was stopped after six hours. During this time, the CPU usage never exceeded 5%, while the average CPU usage was 48% for the experiment with a solid state disk. This is exactly the same observation that was made with the random data experiments described above.

The time deficiency corresponds to a 19.12-fold slowdown with respect to [29], using 1/20 of the main memory. For very large model sizes, the algorithm described in [29] is infeasible. Moreover, using swap space on a solid state disk is prohibited by the operating system, so that a hard disk is mandatory as a swapping partition. Time-efficiency is the main argument why to use solid state disk instead of a hard disk for storing the perfect hash function in semi-external LTL model checking.

20 E.g., *h*(Lifts(8)),P2 is stored in 260,579 buckets (127.99 entries per bucket in average) and a file pointer is 4 bytes long which results in 1042,316 B = 0.99 MB.

8. Conclusions

Recent designs of external memory algorithms [56] exploit hard disks to work with large data sets that do not fit into RAM, while flash memory devices in form of solid state disks ask for new time–space trade-offs [3]. In such solid state disks random reads operate at a speed that (in orders of magnitudes) lies between RAM and hard disks; while random writes are much slower and should be avoided.

Mass production for solid state disks has recently started; in the near future prices are expected to drop and storage capacities are expected to rise. To exploit the advantages of flash memory compared to disk access, semi-external LTL model checking algorithms have been adapted in this article. As random reads are fast, flash media are good for outsourcing static dictionaries, but generally not for outsourcing dynamic dictionaries that change with random access over time. By exploiting fast random seeks, we obtain interesting trade-offs between loss of speed and savings in RAM. Due to easiness of parallel disk connection, large capacities of SSDs are possible.[21]

We have contributed new approaches to hashing applied to SSDs. The most important observation is that with the advent of SSD technology, immediate duplicate detection becomes applicable, offering more flexibility for the choice of the exploration strategy. Monitoring CPU performance, we observed that hashing strategies preserve ratios of 50% or more, suggesting that I/O waits are present, but not thrashing. With SSDs random access time decreasing, SSDs will likely become fast enough to rise the CPU usage to 100% making the SSD fully transparent to the user.

The standard external memory model of Aggarwal and Vitter [1] hardly covers the discrepancy between (random) reads and writes. We verified the suitability of the flash memory model for predicting the running-time of algorithms using SSD as an external memory, i.e., we could predict on how well the behavior of the algorithms on SSDs corresponds to their theoretical analyses on the flash model. Compression, the best performing strategy, requires substantial main memory, which according to current ratios of space between RAM and SSDs is still no bottleneck. Although we have tested DFS and BFS, our SSD hashing strategies can also be applied to directed model checking approaches [30] to increase the number of states that can be visited.

Directly compared to former I/O-efficient algorithms optimized for magnetic disks there can be a threshold in state space exploration, from which these new approaches pay off on flash disks due to their linearity in size of state space — at least for the compress approach. Algorithms optimized for magnetic disks are not linear, but they have good constant factors which allow them to outperform new approaches on many inputs. The threshold is not dependent only on the structure of a state space, but also on parameters of SSD. With higher bandwidth of SSDs, former I/O-efficient algorithms accelerate, while new approaches are not much influenced since they spend most of the time on random access to data. On the contrary, new approaches take advantage of lower access times, while former algorithms do not as they avoid random I/O operations completely.

For existing semi-external algorithms, the number of I/Os for accessing the hard disk does not change. Therefore, the design of semi-external algorithms should be the first step for flash-efficient model checking. By externalizing the perfect hash function to the flash memory, we discussed the transformation of $(1 + c_{PHF})$-bit to 1-bit semi-external memory algorithms. For minimal counterexample searching in LTL model checking, we adapted an internal-memory algorithm to be $O(1 + c_{PHF} + \lceil \log(\epsilon_s + 1) \rceil)$-bit semi-external with $O((|Accept| + l) \cdot scan(|V|) + sort(|E|))$ I/Os. We discussed a flash-efficient solution, which reduces the RAM requirement from $(1 + c_{PHF} + \lceil \log \epsilon_s \rceil)$ bits per state to $(1 + c_{PHF})$ bits by moving the depth-array to the flash memory and to 1 bit per state by additionally moving the perfect hash function to the flash memory.

The fact that for a minimal counterexample search we perform many independent calls to BFS (one for each accepting state) suggests an effective parallelization.

Acknowledgements

We would like to thank Martin Dietzfelbinger for his help to derive the lower bound on perfect hashing, Peter Kissmann for his rigorous proof reading, and the anonymous reviewers for the helpful hints, continued criticism and valuable pointers to the existing literature.

References

[1] A. Aggarwal, J.S. Vitter, The input/output complexity of sorting and related problems, Communications of the ACM 31 (9) (1988) 1116–1127.
[2] D. Ajwani, A. Beckmann, R. Jacob, U. Meyer, G. Moruz, On computational models for flash memory devices, in: J. Vahrenhold (Ed.), SEA'08: Proc. of the 8th International Symposium on Experimental Algorithms, in: LNCS, vol. 5526, Springer, 2009, pp. 16–27.
[3] D. Ajwani, I. Malinger, U. Meyer, S. Toledo, Characterizing the performance of flash memory storage devices and its impact on algorithm design, in: C. McGeoch (Ed.), WEA'08: Proc. of the 7th Intern. Workshop on Experimental Algorithms, in: LNCS, vol. 5038, Springer, Provincetown, USA, 2008, pp. 208–219.
[4] J. Barnat, Distributed memory LTL model checking. Ph.D. Thesis, Faculty of Informatics, Masaryk University Brno, 2004.

21 E.g., RamSan-620–5 TB SSD array by Texas Memory Systems.

[5] J. Barnat, L. Brim, S. Edelkamp, D. Sulewski, P. Šimeček, Can flash memory help in model checking? in: D.D. Cofer, A. Fantechi (Eds.), FMICS'08: Proc. of the 13th International Workshop on Formal Methods for Industrial Critical Systems, in: LNCS, vol. 5596, Springer, 2009, pp. 150–165. Revised Selected Papers.
[6] J. Barnat, L. Brim, I. Černá, P. Moravec, P. Ročkai, P. Šimeček, DiVinE—a tool for distributed verification, in: T. Ball, R.B. Jones (Eds.), CAV'06: Proc. of the 18th International Conference on Computer Aided Verification, in: LNCS, vol. 4144, Springer, 2006, pp. 278–281.
[7] J. Barnat, L. Brim, P. Šimeček, I/O efficient accepting cycle detection, in: W. Damm, H. Hermanns (Eds.), CAV'07: Proc. of the 19th International Conference on Computer Aided Verification, in: LNCS, vol. 4590, Springer, 2007, pp. 281–293.
[8] J. Barnat, L. Brim, P. Šimeček, Cluster-based I/O efficient LTL model checking, Automated Software Engineering, International Conference on (2009) 635–639.
[9] J. Barnat, L. Brim, P. Šimeček, Parallel I/O-efficient state space generation, in: MASSIVE'09: Proc. of Massive Data Algorithmic, Aarhus University, 2009, pp. 82–92.
[10] J. Barnat, L. Brim, P. Šimeček, M. Weber, Revisiting resistance speeds up I/O-efficient LTL model checking, in: C.R. Ramakrishnan, J. Rehof (Eds.), TACAS'08: Proc. of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, in: LNCS, vol. 4963, Springer, 2008, pp. 48–62.
[11] D. Belazzougui, P. Boldi, R. Pagh, S. Vigna, Monotone minimal perfect hashing: searching a sorted table with O(1) accesses, in: C. Mathieu (Ed.), SODA'09: Proc. of the 19th Annual ACM – SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2009, pp. 785–794.
[12] D. Belazzougui, F. Botelho, M. Dietzfelbinger, Hash, displace, and compress, in: A. Fiat, P. Sanders (Eds.), ESA'09: European Symposium on Algorithms, in: LNCS, vol. 5757, Springer, 2009, pp. 682–693.
[13] B.H. Bloom, Space/time trade-offs in hash coding with allowable errors, Communications of the ACM 13 (7) (1970) 422–426.
[14] F.C. Botelho, R. Pagh, N. Ziviani, Simple and space-efficient minimal perfect hash functions, in: F.K.H.A. Dehne, J.-R. Sack, N. Zeh (Eds.), WADS'07: Proc. of the 10th International Workshop on Algorithms and Data Structures, in: LNCS, vol. 4619, Springer, 2007, pp. 139–150.
[15] F.C. Botelho, N. Ziviani, External perfect hashing for very large key sets, in: M.J. Silva, A.H.F. Laender, R.A. Baeza-Yates, D.L. McGuinness, B. Olstad, O.H. Olsen, A.O. Falcão (Eds.), CIKM'07: Proc. of the 16th ACM conference on Conference on Information and Knowledge Management, ACM Press, New York, NY, USA, 2007, pp. 653–662.
[16] L. Brim, I. Černá, P. Moravec, J. Šimša, Accepting predecessors are better than back edges in distributed LTL model-checking, in: A.J. Hu, A.K. Martin (Eds.), FMCAD'04: Proc. of the 5th International Conference on Formal Methods in Computer-Aided Design, in: LNCS, vol. 3312, Springer, 2004, pp. 352–366.
[17] A. Butterfield, L. Freitas, J. Woodcock, Mechanising a formal model of flash memory, Science of Computer Programming 74 (4) (2009) 219–237.
[18] I. Černá, R. Pelánek, Distributed explicit fair cycle detection, in: T. Ball, S.K. Rajamani (Eds.), SPIN'03: Proc. of the 10th International SPIN Workshop on Model Checking Software, in: LNCS, vol. 2648, Springer, 2003, pp. 49–73.
[19] E.M. Clarke, O. Grumberg, D.A. Peled, Model Checking, The MIT Press, 1999.
[20] C. Courcoubetis, M. Vardi, P. Wolper, M. Yannakakis, Memory-efficient algorithms for the verification of temporal properties, Formal Methods in System Design 1 (2–3) (1992) 275–288.
[21] R. Dementiev, L. Kettner, P. Sanders, STXXL: standard template library for XXL data sets, Software: Practice and Experience 38 (6) (2008) 589–637.
[22] R.B. Dial, Algorithm 360: shortest-path forest with topological ordering, Communications of the ACM 12 (11) (1969) 632–633.
[23] M. Dietzfelbinger, S. Edelkamp, Perfect hashing for state spaces in BDD representation, in: B. Mertsching, M. Hund, M.Z. Aziz (Eds.), KI'09: Advances in Artificial Intelligence, 32nd Annual German Conference on AI, Paderborn, Germany, September 15–18, 2009. Proceedings, in: LNCS, vol. 5803, Springer, 2009, pp. 33–40.
[24] M. Dietzfelbinger, A.R. Karlin, K. Mehlhorn, F.M. auf der Heide, H. Rohnert, R.E. Tarjan, Dynamic perfect hashing: upper and lower bounds, in: FOCS (29th Annual Symposium on Foundations of Computer Science, 24–26 October 1988, White Plains, New York, USA), IEEE, 1988, pp. 524–531.
[25] S. Edelkamp, S. Jabbar, Large-scale directed model checking LTL, in: A. Valmari (Ed.), SPIN'06: Proc. of the 13th International SPIN Workshop on Model Checking Software, in: LNCS, vol. 3925, Springer, 2006, pp. 1–18.
[26] S. Edelkamp, S. Jabbar, in: J. R.R. Dopico, J.D.D.L. Calle, A.P. Sierra (Eds.), Disk-based Search, in: Encyclopedia of Artificial Intelligence, Idea Group Reference, 2008, pp. 501–506.
[27] S. Edelkamp, S. Jabbar, D. Midzic, D. Rikowski, D. Sulewski, External memory search for verification of multi-threaded C++ programs, in: KI 22, 2, 2008, pp. 44–50.
[28] S. Edelkamp, S. Jabbar, S. Schrödl, External A*, in: S. Biundo, T.W. Frühwirth, G. Palm (Eds.), KI'04: Advances in Artificial Intelligence, 27nd Annual German Conference on AI, in: LNCS, vol. 3238, Springer, 2004, pp. 226–240.
[29] S. Edelkamp, P. Sanders, P. Šimeček, Semi-external LTL model checking, in: A. Gupta, S. Malik (Eds.), CAV'08: Proc. of the 20th International Conference on Computer Aided Verification, in: LNCS, vol. 5123, Springer, 2008, pp. 530–542.
[30] S. Edelkamp, V. Schuppan, D. Bosnacki, A. Wijs, A. Fehnker, H. Aljazzar, Survey on directed model checking, in: D. Peled, M. Wooldridge (Eds.), MoChArt'08: Proc. of 5th International Workshop on Model Checking and Artificial Intelligence, in: LNCS, vol. 5348, Springer, 2009, pp. 65–89.
[31] S. Edelkamp, D. Sulewski, Flash-efficient LTL model checking with minimal counterexamples, in: A. Cerone, S. Gruner (Eds.), SEFM'08: Proc. of the 2008 Sixth IEEE International Conference on Software Engineering and Formal Methods, IEEE Computer Society, Washington, DC, USA, 2008, pp. 73–82.
[32] S. Evangelista, Dynamic delayed duplicate detection for external memory model checking, in: K. Havelund, R. Majumdar, J. Palsberg (Eds.), SPIN'08: Proc. of the 15th International SPIN Workshop on Model Checking Software, in: LNCS, vol. 5156, Springer, 2008, pp. 77–94.
[33] M.L. Fredman, J. Komlòs, E. Szemerédi, Storing a sparse table with O(1) worst case access time, in: SFCS'82: Proc. of the 23rd Annual Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, USA, 1982, pp. 165–169.
[34] P. Gastin, P. Moro, Minimal counterexample generation for SPIN, in: A. Cerone, S. Gruner (Eds.), SPIN'07: Proc. of the 14th International SPIN Workshop on Model Checking Software, in: LNCS, vol. 4595, Springer, 2007, pp. 24–38.
[35] T. Hagerup, T. Tholey, Efficient minimal perfect hashing in nearly minimal space, in: A. Ferreira, H. Reichel (Eds.), STACS'01: Proc. of the 18th Annual Symposium on Theoretical Aspects of Computer Science, in: LNCS, vol. 2010, Springer, 2001, pp. 317–326.
[36] M. Hammer, M. Weber, "To Store or Not To Store" reloaded: reclaiming memory on demand, in: L. Brim, B.R. Haverkort, M. Leucker, J. van de Pol (Eds.), FMICS'06: Proc. of the 5th International Workshop on Formal Methods for Industrial Critical Systems, in: LNCS, vol. 4346, Springer, 2007, pp. 51–66.
[37] N. Hart, J. Nilsson, B. Raphael, A formal basis for the heuristic determination of minimum cost paths, IEEE Transactions on System Science and Cybernetics 4 (2) (1968) 100–107.
[38] G.J. Holzmann, An analysis of bitstate hashing, Formal Methods in System Design 13 (3) (1998) 289–307.
[39] G.J. Holzmann, D. Peled, M. Yannakakis, On Nested Depth First Search, in: J.-C. Grégoire, G.J. Holzman, D.A. Peled (Eds.), The SPIN Verification System, American Mathematical Society, 1996, pp. 23–32.
[40] G.J. Holzmann, A. Puri, A minimized automaton representation of reachable states, International Journal on Software Tools for Technology Transfer (STTT) 2 (3) (1999) 270–278.
[41] Intel, October 2009. Intel X25-M and X18-M Mainstream SATA Solid-State Drives. http://www.intel.com/design/flash/nand/mainstream/.
[42] Kirsch, N., July 2009. Intel X25-M 160GB 34nm MLC G2 SSD Benchmark Review. URL: http://www.legitreviews.com/article/1022/13/.
[43] D.E. Knuth, The Art of Computer Programming, Vol. 3: Sorting and Searching, Addison Wesley, 1973.
[44] R.E. Korf, Best-first frontier search with delayed duplicate detection, in: D.L. McGuinness, G. Ferguson (Eds.), AAAI'04: Proc. of the 19th National Conference on Artificial Intelligence, AAAI Press, The MIT Press, 2004, pp. 650–657.
[45] R.E. Korf, P. Schultze, Large-scale parallel breadth-first search, in: M.M. Veloso, S. Kambhampati (Eds.), AAAI'05: Proc. of the 20th National Conference on Artificial Intelligence, AAAI Press, The MIT Press, 2005, pp. 1380–1385.

[46] P. Lamborn, E.A. Hansen, Layered duplicate detection in external-memory model checking, in: K. Havelund, R. Majumdar, J. Palsberg (Eds.), SPIN'08: Proc. of the 15th International SPIN Workshop on Model Checking Software, in: LNCS, vol. 5156, Springer, 2008, pp. 160–175.
[47] B.S. Majewski, N.C. Wormald, G. Havas, Z.J. Czech, A family of perfect hashing methods, The Computer Journal 39 (6) (1996) 547–554.
[48] K.L. McMillan, Symbolic Model Checking, Kluwer Academic Press, 1993.
[49] K. Mehlhorn, Data Structures and Algorithms 1–3, in: Monographs in Theoretical Computer Science. An EATCS Series, vol. 2, Springer, 1984.
[50] K. Mehlhorn, Sorting and Searching, Springer, 1984.
[51] S.L. Min, E.H. Nam, Y.H. Lee, Evolution of NAND flash memory interface, in: L. Choi, Y. Paek, S. Cho (Eds.), ACSAC'07: Proc. of the 12th Asia–Pacific Conference on Advances in Computer Systems Architecture, in: LNCS, Springer, 2007, pp. 75–79.
[52] K. Munagala, A. Ranade, I/O-complexity of Graph Algorithms, in: SODA'99: Proc. of the 10th annual ACM–SIAM symposium on Discrete algorithms, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999, pp. 687–694.
[53] R. Pagh, F.F. Rodler, Cuckoo hashing, in: F.M. Auf der Heide (Ed.), ESA'01: Proc. of the 9th Annual European Symposium on Algorithms, in: LNCS, vol. 2161, Springer, 2001, pp. 121–133.
[54] R. Pelánek, BEEM: benchmarks for explicit model checkers, in: D. Bosnacki, S. Edelkamp (Eds.), SPIN'07: Proc. of the 14th International SPIN Workshop on Model Checking Software, in: LNCS, vol. 4595, Springer, 2007, pp. 263–267.
[55] A. Pnueli, The Temporal Logic of Programs, in: FOCS'77: Proc. of the 18th Annual Symposium on Foundations of Computer Science, IEEE Computer Society, Providence, 1977, pp. 46–57.
[56] P. Sanders, U. Meyer, J.F. Sibeyn (Eds.), Memory Hierarchies, Springer, 2003.
[57] J.P. Schmidt, A. Siegel, The spatial complexity of oblivious k-probe hash functions, SIAM Journal on Computing 19 (5) (1990) 775–786.
[58] V. Schuppan, A. Biere, Efficient reduction of finite state model checking to reachability analysis, International Journal on Software Tools for Technology Transfer (STTT) 5 (2–3) (2004) 185–204.
[59] S. Schwoon, J. Esparza, A note on on-the-fly verification algorithms, in: N. Halbwachs, L.D. Zuck (Eds.), TACAS'05: Proc. of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, in: LNCS, vol. 3440, Springer, 2005, pp. 174–190.
[60] U. Stern, D.L. Dill, Combining state space caching and hash compaction, in: B. Straube, J. Schoenherr (Eds.), Methoden des Entwurfs und der Verifikation digitaler Systeme, 4. GI/ITG/GME Workshop, Shaker Verlag, Aachen, 1996, pp. 81–90.
[61] U. Stern, D.L. Dill, Using magnetic disks instead of main memory in the murphi verifier, in: A. J.Hu, M.Y. Vardi (Eds.), CAV'98: Proc. of the 10th International Conference on Computer Aided Verification, in: LNCS, Springer, 1998, pp. 172–183.
[62] R.E. Tarjan, A.C.-C. Yao, Storing a sparse table, Communications of the ACM 22 (11) (1979) 606–611.
[63] J. Vitter, External memory algorithms and data structures: dealing with massive data, ACM Computing Surveys 33 (2) (2001) 209–271.