Can Flash Memory Help in Model Checking?[⋆]

Jiří Barnat[1], Luboš Brim[1], Stefan Edelkamp[2], Damian Sulewski[2], and Pavel Šimeček[1]

[1] Masaryk University Brno, Czech Republic
[2] Technische Universität Dortmund, Germany

Abstract. As flash media become common and their capacities and speed grow, they are becoming a practical alternative for standard mechanical drives. So far, external memory model checking algorithms have been optimized for mechanical hard disks corresponding to the model of Aggarwal and Vitter [1]. Since flash memories are essentially different, the model of Aggarwal and Vitter no longer describes their typical behavior. On such a different device, algorithms can have different complexity, which may lead to the design of completely new flash-memory-efficient algorithms. We provide a model for computation of I/O complexity on the model of Aggarwal and Vitter modified for flash memories. We discuss verification algorithms optimized for this model and compare the performance of these algorithms with approaches known from I/O efficient model checking on mechanical hard disks. We also give an answer, when the usage of flash devices pays off and whether their further evolution in speed and capacity could broaden a range, where new algorithms outperform the old ones.

1 Introduction

There are numerous computational tasks that require to generate and process that huge amount of data that cannot be simply kept in internal memory. Unfortunately, it is not acceptable in terms of performance to rely on the standard memory virtualization techniques provided by the operating system, and specialized algorithms must be devised to efficiently manipulate data stored externally. These are the so called I/O efficient or external-memory algorithms [2].

I/O efficient algorithms reflect 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, likewise all the PC components, also the external memory devices are being continuously developed and their properties are improving in time. Recently, flash memory based external memory devices became widely used as the so called *solid state disks* (SSDs). Unlike its magnetic counterpart, SSD does not rely on physical movements of the head(s) to access the data. Therefore, the access time is much smaller for a solid state disk

[⋆] 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 74/4-1.

compared to the magnetic one. For example, the speed of random reads for a solid state disk build with NAND flash memory lies roughly at the geometric mean of the speeds of random access memory (RAM) and magnetic hard drive (HDD) [3]. The only factor limiting solid state disks from being massively spread is the cost of the device if expressed per stored bit. The cost per stored bit is still significantly higher for SSDs than for magnetic disks. However, the cost per bit is definitely subject to change in the future.

I/O efficient algorithms have been studied also in the context of formal verification, model checking [4] in particular, as one of the techniques to fight the well known state explosion problem. In this paper we focus on enumerative on-the-fly LTL model checking, which is the standard option for analyzing software systems. Our goal 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 would take advantage of the fast random reads of a solid state disk, or if it is satisfactory to apply the existing I/O efficient LTL model checking algorithms even for SSDs whose characteristics differ significantly from the characteristics of the traditional magnetic disks.

To answer the question we design several techniques to implement an *SSD efficient* graph traversal procedure, namely we discuss several variants of hashing mechanism that is used by the Nested DFS algorithm to efficiently identify already generated states during the graph traversal. We also report on a preliminary experimental comparison of newly suggested SSD efficient and the standard I/O efficient techniques, and discuss the impact of possible technology improvements that may come in the future.

The paper is organized as follows. In Section 2 we briefly recall the standard I/O efficient techniques used for enumerative external memory LTL model-checking. In Section 3 we state the differences between the standard magnetic and new solid state disks. In Section 4 we describe several SSD efficient hashing techniques, and we show in Section 5 how these can be used to design SSD efficient Nested DFS algorithm. Section 6 report on our experimental evaluation of both the SSD and I/O efficient techniques. Finally, in Section 7 we conclude the paper and plot what impact may have possible future technological improvements.

2 I/O Efficient Model Checking with Mechanical Disks

LTL model checking problem can be reduced to the problem of *accepting cycle detection* in the graph [4]. In the context of enumerative LTL model checking, the graph to be searched for the presence of an accepting cycle is generated on-the-fly meaning that if a graph traversal algorithm needs to proceed to an immediate successors t of a state s, it computes state t from the state vector of s. To prevent re-visiting of already explored states, all states that have been processed are stored in memory, hence, if 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 the *duplicate detection*.

Due to the huge number of states, their large size, and the 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 this circumstances a check whether a state has been visited may involve I/O operation as not only the states stored in memory, but also the states stored on external memory device must be considered. This however renders a standard graph traversal algorithm inefficient as the I/O operation is in orders of magnitude slower than a single or several reads from the internal memory.

2.1 Graph Traversal

The core technique that gave birth to I/O efficient algorithms is the so called *delayed duplicate detection* [5,6,7,8] whose idea is to postpone the individual checks against the set of visited states and perform them together in a group amortizing thus the cost of I/O operations per a single check.

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 less amount of data to be transferred between external and internal memory. Another quite successful improvement builds upon using a Bloom filter maintained in main memory in order to reduce unnecessary I/O operations. Also simple partitioning of states stored on external memory may have impact on the performance of an I/O efficient graph traversal procedure. For more details on these techniques we kindly refer the reader to [9].

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 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 achieve almost double bandwidth that the block of data may be read with from the external memory device. Note that this approach basically improves bandwidth only while 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 external memory devices. This is, however, possible only if the algorithm involved allows parallel processing, which is, for example, the case of breadth-first search, but is not the case of depth-first search [10].

2.2 LTL Model Checking

For accepting cycle detection there is a space efficient optimal algorithm called *Nested Depth-First Search* [11]. Unfortunately, the algorithm becomes rather

Table 1. Characteristics of solid state and hard disk drives

	HDD	SSD
Read Bandwidth	65 MB/s	72 MB/s
Write Bandwidth	60 MB/s	70 MB/s
Random Read Access Time	11 ms	0.1 ms
Random Write Access Time	11 ms	5 ms

inefficient, as soon as states to be stored cannot be maintained in the main memory [10,12].

Recently, three different I/O efficient algorithms for solving the LTL model checking problem have been published [12,13,14]. In [12] the authors suggested to avoid the DFS-based accepting cycle detection by the reduction of the problem to the problem of testing reachability relation [15,16] whose I/O efficient solution was further improved by using the directed A* search and parallelism. Since the reduction to the reachability relation testing may result in 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 [13]. 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 external memory device before it is checked for the presence of an accepting cycle. This disadvantage makes the algorithm quite inefficient in the cases an error can be discovered quickly using some on-the-fly algorithm. Finally, the algorithm given in [14] is both on-the-fly and linear in the space requirements with the respect to the size of the state space.

3 From Mechanical to Solid State Disks

Mechanical hard disks have been around for quite a long time, and they have provided us with reliable service over these years. This is about to change with the advent of *Solid State Disks* (SSD). A solid state disk is electrically, mechanically and software compatible with a conventional (magnetic) hard disk drive. The difference is that the storage medium is not magnetic (like a hard disk) or optical (like a CD) but solid state semiconductor (NAND flash) such as battery backed RAM, EEPROM or other electrically erasable RAM-like chip. In last years, NAND flash memories outpaced DRAM in terms of bit-density [17] and the market with SSDs continues to grow. This provides faster access time than a disk, because the data can be randomly accessed and does not rely on a read/write interface head synchronising with a rotating disk. We list a typical data transfer bandwidth and access time for both magnetic and solid state disk in Table 1.

It became the standard to measure the analytical complexity of an I/O efficient algorithm using the complexity model by Aggarwal and Vitter [1]. However, for solid state disk, the model is no more 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.

4 I/O Efficient Graph Traversal with Solid State Disks

We observe that random read operations on SSDs are substantially faster than on mechanical disks, while other parameters are similar. Therefore, it appears natural to ask, whether it is necessary to employ *delayed duplicate detection* (DDD) known from the current I/O efficient graph algorithms, or it is possible to build an efficient SSD algorithm using the standard *immediate duplicate detection* (IDD), *hashing* in particular.

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 secondary memory only. For this case states are written sequentially to the background memory in the order of generation. For resolving hash synonyms, states lookup random reads are needed. Even though linear probing shows performance deficiencies for internal hashing, for block-wise strategies, it is the apparent candidates. Alternative hashing strategies can reduce the number of random reads.

The third option fosters 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 background 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 background readings and writing, which mainly depend on increasing the locality of the access.

4.1 Hashing

The general setting (see Fig. 1) is a background hash table H_b kept on the SSD, which can hold $m = 2^b$ entries. As said, SSDs prefer sequential writes and sequential read, but can cope with an acceptable number of random reads. We additionally assume a foreground hash table H_f with $m' = 2^f$ entries. The ratio between fore- and background is, therefore, $r = 2^k = 2^{b-f}$. Collisions especially on the background hash table can yield additional burden. As chaining requires overhead for storing and following links, we are left with open addressing and adequate probing strategies.

As linear probing finds elements through sequential scanning, it is I/O efficient. The efficiency analysis goes back to Knuth [18]. For a load factor of α a successful search requires about $1/2\,(1 + 1/(1-\alpha))$ accesses on the average, while an unsuccessful search requires about $LP_\alpha = 1/2\left(1 + 1/(1-\alpha)^2\right)$ accesses

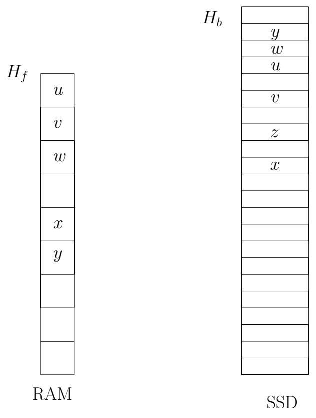


Fig. 1. Fore- and Background Memory, such as RAM and SSD

on the average. For a hash table that is filled up to $\alpha = 50\%$ we have less than three states to look at on the average, which easily fit into the I/O buffer. Given that random access is slower than sequential access, this implies that unless the hash table becomes filled, linear probing with one I/O per lookup per node is an appropriate option for SSD-based hashing.

4.2 Mapping

The simplest method to apply SSDs in graph search is to store each node at its background hash address in a file, and – if occupied – to apply conflict resolution strategy on disk. By their large seek times, this option is clearly infeasible for HDDs, but it does apply to some extent to SSDs. Nonetheless, besides extensive use of random writes that operate block-wise and are, thus, expected to be slow, one problem of the approach is the initialization time, incurred by erasing all existing data stored in background memory.

Hence, we apply a refinement to speed-up search. With one additional bit-vector array kept in RAM, we denote, whether or not a position is already occupied. This limits initialization time to reset all bits in main memory, which is much faster. 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 [19], with conflict resolution on disk. Figure 2 (left) illustrates the approach. The bit-vector *occupied* memorizes, whether the address on the SSD is in use or not.

The extra amount of RAM additionally limits the size of the search spaces to be processed. In search practice with a full state vector of several bytes to be stored in the background memory, however, investing one bit per state in RAM does not harm much, given that the ratio between main and external memory remains moderate. The only limit for the exploration is imposed by the number of states that can be stored on the solid state disk, which we assume to be sufficiently large.

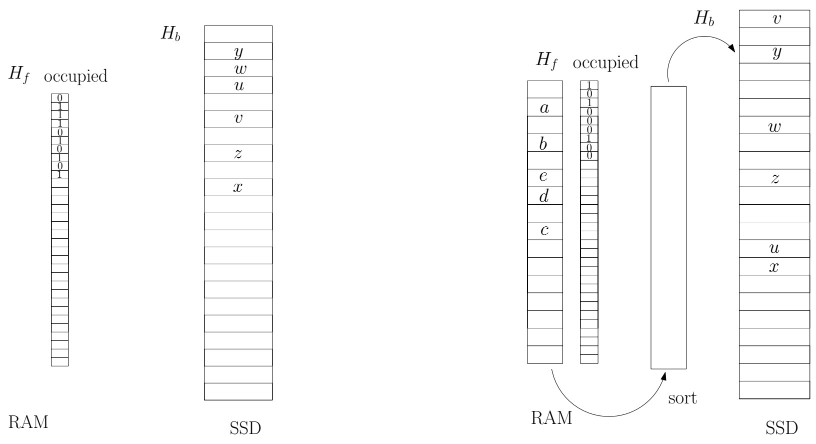


Fig. 2. External hashing without and with merging

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. Without occupied vector requires e lookup and n insert operations. Let B is 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[1] blocks are read for each lookup[2]. 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 a flash 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.

As the main bottleneck of the approach is random writing to the background memory, as another refinement we can additionally employ a foreground hash table as a write buffer. Due to numerous insert operations, the foreground hash table will once become filled, and then has to be flushed to the background, which incurs writes and subsequent reads. One option that we call *merging* is to sort the internal hash table wrt. to the external hash function before flushing. If the hash functions are correlated, the sequence is already presorted, by means that the number of inversions $inv(H_f) = |\{(i,j) \mid h_f(s_i) < h_f(s_j) \wedge\ h_b(s_i) > h_b(s_j)\}|$ is small. If $inv(H_f) = O(m')$ and given that we use an algorithm that exploits presorting[3], we obtain a linear time sorting algorithm. While flushing we 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. Figure 2 (right) illustrates the approach. As we are able to exploit sequential data processing, updating the background hash table

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

[2] at our system $B = 4,096$ bytes, and $|s| \approx 40$ bytes.

[3] e.g. adaptive sort that runs in time $m' + m' \log\left(1 + \frac{inv(H_f)}{m'}\right)$.

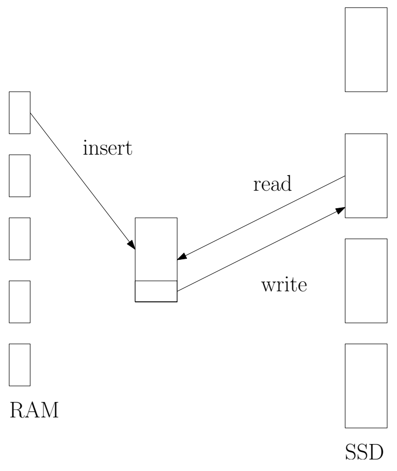


Fig. 3. Updating Tables in Hashing with Linear Probing while Merging

corresponds to a scan (Figure 3). Blocks are read into the RAM and merged with the internal information and then flushed back to SSD.

4.3 Compressing

State compression is a common option in LTL model checking. There are lossless compression strategies like FSM compaction [20], as well as lossy compression strategies like bit-state hashing [21] or hash compaction [22]. For the sake of completeness, in this paper we avoid lossy hash methods as they imply partial state space coverage.

Probably the best lossless compression ratio is obtained using practical perfect hash function [23,24]. Perfect hashing corresponds to an one-to-one mapping of some set S to $\{1, \dots, |S|\}$. Different off-line algorithms [25] have been developed that include perfect hash functions for what has been coined to the term *semi-external* LTL model checking. We do not apply perfect hashing at all, as for the construction of perfect hash functions, set S has to be known, which contradicts the purpose of on-the-fly model checking.

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, that is $m \lceil \log m \rceil$ bits in total. Figure 4 illustrates the approach with arrows denoting the position on external memory. An additional bit-vector *occupied* is no longer needed.

This strategy also results in e lookups 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 $O(LP_\alpha \cdot e)$

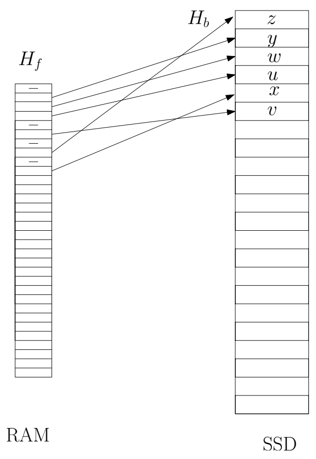


Fig. 4. State Compressing

random reads to the SSD. As long as $LP_\alpha < 2$ this approach performs less random read operations then mapping. By using another internal hashing strategy, e.g. cuckoo hashing [26] one reduces the number of lookups to at most 2. As sequential writing of n states of s bytes requires $n|s|/B$ I/Os, the total flash-memory I/O complexity is $O(LP_\alpha \cdot e + n|s|/B)$.

4.4 Flushing

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

One first solution that we call *padding* is to append the entire foreground hash table as it is to the existing data on the background table. Hence, the background 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 conflict resolution strategy is inherited from the internal memory. For several flushing reading a state for answering membership queries becomes involved, as the search for one state incurs up to r many table lookups. 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. As far as all states can reside in main memory no access to the background memory is needed.

We can safely assume that load factor α is small enough, so that the extra amount of work due to linear probing is transparent by using block accesses. Again e lookups 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 background 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 flash memory complexity of $O(n|s|/B + rp + \sum_{0 \leq i < r} i \cdot e_i)$ I/Os.

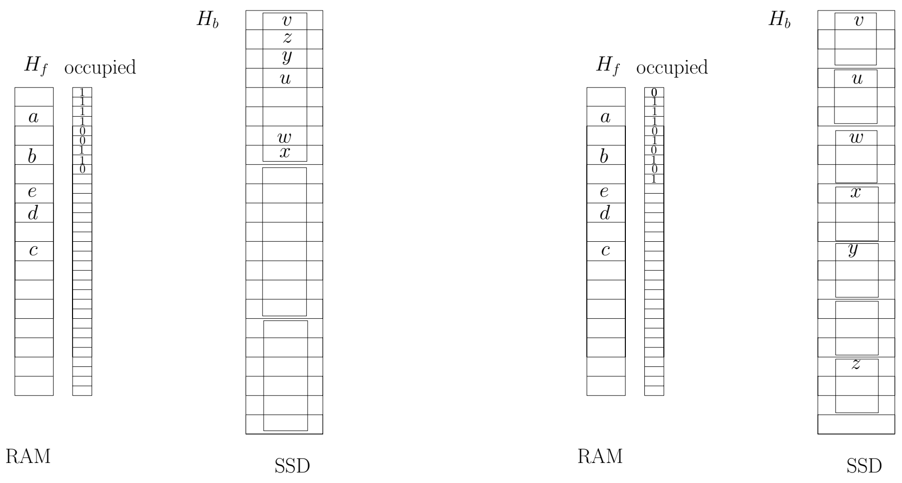


Fig. 5. Padding and slicing

An illustration is provided in Figure 5 (left). The entire foreground hash table has been flushed once, while the maximum number of flushes is set to 3.

The obvious alternative is to *slice* the background hash table such that $h_b(s)$ becomes $h_f(s) \cdot r + i$. An illustration is provided in Figure 5 (right); situation after one flush, and, again, at most 3 flushes are assumed.

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.

5 I/O Efficient Model Checking with Solid State Disks

In Section 4 various implementations of graph traversal with SSD are shown. 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 of RAM, but perform much less I/O operations in the ordinary case (compress strategy).

On the basis of these graph traversals, it is relatively easy to construct LTL model checking algorithms. Nested DFS, as introduced above, can be implemented with two independent hash tables. To save space it is, however, recommended to use one hash table for storing the states and one internal bit-vector array *flagged* to memorize if a state has been visited in the second depth-first search.

With the above hashing schemes, we arrive at full flexibility in applying immediate duplicate detection in Nested DFS. Table 2 summarizes the hash functions applied and the amount of memory required for the different hashing strategies in

Table 2. Trademarks for different hash strategies for on-the-fly LTL model checking algorithm. Upper two lines give an overview of hash functions, lower three lines show a space complexity in bits for different levels in memory hierarchy.

	Mapping	Compressing	Padding	Slicing
h_f	–	$h_d mod\ m$	$h_d mod\ m$	$h_d mod\ m$
h_b	$h_d mod\ m$	$h_d mod\ m$	$i \cdot m' + h_d mod\ m'$	$(h_d mod\ m) \cdot r + i$
RAM	$2m$	$m + m\lceil \log m \rceil$	$2m + m' \times \|s\|$	$2m + m' \times \|s\|$
SSD	$m \times \|s\|$	$m \times \|s\|$	$m \times \|s\|$	$m \times \|s\|$
HDD	$\max_i \|Open_i\| \times \|s\|$	$\max_i \|Open_i\| \times \|s\|$	$\max_i \|Open_i\| \times \|s\|$	$\max_i \|Open_i\| \times \|s\|$

$m = 2^b$, $m' = 2^f$, h_d is hash function in DiVinE [27], m is the size of background hash table (in the number of elements), $|s|$ is state vector size (measured in bits), $Open_i$ is the number of states in the search frontier in iteration i.

LTL model checking. Note that there are recent refinements to Nested DFS [28] that are faster, but need more bits.

6 Experimental Evaluation

We implemented our algorithms in DiVinE (DIstributed VerIficatioN Environment) [27], including only part of the library deployed with DiVinE, namely state generation and internal storage. For the implementation of external-memory container and for algorithms for efficient sorting and scanning we use STXXL (Standard Template Library for Extra Large Data Sets) [29]. Models are taken from the BEEM library [30].

For the first set of 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 3.5" SATA high-speed flash memory solid state disk (HAMA), which has 0.14 ms seek time and scales to about 93 MB/s for sequential reading.

To confirm the theoretical results we check the Rether-4 protocol from the BEEM library (Fig. 6). The plot shows Nested DFS runs with different immediate duplicate detection strategies. All experiments, aside from the *mapping* strategy, were stopped after 40,000s (this strategy was stopped after 1,800s due to its obvious lack of performance). The mapping strategy is the worst one because of numerous random writes. We use padding as a flushing strategy. As linear probing is used to store the positions of the saved states, we observe an increased number of *read* operations as the internal hash table becomes filled. Compress strategy appears to perform the best, which corresponds to its I/O complexity without any penalties for random write operations. 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 hard disk gives another speed-up of about 30% for the state space traversal.

The motivation for use of SSDs was to exploit fast random access to them. Now, we compare new algorithms designed for SSDs to traditional I/O efficient

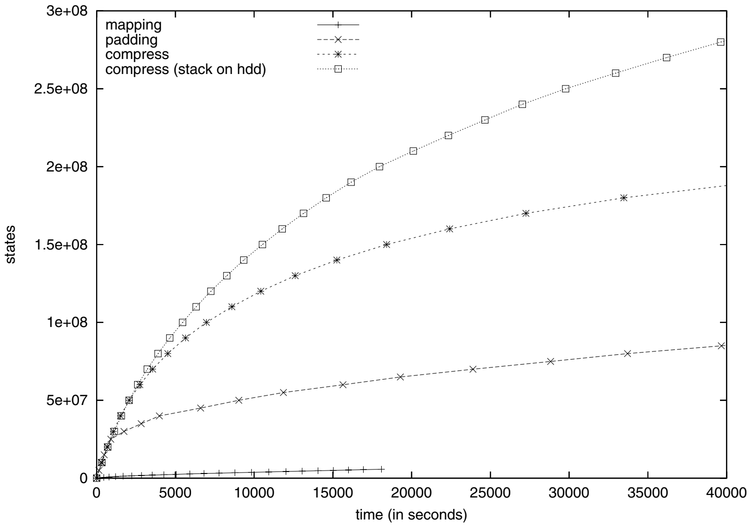


Fig. 6. Comparing the three strategies on the Rether-4 model

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 (Compress BFS). As a traditional approach we run a standard external BFS with delayed duplicate detection after each level (External BFS).

First, the state space of the Szymanski (5 prop4) model was generated using both approaches. The plot in Fig. 7 demonstrates the dependency in expanding speed between the Compress BFS and the BFS layer size, while the expanding time per layer remains almost the same for External BFS. 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 large search depth, immediate duplicate detection saves much time, compared to delayed duplicate detection.

Therefore, it is apparent that results strongly depend on a structure of a state space. Provided that I/O complexity of External BFS is $O((e/m + \#layers)(n|s|/B)$ [13], it is clear that its I/O complexity is highly dependent on the number of BFS layers, while the I/O complexity of Compress BFS is not. This can be demonstrated on the model Rether-2, with 552 BFS layers (see Fig. 8). While External BFS performs poor on this model, Compress BFS 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 disk. This is a case of model Train-Gate, 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. Model MCS is an example, where External BFS performs better – the state space has relatively low number of BFS levels (90).

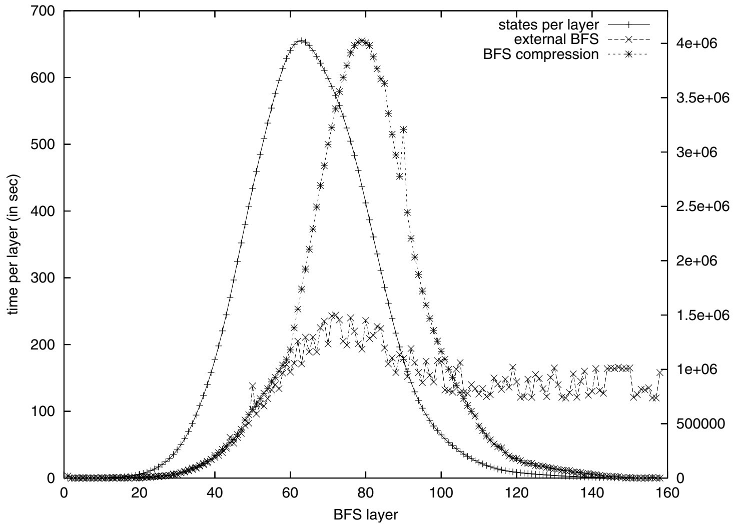
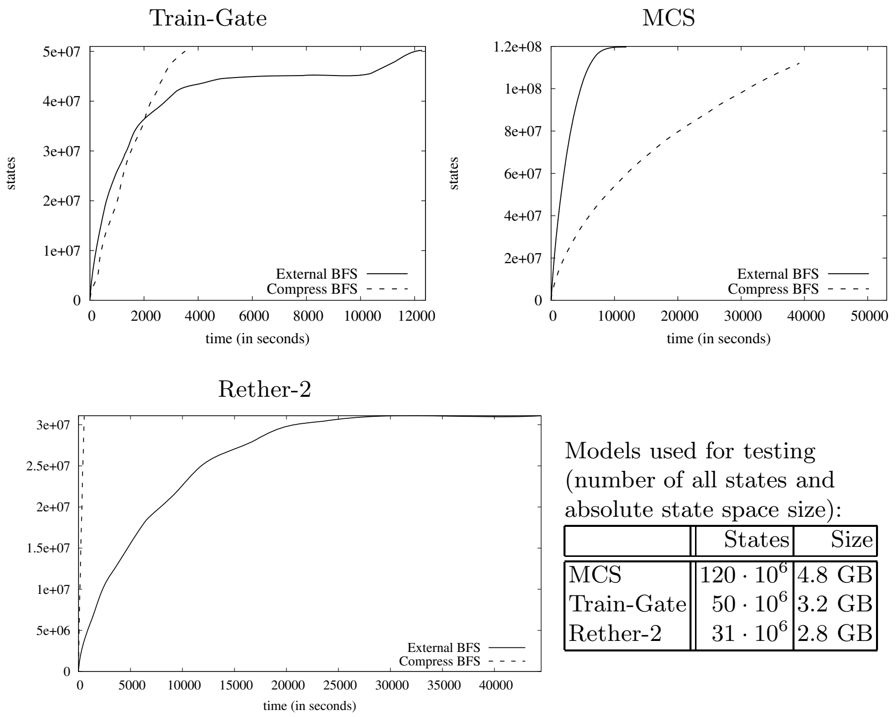


Fig. 7. Comparing Compress BFS to External BFS on the Szymanski 5 prop4 model. The right axis, together with the crossed plot shows the size of each layer. The remaining curves shows time per layer for the different approaches.



Models used for testing (number of all states and absolute state space size):

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

Fig. 8. Comparison of External BFS and Compress BFS

From the I/O complexities of both algorithms and from our measurements it follows that External BFS has to slow down the exploration faster than Compress BFS with increasing portion of the state space explored. Thus, Compress BFS can often outperform it from some BFS level due to its linearity in I/O complexity. The moment, when Compress BFS outperforms External BFS depends to high 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 implemented heuristics from [14] or [9]). Even though it is not easy to predict, whether or from which point of exploration Compress BFS outperforms External BFS, the main impact of behaviour of both algorithms is that there can be a threshold, from which Compress BFS outperforms External BFS on a given input and so algorithms for SSDs like Compress BFS are practical.

7 Conclusions

We have contributed several new approaches to hashing applied to SSDs. The most important observation is with the advent of SSD technology, immediate duplicate detection becomes tractable, offering much more flexibility for the choice of the exploration strategy. Monitoring CPU performance, we observed 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[4].

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, non heuristic algorithms, our SSD hashing strategies can also be applied to heuristic approaches, e.g. A* to rise the amount of states that can be visited. Using SSDs as a shared external storage device for cluster computers will result in an even higher throughput, even for random reads, giving a better possibility for parallel processing.

Directly compared to standard I/O algorithms, for a given model there can be a threshold in state space exploration, from which these new approaches pay off due to their linearity in size of state space – at least for the compress approach. Traditional I/O efficient algorithms are not linear, but they have good constant factors which allow them to outperform new approaches on many inputs. If the bandwidth of SSDs will grow faster, traditional I/O algorithms pay off. If the access time of SSDs will decrease faster than their bandwidth, the importance of new approaches will increase.

Due to easiness of parallel disk connection, large capacities of SSD are possible[5] . Nevertheless, prices for SDDs are still high. Fortunately, in last years they decrease reasonably as the market with flash memories grows.

[4] According to Dell, current prices for 32GB RAM are 6 times higher than for 32GB SSDs.

[5] E.g. StorgeSpire – 1 TB SDD array by Solid Data (`http://www.soliddata.com/products/storagespire`).

References

1. Aggarwal, A., Vitter, J.S.: The input/output complexity of sorting and related problems. Commun. ACM 31(9), 1116–1127 (1988)
2. Sanders, P., Meyer, U., Sibeyn, J.F.: Algorithms for Memory Hierarchies. Springer, Heidelberg (2002)
3. Min, S.L., Nam, E.H., Lee, Y.H.: Evolution of NAND flash memory interface. In: Choi, L., Paek, Y., Cho, S. (eds.) ACSAC 2007. LNCS, vol. 4697, pp. 75–79. Springer, Heidelberg (2007)
4. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (2000)
5. Korf, R.: Best-First Frontier Search with Delayed Duplicate Detection. In: AAAI 2004, pp. 650–657. AAAI Press / The MIT Press (2004)
6. Korf, R., Schultze, P.: Large-Scale Parallel Breadth-First Search. In: AAAI 2005, pp. 1380–1385. AAAI Press / The MIT Press (2005)
7. Munagala, K., Ranade, A.: I/O-Complexity of Graph Algorithms. In: SODA 1999, Philadelphia, PA, USA, pp. 687–694. Society for Industrial and Applied Mathematics (1999)
8. Stern, U., Dill, D.L.: Using Magnetic Disk Instead of Main Memory in the Murphi Verifier. In: Y. Vardi, M. (ed.) CAV 1998. LNCS, vol. 1427, pp. 172–183. Springer, Heidelberg (1998)
9. Hammer, M., Weber, M.: To Store Or Not To Store Reloaded: Reclaiming Memory On Demand. In: Brim, L., Haverkort, B.R., Leucker, M., van de Pol, J. (eds.) FMICS 2006 and PDMC 2006. LNCS, vol. 4346, pp. 51–66. Springer, Heidelberg (2007)
10. Barnat, J.: Distributed Memory LTL Model Checking. PhD thesis, Faculty of Informatics, Masaryk University Brno (2004)
11. Courcoubetis, C., Vardi, M., Wolper, P., Yannakakis, M.: Memory-efficient algorithms for the verification of temporal properties. Form. Methods Syst. Des. 1(2-3), 275–288 (1992)
12. Edelkamp, S., Jabbar, S.: Large-Scale Directed Model Checking LTL. In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 1–18. Springer, Heidelberg (2006)
13. Barnat, J., Brim, L., Šimeček, P.: I/O Efficient Accepting Cycle Detection. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 281–293. Springer, Heidelberg (2007)
14. Barnat, J., Brim, L., Šimeček, P., Weber, M.: Revisiting Resistance Speeds Up I/O-Efficient LTL Model Checking. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 48–62. Springer, Heidelberg (2008)
15. Biere, A., Artho, C., Schuppan, V.: Liveness checking as safety checking. Electr. Notes Theor. Comput. Sci. 66(2) (2002)
16. Schuppan, V., Biere, A.: Efficient Reduction of Finite State Model Checking to Reachability Analysis. International Journal on Software Tools for Technology Transfer (STTT) 5(2–3), 185–204 (2004)
17. Kim, K., Choi, J.H., Choi, J., Jeong, H.S.: The future prospect of nonvolatile memory. In: 2005 IEEE VLSI-TSA International Symposium on VLSI Technology (VLSI-TSA-Tech), pp. 88–94 (2005)
18. Knuth, D.E.: The Art of Computer Programming: Sorting and Searching, vol. 3. Addison Wesley, Reading (1973)
19. Bloom, B.: Space/time trade-offs in hashing coding with allowable errors. Communication of the ACM 13(7), 422–426 (1970)

20. Holzmann, G.J., Puri, A.: A minimized automaton representation of reachable states. International Journal on Software Tools for Technology Transfer 2(3), 270–278 (1999)
21. Holzmann, G.J.: An analysis of bitstate hashing. Formal Methods in System Design 13(3), 287–305 (1998)
22. Stern, U., Dill, D.L.: Combining state space caching and hash compaction. In: Methoden des Entwurfs und der Verifikation digitaler Systeme, 4. GI/ITG/GME Workshop, pp. 81–90. Shaker Verlag, Aachen (1996)
23. Botelho, F.C., Pagh, R., Ziviani, N.: Simple and space-efficient minimal perfect hash functions. In: Dehne, F., Sack, J.-R., Zeh, N. (eds.) WADS 2007. LNCS, vol. 4619, pp. 139–150. Springer, Heidelberg (2007)
24. Botelho, F.C., Ziviani, N.: External perfect hashing for very large key sets. In: CIKM 2007: Proceedings of the sixteenth ACM Conference on information and knowledge management, pp. 653–662 (2007)
25. Edelkamp, S., Sanders, P., Simecek, P.: Semi-external LTL model checking. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 530–542. Springer, Heidelberg (2008)
26. Pagh, R., Rodler, F.F.: Cuckoo hashing. In: Meyer auf der Heide, F. (ed.) ESA 2001. LNCS, vol. 2161, pp. 121–133. Springer, Heidelberg (2001)
27. Barnat, J., Brim, L., Černá, I., Moravec, P., Ročkai, P., Šimeček, P.: DiVinE – A Tool for Distributed Verification (Tool Paper). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 278–281. Springer, Heidelberg (2006)
28. Schwoon, S., Esparza, J.: A note on on-the-fly verification algorithms. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 174–190. Springer, Heidelberg (2005)
29. Dementiev, R., Kettner, L., Sanders, P.: STXXL: Standard template library for XXL data sets. In: Brodal, G.S., Leonardi, S. (eds.) ESA 2005. LNCS, vol. 3669, pp. 640–651. Springer, Heidelberg (2005)
30. Pelánek, R.: BEEM: Benchmarks for Explicit Model Checkers. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 263–267. Springer, Heidelberg (2007)