

Model Checking via Delayed Duplicate Detection on the GPU

Stefan Edelkamp Damian Sulewski
Technische Universität Dortmund, Germany

August 28, 2008

Abstract

In this paper we improve large-scale disk-based model checking by shifting complex numerical operations to the graphic card, enjoying that during the last decade graphics processing units (GPUs) have become very powerful. For disk-based graph search, the delayed elimination of duplicates is the performance bottleneck as it amounts to sorting large state vector sets. We perform parallel processing on the GPU to improve the sorting speed significantly. Since existing GPU sorting solutions like BITONIC SORT and QUICKSORT do not obey any speed-up on state vectors, we propose a refined GPU-based BUCKET SORT algorithm. Alternatively, we study sorting a compressed state vector and obtain speed-ups for delayed duplicate detection of more than one order of magnitude with a single GPU, located on an ordinary graphic card.

1 Introduction

In the last few years there has been a remarkable increase in the performance and capabilities of the graphics processing unit. Modern GPUs are not only powerful, but also parallel programmable processors featuring high arithmetic capabilities and memory bandwidths. Deployed on current graphic cards, GPUs have outpaced CPUs in numerical algorithms like matrix operations, Fourier transformation by orders of magnitudes [32].

The GPU’s rapid increase in both programmability and capability has inspired researchers to map computationally demanding, complex problems to the GPU. Applications include studying the folding behavior of proteins and the simulation of bio-molecular systems [21, 32]. High-level programming interfaces have been designed for using GPUs as ordinary computing devices. These efforts in general purpose programming on the GPU, also known as GPGPU¹ computing, has positioned the GPU as a compelling alternative to traditional microprocessors in high-performance computer systems of the future.

Since the memory transfer between the card and main board on the express bus is about one gigabytes per second, GPUs have become an apparent candidate to speed-up large-scale computations like sorting numerical data on disk [16].

In this paper, we consider the integration of GPUs to accelerate external memory model checking, traversing state spaces that are too large to fit in main memory. Relying on the operating system’s virtual memory management results in memory performance break-downs, known as *thrashing*. The core reason is that duplicate elimination via ordinary hashing does not offer memory access locality. Duplicate elimination has to be delayed [28] by applying sorting-based duplicate detection [20].

In this paper we contribute commonly refined sorting algorithm, using the GPU for accelerated delayed duplicate detection in breadth-first search. Since moving state vectors in the GPU memory is expensive, we illustrate that existing GPU-sorting algorithms designed for floating point data often fail short on sorting state sets. Then, we contribute a new BUCKET SORT algorithm that partially sorts partitioned data on the GPU. We hook the algorithm onto an explicit-state model checker and show in experiments that this sorting strategy yields a significant speed-up in analyzing communication protocols. The paper is structured as follows. First, we provide a brief review to external memory model checking and the issue of delayed duplicate detection. Next, we reflect the complexity model behind GPU computing and advances to GPU programming. Then we turn to strategies for delayed duplicate detection on the GPU, including hash-based partitioning, state compression, and array compaction. Finally, we present empirical results and discuss future research avenues.

¹www.gpgpu.org

2 External Memory Model Checking

LTL model checking amounts to search a graph for the presence of an accepting cycle. This graph is generated on-the-fly. To prevent revisiting of already explored states, all states that have been processed are stored. If a state is generated, it is first checked against the set of stored states.

Due to the huge number of states, their large size, and the speed of generating them, the memory demands for analyzing systems rise rapidly. To release internal memory, states have to be written to the external memory. A check, whether or not a state has been visited, now involves I/O as not only the states stored in memory, but also the states stored on the external memory device have to be considered. An important aspect is that the data is accessed in blocks.

Different I/O-efficient algorithms for solving the LTL model checking problem have been published [11, 4, 3]. In [11] the authors avoid DFS-based accepting cycle detection by reducing the liveness problem to a safety one [33]. This I/O-efficient solution was further improved by using A* search and parallelism. Another disk-based algorithm for LTL model checking [3] avoids the expensive increase in space, but does not operate on-the-fly. The algorithm given in [4] is both on-the-fly and linear in the space requirements with the respect to the size of the state space, but its worst-case time complexity is large.

Some recent model checking algorithms [13, 14] have been developed that include perfect hash functions for what has been coined to the term *semi-external* LTL model checking. After generating the state space externally, a space-efficient perfect hash function [7] is constructed that re-invents immediate duplicate detection and the application of depth-first model checking algorithms. It is also possible to reduce time, if, instead of the serial I/O-efficient algorithm, working over a single external device, a parallel I/O-efficient algorithm is used, exploiting multiple external devices. This is, however, possible only if the algorithm involved allows parallel processing [1].

3 Delayed Duplicate Detection

The main technique for I/O-efficient search algorithms has been coined to the term *delayed duplicate detection* [23] and goes back to [28]. The idea is to postpone individual checks against the set of visited states and to perform

them together in a bulk operation; while amortizing the cost of I/O.

There are different external memory search strategies including external memory breadth-first and A* search [12]. External memory A* partitions state sets in each breadth-first layer residing on disk wrt. the expected error distance. As delayed duplicate detection remains the crucial issue, in this paper we restrict to external memory breadth-first search.

There are two stages within delayed duplicate detection. First duplicates within one layer have to be eliminated, a procedure that we refer to as *compaction*. Next duplicates with respect to previous layers have to be eliminated, a procedure that we refer to as *subtraction*.

The additional efforts for eliminating duplicates late slows down the verification, so that alternatives for accelerated external memory state space generation have been studied; resisting revisiting states in large search depths [4], dynamically predicting elimination efforts [15]; or layered, while sacrificing completeness [27]. Solid-state disks allow more flexibility in the choice of search algorithms; they reinvent immediate duplicate detection [2], and – in combination with semi-external algorithms – they accelerate the search with perfect hash functions [14].

Delayed duplicate detection as proposed in this paper is a new compromise, manifesting a trade-off between sorting-based duplicate detection [28] and hash-based delayed duplicate detection [26]. The strategy we propose sorts buckets that have been filled through applying a hash function.

3.1 Sorting-based Delayed Duplicate Detection

The main aspects of not using full sorting-based delayed duplicate detection are the larger efficiency gains on the GPU. If the buckets are small, they fit into local memory and can be processed in parallel. Hashing covers distant move operations while sorting addresses the local changes.

Instead of sorting the buckets after they have been filled, it is possible to use chaining, checking each individual successor for having a duplicate against the states stored in its bucket. Keeping the list of states sorted, as in ordered hashing [22], accelerates the search, but may require some additional work for insertion. Viewed from a different angle of delayed processing, ordered hashing works similar to INSERTIONSORT. Implemented with binary search in an array of N elements, INSERTIONSORT calls for $\lceil \log_2(N!) \rceil$ state comparisons and $\Theta(N^2)$ state moves in the worst case. As this is a considerable amount of computation, ordered hashing in the buckets is not expected

to significantly accelerate the computation, if compared to parallel sorting the buckets on the GPU.

3.2 Hash-based Delayed Duplicate Detection

Hash-based delayed duplicate detection has successfully been applied to puzzles like the Towers of Hanoi problem [25]. For the example of the 30-disc 4-peg Towers-of-Hanoi problem the approach divides the state uniquely by the location of the discs. Using two bits per disc this gives a total of 60 bits. The discs are divided into the 16 largest and 14 smallest discs. States are written to the file based on the position of the 16 largest discs. Thus, all states in any given file have the 16 largest discs in identical positions, and any set of duplicate nodes must be confined to the same file. This allows to read one file at a time into a hash table in memory, detect and merge any duplicate nodes in that file, and write out just one copy to an output file.

A similar approach has been applied to the 15-Puzzle [26]. The representation of a state in the 15-Puzzle corresponds to 64 bit, but by the disjoint and orthogonal hash functions that take the first few tiles and the remaining tiles, the space required for the entire puzzle could be reduced to about 1.4 terabytes.

For hash-based delayed duplicate detection to work there are some implicit assumption that we will try to make explicit to illustrate the differences we have for model checking domains. Consequently, we formalize different characteristics of hash functions.

Definition 1 (Hash Function) *A hash function h is a mapping of a universe U to an index set $[0..m - 1]$.*

The set of reachable states S is a subset of U , i.e., $S \subseteq U$. The first aspect are hash functions that are injective.

Definition 2 (Perfect Hash Function) *A hash function is perfect, if for all $s \in S$ with $h(s) = h(s')$ we have $s = s'$.*

Given that the entire state viewed as a bit-vector can be considered as a number, one design of a perfect hash functions is immediate. In contrast the space requirements of the corresponding hash table can be tremendous. If the hash function is one-to-one, we call it a minimal perfect hash function.

Definition 3 (Minimal Perfect Hash Function) *A perfect hash function is minimal, if $|\{h(s) \mid s \in S\}| = |S|$.*

Minimal perfect hash functions allow direct-addressing in a bit-state hash table instead of an open-addressed or chained hash table. The index uniquely identifies the state and states can be packed into 1 bit. In the puzzle domains minimal perfect hash functions are known, but for general state vectors in model checking domains, one-to-one correspondences are rare. As already mentioned there are practical (minimal) hash functions [7], mapping the state space $S \subseteq U$ to $[0..|S| - 1]$. However, this approach itself requires additional space and is only applicable if the state space has been seen before.

One other important property of a perfect hash function for state expansion is that the state vector can be reconstructed given the hash value.

Definition 4 (Invertible Hash Function) *A perfect hash function h is called invertible, if given $h(s)$, $s \in S$ can be reconstructed. The inverse h^{-1} of h is a mapping from $[0..m - 1]$ to S .*

An example for invertible mappings are Myvold and Ruskey’s rank and unrank functions that apply to permutation vectors as apparent in the 15-Puzzle [29]. Unfortunately, invertible perfect hash functions are difficult to obtain for model checking state spaces.

For hash-based delayed duplicate detection, orthogonal hash functions are helpful.

Definition 5 (Orthogonal Hash Functions) *Two hash functions $h_1 : U \rightarrow [0..m_1 - 1]$ and $h_2 : U \rightarrow [0..m_2 - 1]$ are orthogonal, if for all $s, s' \in U$ with $h_1(s) = h_1(s')$ and $h_2(s) = h_2(s')$ we have $s = s'$.*

The hash functions for Towers of Hanoi are orthogonal. This allows to take the first hash function as a file index and the second one to merge the files. The following result refers to a straight-forward, but important observation.

Theorem 1 (Orthogonal Hash Functions imply Perfect Ones) *If the two hash functions $h_1 : U \rightarrow [0..m_1 - 1]$ and $h_2 : U \rightarrow [0..m_2 - 1]$ are orthogonal, their concatenation (h_1, h_2) is perfect.*

Proof: We start with two hash functions h_1 and h_2 . Let s be any state in U . Given $(h_1(s), h_2(s)) = (h'_1(s), h'_2(s))$ we have $h_1(s) = h_1(s')$ and $h_2(s) = h_2(s')$. Since h_1 and h_2 are orthogonal, this implies $s_1 = s_2$. \square

In other words, if assigned to buckets according to the first hash value, it suffices to look for duplicate in each file according to the second hash value. For reducing the size of the state vector to be addressed in the second hash, the following property is desirable.

Definition 6 (Disjoint Hash Functions) *Two hash functions h_1 and h_2 on state vectors $s = (s_1, \dots, s_n)$ are disjoint, if $h_1(s) = h_1(s_{i_1}, \dots, s_{i_k})$ and $h_2(s) = h_2(s_{i_{k+1}}, \dots, s_{i_n})$ and $k \in \{i_1, \dots, i_n\} = \{1, \dots, n\}$.*

The orthogonal hash functions for the Towers of Hanoi problem are disjoint. This reduces the size of the state description. For example, representing a complete state of the 30-disc four peg Towers of Hanoi problem requires 60 bits. If all the states have their 16 largest discs on the same pegs and their positions is encoded in the filename then only 28 bits are required to identify the positions of the 14 smallest discs. Note that the theoretical maximum of 4^{16} files does not occur, since the symmetry of the state space is explored to essentially cut down the search depth to a half [25].

What makes the difference of hash-based delayed duplicate detection for the mixture of hash and sorting-based delayed duplicate detection that we propose? Asked differently, why haven't we applied hash-based delayed duplicate detection as in the puzzles?

First, in model checking domains we usually have to consider non-invertible hash functions. Such one-way functions urge us to store the list of frontier states on disk, as full state information is needed for expansion.

Neglecting these additional space requirements for storing the search frontier, we are back to storing the visited state set for duplicate detection. We apply a compression to a 64 bit vector by applying two supposedly orthogonal hash functions on 32 bits. As some theoretical observations will indicate, the approach of using a 64 bit hash key might result in collisions, but with an acceptable low probability.

Assuming a lossless compression to 64 bits, there is still the question of why we haven't used hash-based duplicate detection on the 64 bit state vectors. The answer is found by looking at the practical space consumption of the approach. One first issue is that our hash functions are not disjoint,

so that we cannot truncate the state vector for computing the second hash and storing a reduced state vector.

When generating the successors, we either have to append all states to the file, or to set a bit in the file to denote that the state has been visited. The latter approach incurs too many I/Os, so that we are forced to use the former one.

Partitioning the vector of 64 bits into 32 bits for the file address and 32 bits for the remaining state, yields a space requirement of 4 GB RAM for the direct-access bit-vector for duplicate detection in RAM, which might be available. However, maintaining 2^{32} files in total is way too much for reasonable disk sizes. If we reduce the number of files to say 2^{16} , storing a 2^{48} sized bit-vector in RAM becomes infeasible. It urges an internal chained representation of the states in RAM, which has an even worse memory reputation than direct access.

Given the unsatisfied strong set of assumption on the hash functions, hash-based delayed duplicate detection as proposed by [26] seems to be out of reach for model checking. Moreover, under the strong assumption of invertible and minimum perfect hash functions prior to the search, alternative approaches are applicable. For example, a two-bit breadth-first search as proposed by [24] does not store the state vector at all.

As a result, the time-efficient generation of the state space on disk with delayed duplicate removal, remains a challenge to verification, so that GPUs provide an interesting alternative.

4 Efficient GPU Algorithms

Future microprocessor development efforts will concentrate on adding cores rather than increasing single-thread performance. One example of this trend is the main processor in the Sony Playstation 3, which has attracted substantial interest from the scientific computing community.

Similarly, the highly parallel graphics processing unit has rapidly gained maturity as a powerful engine for computationally demanding applications. The GPU's performance is promising, yet the architecture and programming model of the GPU are markedly different from most multi-core processors. As the GPU model has shared memory but forbids common writes to a memory cell, the computational model of the graphic card is closest to the CREW (Common Read Exclusive Write) PRAM [10]. Many efficient algorithms on

this model like computing prefix sums [17] transfer to GPU programming.

The significant difference between shared memory usage in multi-core architectures and GPUs is apparent, as GPU access is streamed using a kernel function given to every processing unit. As a result Holzmann and Bosnacki’s multi-core parallelization of SPIN via slicing the depth-first search tree at certain hand-off depth [18] hardly transfers to GPU programming. However, GPUs are effective for external memory search, assuming that hierarchical memory structure of SRAM (small, but fast and parallel access) and VRAM (large, but slow access) is used. The setting is displayed in Fig.1.

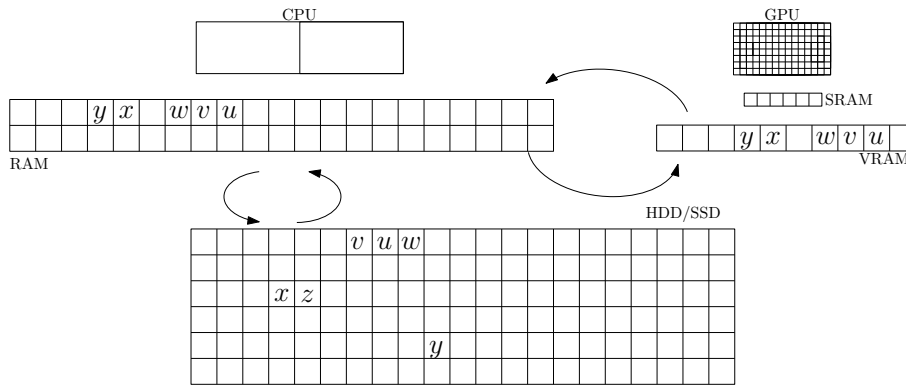


Figure 1: External Memory Search with GPU.

Disk-based sorting with the help of GPU processing refers to one of the major success stories on the GPU, most notably illustrated by TERA SORT [16], which used GPU-based code to win the 2006 Indy PennySort category of the TeraSort competition, a sorting benchmark testing performance for database operations. Since then, various GPU implementations have been proposed, including MP5 SORT², a variant of BITONIC SORT [5], and GPU QUICKSORT³ [9].

5 Delayed Duplicate Detection on the GPU

For delayed elimination of duplicates, we have to order a BFS layer wrt. a comparison function that operates on states (sorting phase). The array is then scanned and duplicates are removed (compaction).

²courses.ece.uiuc.edu/ece498/al1/mps/MP5-TopWinners/kaatz/MP5-parallel.sort.zip

³www.cs.chalmers.se/~dcs/gpuqsortdcs.html

Depending on the external memory model checker in use, sorting consumes a considerable amount of run-time⁴, such that accelerating sorting results also in large savings also in overall run-time. In model checking the vectors can be interpreted either as bit, byte or integer vectors. We chose integer vectors, as the experiments showed better savings. We then extended both mentioned GPU sorting algorithms by including a state vector comparison function (similar to string comparison) with one integer operation per state vector entry. The obtained run-time results were disappointing. Even after further refinements the best speed-up we could achieve wrt. to CPU QUICKSORT was about 1.2. The first reason is that state vectors have to be moved within the VRAM, which slows down the computation significantly. Trying to sort an array of indexes also failed badly, as now the comparison operator exceeds the boundary of the thread memory. The lesson to be learned is that for effective GPU-sorting the sorted elements should be as short as possible, and that sorting has to stay local. This leads to new designs of GPU sorting algorithms for model checking domains.

5.1 Hash-based Partitioning

We observed that in MP5 SORT, the first phase of sorting states in smaller blocks is fast, while merging the pre-sorted sequences into a totally ordered sequence slows down the performance drastically. Therefore, we employed hash-based partitioning on the CPU to distribute the elements into buckets of adequate size (see Fig.2).

The state array to be sorted is scanned once. Using hash function h and a distribution of the graphic card memory into k blocks the state s is written to the bucket with index $h'(s) = h(s) \bmod k$, while updating the number b_1, \dots, b_k of states in a bucket. The number of states that fit into a bucket is $b = \lfloor \text{SRAM} / l \rfloor$. More precisely, we have $b = 2^j$ with j being the largest value such that $2^j < \lfloor \text{SRAM} / l \rfloor$.⁵ If the distribution of the hash function is good and the maximal bucket sizes are not too small, we observe that if one of the buckets encounters an overflow, the table is filled with more than 50% of the hash table size.⁶ All remaining elements are set to a pre-defined illegal state

⁴For DiVinE that is about 60%, depending on the model, for other model checkers like SPIN, the ratio is higher, as the per node performance is often better

⁵As a implementation refinement we do not maintain b_1, \dots, b_k but the position of each last element c_1, \dots, c_k with $c_i = i \times b + b_i$.

⁶As another refinement we do not insert an element to a bucket, if it is already on top.

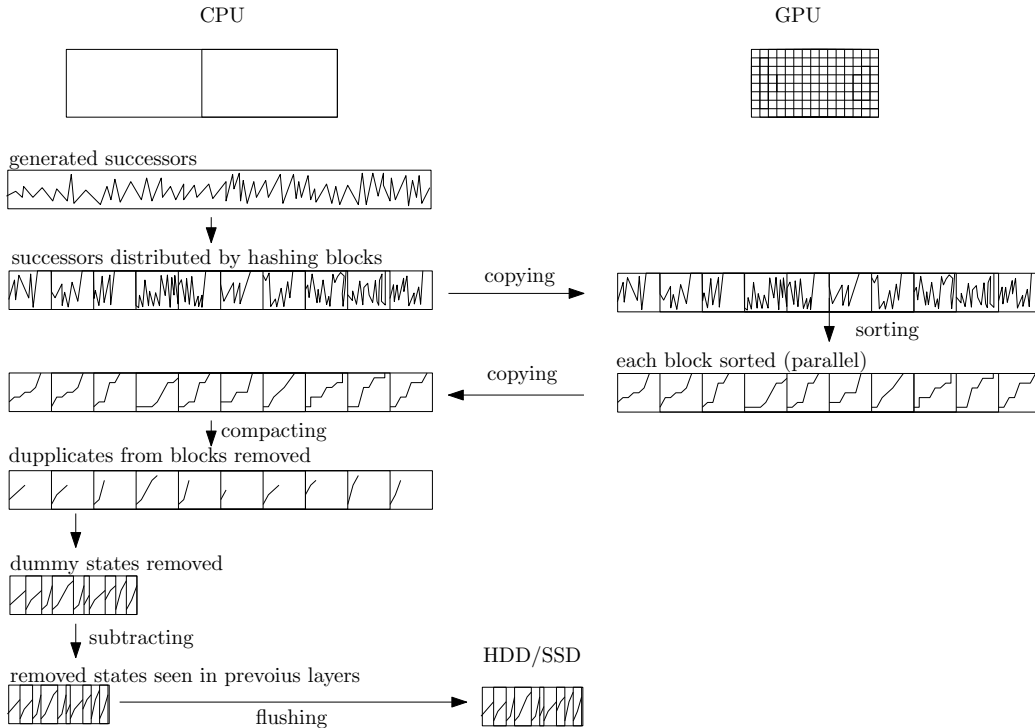


Figure 2: Hash-based Partitioning.

vector that realizes the largest possible value in the ordering of states.

This hash-partitioned vector is copied to the graphic card and sorted by the first phase of MP5 SORT. The crucial observation is that – due to the presorting – the array is not only partial sorted wrt. to the comparison function operating on states s , but totally sorted wrt. to the extended comparison function operating on the pairs $(h'(s), s)$. The sorted vector is copied back from VRAM to RAM, and the array is compacted by eliminating duplicates with another scan through the elements. Subtracting visited states is made possible through scanning all previous layers residing on disk. Finally, we flush the duplicate-free file for the current BFS layer to disk and iterate. To accelerate discrimination and to obey the imposed order on disk, the hash bucket value $h'(s)$ is added to the front of the state vector s .

We observed that subtraction even with respect to the entire set of existing layers is fast in practice, due to the large number of successors that are eliminated within one layer, so that successor generation and sorting are the main performance bottlenecks, with sorting being the hardest one in large

instances. As far as the files in breadth-first or best-first search do not exceed the memory on the GPU, the above exploration strategy is efficient. For this case, however, no external memory would be needed at all, given that the RAM resources usually exceed the VRAM resources. Under this circumstance, immediate duplicate detection in the RAM applies, questioning accelerating sorting-based duplicate detection.

If a BFS layer becomes too large to be sorted on the GPU, we split the search frontier into parts that fit in the VRAM. This yields some additional state vector files to be subtracted to obtain a duplicate-free layer, but in practice time performance is still dominated by expansion and sorting. For the case that subtraction becomes harder, we can exploit the hash-partitioning, inserting previous states into files partitioned by the same hash value, a technique with strong relation to hash-based duplicate detection [26] and structured duplicate detection [35]. States that have a matching hash value are mapped to the same file. Provided that the sorting order is first on the hash value then on the state, after the concatenation of files (even if sorted separately) we obtain a total order on the sets of states. This implies that we can restrict duplicate detection including subtraction to states that have matching hash values. Another alternative is to compute hash values of states on the GPU.

5.2 A Note on the Complexity

We would like to compare our hash-based sorting algorithm with CPU QUICKSORT. It is well-known that CPU QUICKSORT for N elements requires about $1.386N \log N - 2.846N + O(\log N) = \Theta(N \log N)$ comparisons on the average. There are better algorithms in theory, but simple refinements like CLEVER QUICKSORT with $1.188N \log N - 2.255N + O(\log N)$ comparisons on the average are hard to beat in practice [8]. Hence, sorting layer i having e_i states thus results in $O(e_i \log e_i)$ comparisons. The total sorting complexity results in $\sum_i O(e_i \log e_i) = O(e \log e)$, with e being the number of edges in the state space graph. If the state vector size is denoted by l , one element comparison (or movement) may induce $O(l)$ elementary operations.

On the GPU, we have a fixed amount of $O(|\text{VRAM}|/|\text{SRAM}|)$ group operations, where each group is sorted by BITONIC SORT. In our GPU sorting algorithm, the sorting complexity is independent from the number of elements to be sorted, as in each iteration the entire vector is processed. With a good distribution function, we assure that on the average each bucket

is at least 50% filled with successor states, such that we loose less than factor 2 by not dealing with entirely filled buckets.

As an example, in our case, we have $|\text{VRAM}| = 1 \text{ GB}$, and $|\text{SRAM}| = (16 - c) \text{ KB}$, where c is a small constant, imposed by the internal memory requirements of the graphics card. For a state vector of 32 byte, we arrive at $k = 256$ elements in one group. Within each group BITONIC SORT is applied, known to induce $O(k \log^2 k)$ work and $O(\log k)$ iterations. In each iteration the number of comparisons that can be executed in parallel depends on the number of available threads t , which in turn depends on the graphic card chosen. In our setting we work with $t = 128$ threads, such that one scan induces time for $(256/2)/t = 1$ comparison.

Similar to the time complexity of generating at most e_i successors in layer i , distributing them into groups is a linear scan through the data and is bounded by time $O(e_i)$ times the state vector size l . If they fit into VRAM and assuming a sufficient number of threads to process a group in parallel, the number of comparisons for sorting on the GPU is logarithmic in the group size and linear in the number of groups and is thus bounded by $O(e_i \log k)$ operations, which cumulates to a value bounded $O(e \log k)$ operations for the total amount of sorting. This is considerably smaller than $O((e+l) \log(e+l))$ obtained for CPU QUICKSORT. Note that the complexities are coarse estimates, as there are many other factors that play a role in sorting algorithms. For example, given its recursive structure, CPU QUICKSORT is known to have a very good cache performance.

5.3 State Compression

State compression is a hot topic in model checking research. Compression may be lossy, by means that false positives may arise. This is often the case for (single, double or triple) bit-state hashing [19] and with hash compaction to a smaller number of bits [34]. In our case we apply two independent 32-bit hash values from a set of universal hash functions that have been designed for cuckoo hashing [30]. The core observation was that with this 64 bit hash address we did not encounter any collision even in very large state spaces.

Henceforth, given hash functions h_1 and h_2 , we compress the state vector for s to $(h_1(s), h_2(s), a(s))$, where $a(s)$ is the index of the state vector residing in RAM that is needed for expansion. We sort the triples on the GPU with respect to the lexicographic ordering of h_1 and h_2 . The shorter the state vector, the more elements fit into one group, and the better the expected

speed-up on the GPU. We participate in the fact that GPUs are designed for 3D vectors, such that assigning, comparing, and swapping such 3D vectors is fast.

If we insist on a complete model checking algorithm, during the compaction phase we have to check each state vector duplicate with address pair $(a(s), a(s'))$ and $(h_1(s), h_2(s)) = (h_1(s'), h_2(s'))$ for $s = s'$. This equality check is costly, even if we can obtain it while scanning the data once.

Therefore, we estimate the probability of an error. We assume a state space of $n = 2^{30}$ elements uniformly hashed to the $m = 2^{64}$ possible bit-vectors of length 64. According to the birthday problem [6], the probability of having no duplicates is $m!/(m^n(m-n)!)$. One known upper bound is $e^{-n(n-1)/2m}$, which in our case resolves to .9692, such that we have a less than 96.92% chance of having no collision. But how much less can this be? For better confidence on our algorithm, we need an lower bound. We have

$$\begin{aligned} m!/(m^n(m-n)!) &= m \cdot \dots \cdot (m-n+1)/m^n = m/m \cdot \dots \cdot (m-n+1)/m \\ &\geq ((m-n+1)/m)^n \geq (1-n/m)^n. \end{aligned}$$

For our case this resolves to

$$(1 - 2^{-34})^{2^{30}} = (.99999999994179233909)^{1073741824}.$$

If we set $x = .99999999994179233909$ and rewrite the equation into the equivalent $(x^{10737})^{100000} + x^{41824}$ to please our arbitrary-precision calculator, we arrive at a confidence of at least 93.94% that no duplicate arises while hashing the entire state space to 64 bits. Recall, that missing a duplicate harms, only if the missed state is the only way to reach the error in the system. If the above confidence appears still to be too low, one may re-run the experiment with another independent hash function, certifying that with more than a 99.6% chance, no false positive has been produced, while traversing the entire state space.

5.4 Array Compaction on the GPU

The compaction operation can be accelerated on a parallel machine using an additional vector *unique* as follows. The vector is initialized with 1s, denoting that we initially assume that states are unique. In a parallel scan, we then compare a state with its left neighbor and mark the ones that are not unique by setting its entry to 0. Next, we compute the prefix sum, which

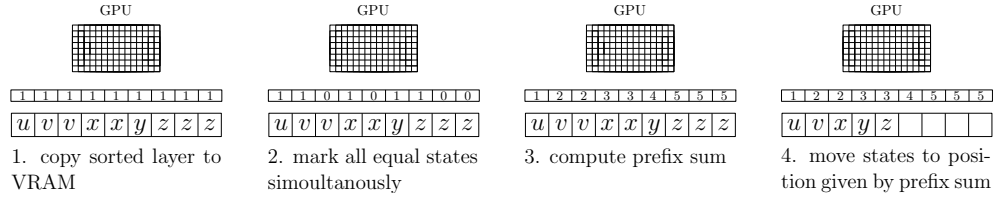


Figure 3: Stages in the compaction phase in the GPU.

can efficiently be parallelized (see Fig.3). The prefix sum denotes the array index in the compacted representation.

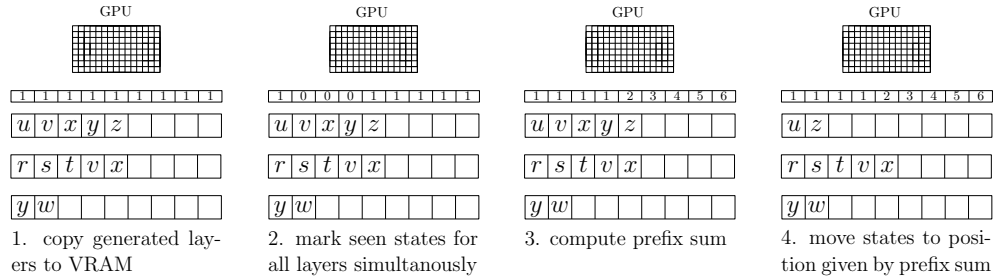


Figure 4: Stages in the subtraction phase in the GPU.

Duplicates wrt. to previous layers can be integrated into this process and eliminated in parallel as follows. First we map as many BFS layers as possible into the GPU. Processor p_i scans the current layer t and a selected previous layer $i \in \{0, \dots, t - 1\}$. As both arrays $Open(t)$ and $Open(i)$ are sorted, the time complexity for the parallel scan is acceptable, as the arrays have to be mapped from RAM to VRAM and back anyway. If a match is found, array *unique* is updated, setting the according bit in $Open_t$ to 0 (see Fig.4). The parallelization we obtain for level subtraction is on processing different BFS layers, while parallelization for sorting and scanning are on partitioning the array. As all processors read the array $Open_t$ we allow concurrent read. Additionally, we have allowed each processor to write into the array *unique*. Since previous layers are mutually disjoint, no processor will access the same position, so that the algorithm preserves exclusive writes.

6 Experiments

We implemented our algorithms in DiVinE (DIstributed VerificatiON Environ-ment)⁷, including only parts of the library deployed with DiVinE, namely state generation and internal storage. Models are taken from the BEEM library [31]. We used a NVIDIA GeForce 8800 GT MSI graphic card with 1 GB VRAM having 112 stream processors and a 256-bit frame buffer. The C-like programming language CUDA applies kernel functions to the GPU and links to ordinary C-sources of the model checker. Moreover, RAM amounts to 4 GB, SSD (HAMA SATA 3,5”) to 32 GB, and the SATA hard disk encompasses 300 GB.

For comparing delayed duplicate detection strategies, we implemented different sorting strategies: The in-built CPU QUICKSORT implementation (qsort CPU), the GPU QUICKSORT implementation of [9] (qsort GPU) and the MP5 SORT routine⁸, all adapted to sort state vectors instead of numbers. As implementation refinements, we apply hash partitioning and state compression to MP5 SORT, abbreviated with MP5 hash and MP5 compress.

Figure 5 compares the total run-time of the model checker subject to various CPU- and GPU-based algorithms for the delayed elimination of duplicates. On the x-axis CPU time is drawn, while on the y-axis the covered part of the state space is referred to. We observe hash-partitioned and compressed MP5 SORT induce the model checker to perform consistently better than with GPU and CPU QUICKSORT. The impact is even more obvious on the larger Telephony instance displayed in the graph on the bottom. Here, we stopped qsort-CPU after 33,000 seconds, when it had even not generated half of the state space.

We validate that exploration gains are limited by the time to generate successors. If we restrict to the impact of the sorting, the picture becomes much clearer. Fig.6 displays the gains of GPU-based duplicate detection for a compressed state vector. We see that we obtain a speed-up of up to Factor 14 for the szymanski protocol with respect to internal sorting. Note that to the begin of the exploration for small layers with less than 300,000 expanded nodes, we stick to internal sorting, while for small layers to the end of the exploration we have not switched back to internal sorting. This explains, why QUICKSORT is superior in the BFS layers 110-160 for Szymanski. The

⁷anna.f.muni.cz/divine

⁸We have not included the original MP5 SORT algorithm to our data, as it performs worse than GPU QUICKSORT or CPU QUICKSORT

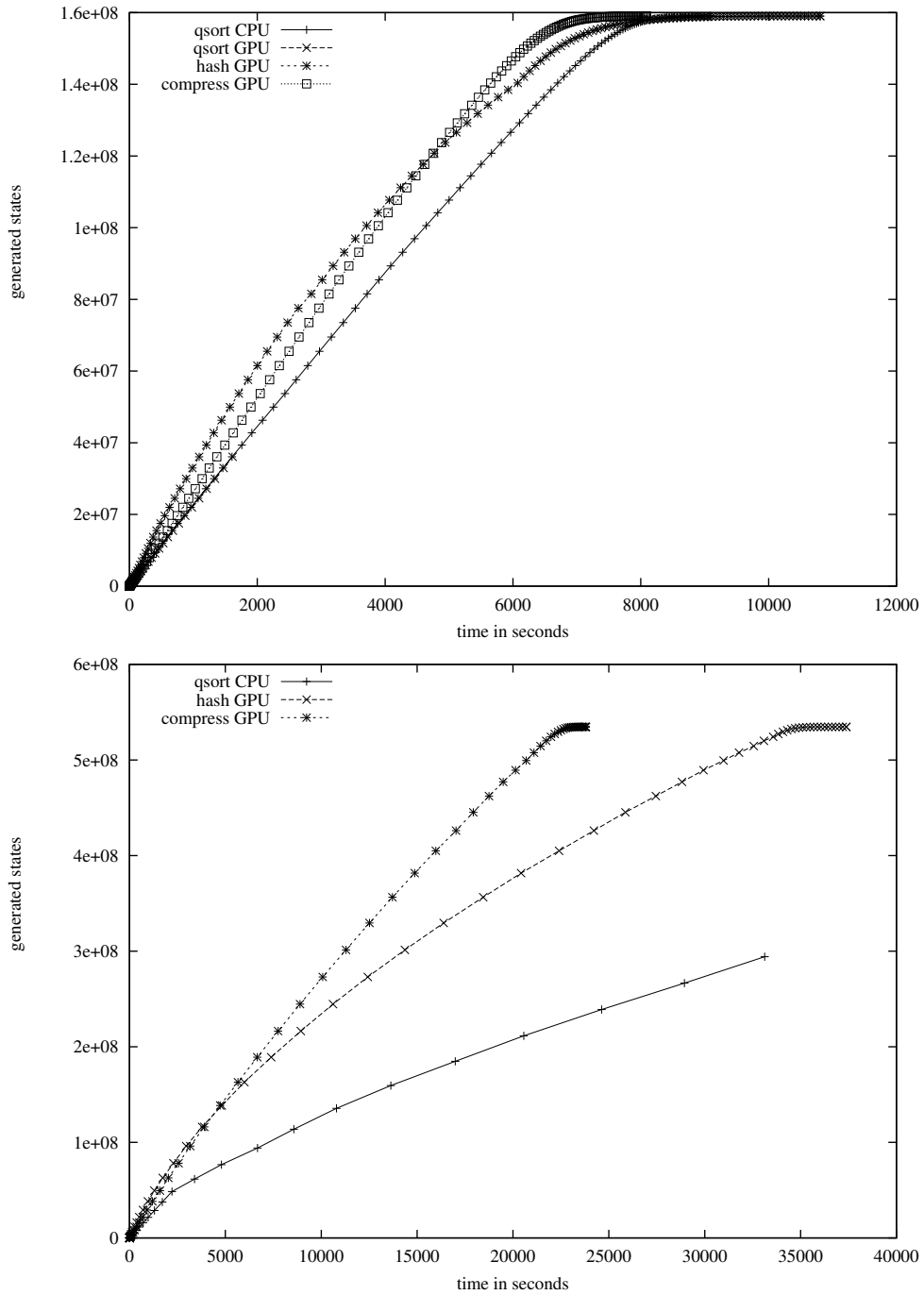


Figure 5: Comparing GPU-based with CPU-based performance for Delayed Duplicate Detection on the Szymanski 5 P4 (top) and the Telephony 5 (bottom) protocol.

Telephony protocol, having larger BFS layers, benefits even more from the GPU. The speed-up is 24 (we even had to change to logarithmic scale on the second graph to make the sorting time visible).

Two other protocols for which the state space was explored successfully were the At.7 protocol with 819,243,818 states expanded in 28,048s and the Anderson.8 protocol with 538,699,030 states expanded in 31,172s. We applied the compression strategy (compress GPU).

7 Conclusion

Parallelism is the future of computing. We have seen a significant improvement of delayed duplicate detection in external memory model checking by integrating GPU computation to the sorting process. The speed-up of more than one order of magnitude on a single, only moderately advanced graphic card is significant, and should further increase on more recent or multiple cards.

The remaining bottleneck is the CPU performance in generating the successors, which can be reduced by applying parallel computation. In the future, we therefore aim at porting the expansion of state sets to the GPU, parallelizing the entire model checking process. Most external memory algorithms that have discussed in literature are streamed, giving rise to the notation of write and read buffers. This makes external memory algorithms simpler to be ported to the GPU in order to share the efforts for expansion among all participating processors. More importantly, the order of expansions within one bucket does not matter, so that almost no communication between threads is required. Each processor simply takes its share of work and starts expanding. Nonetheless, there are some subtleties to be resolved. First, the state size is dependent of the model to be checked and may also vary during the verification. Fortunately, common model checkers provide upper bounds on the state vector (e.g. SPIN), or induce the maximal size of the state vector after the initial state has been read (e.g. DiVinE). Another technical problem is that the GPU kernel - though being C-like - does not link directly to the sources of the model checker, such that all sub-procedures have to be ported to the GPU.

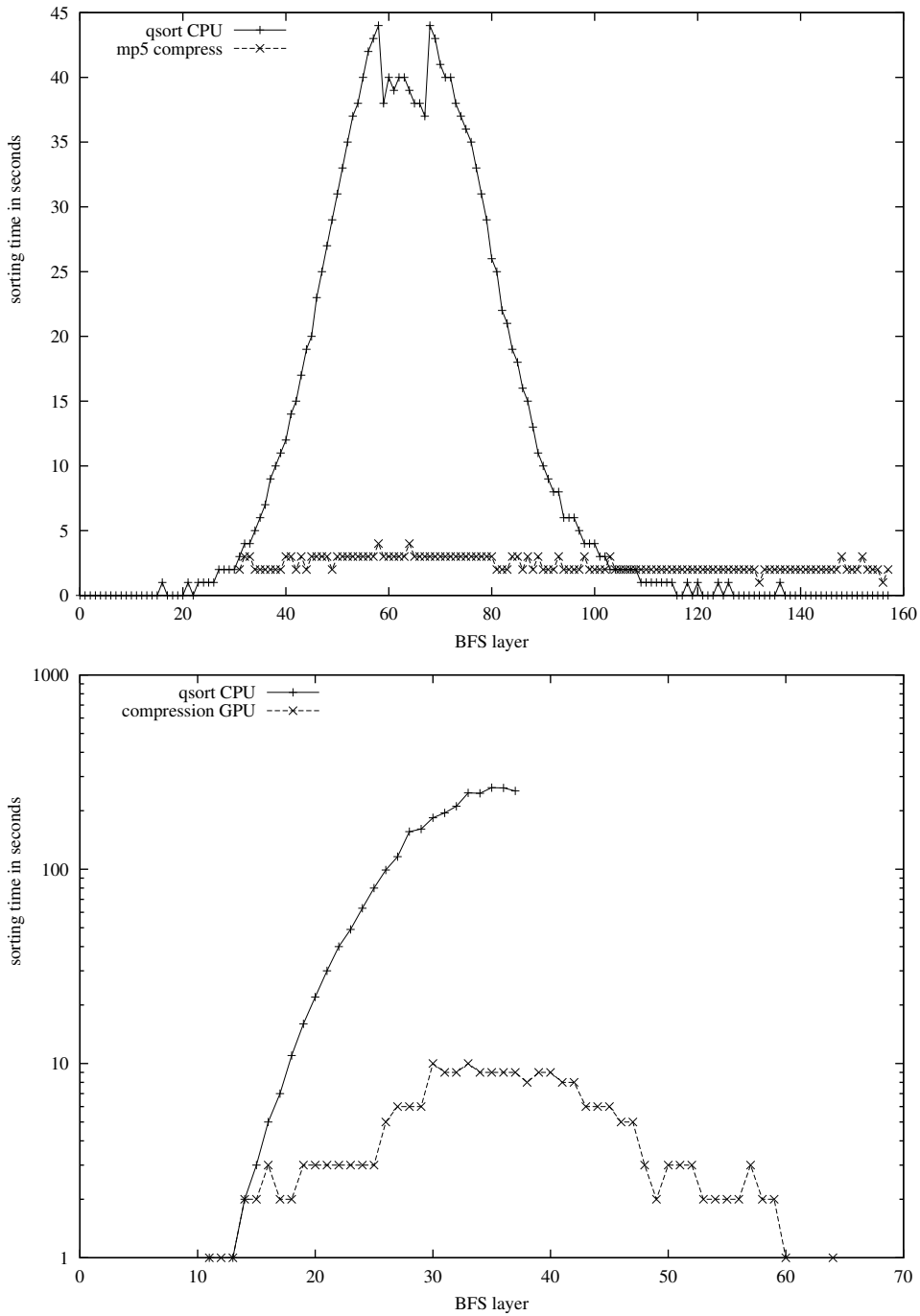


Figure 6: Comparing delayed duplicate detection via internal Quicksort.

References

- [1] J. Barnat. *Distributed Memory LTL Model Checking*. PhD thesis, Faculty of Informatics, Masaryk University Brno, 2004.
- [2] J. Barnat, L. Brim, S. Edelkamp, P. Šimeček, and D. Sulewski. Can Flash Memory Help In Model Checking? In *FMICS*, 2008. To appear.
- [3] J. Barnat, L. Brim, and P. Šimeček. I/O efficient accepting cycle detection. In *Computer-Aided Verification (CAV)*, volume 4590 of *LNCS*, pages 281–293. Springer, 2007.
- [4] J. Barnat, L. Brim, P. Šimeček, and M. Weber. Revisiting resistance speeds up I/O-efficient LTL model checking. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 48–62, 2008.
- [5] K. E. Batchier. Sorting networks and their applications. *AFIPS Spring Joint Computing Conference*, 32:307–314, 1968.
- [6] D. Bloom. A birthday problem. *American Mathematical Monthly*, 80:1141–1142, 1973.
- [7] F. C. Botelho and N. Ziviani. External perfect hashing for very large key sets. In *ACM Conference on Information and Knowledge Management (CIKM)*, pages 653–662, 2007.
- [8] D. Cantone and G. Cinotti. QuickHeapsort, an efficient mix of classical sorting algorithms. *Theoretical Computer Science*, 285(1):25–42, 2002.
- [9] D. Cederman and P. Tsigas. A practical quicksort algorithm for graphics processors. Technical Report 2008-01, Chalmers University of Technology, 2008.
- [10] A. Czumaj. *Parallel Algorithmic Techniques: PRAM Algorithms and PRAM Simulations*. Dissertation, Universität Paderborn, Heinz Nixdorf Institut, Theoretische Informatik, 1995.
- [11] S. Edelkamp and S. Jabbar. Large-scale directed model checking LTL. In *SPIN*, pages 1–18, 2006.

- [12] S. Edelkamp, S. Jabbar, and S. Schroedl. External A*. In *German Conference on Artificial Intelligence (KI)*, number 3238 in LNCS, pages 226–240. Springer, 2004.
- [13] S. Edelkamp, P. Sanders, and P. Šimeček. Semi-external LTL model checking. In *Computer-Aided Verification (CAV)*, 2008. To appear.
- [14] S. Edelkamp and D. Sulewski. Flash-efficient LTL model checking with minimal counterexamples. In *Software Engineering and Formal Methods (SEFM)*, 2008. To Appear.
- [15] S. Evangelista. Dynamic delayed duplicate detection for external memory model checking. In *SPIN*, 2008. To appear.
- [16] N. K. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUteraSort: High performance graphics coprocessor sorting for large database management. In *International Conference on Management of Data (SIGMOD)*, pages 325–336, 2006.
- [17] M. Harris, S. Sengupta, and J. D. Owens. Parallel prefix sum (scan) with CUDA. In H. Nguyen, editor, *GPU Gems 3*, pages 851–876. Addison-Wesley.
- [18] G. Holzmann and D. Bosnacki. The design of a multicore extension of the SPIN model checker. *IEEE Trans. Software Eng.*, 33(10):659–674, 2007.
- [19] G. J. Holzmann. An analysis of bitstate hashing. *Formal Methods in System Design*, 13(3):287–305, 1998.
- [20] S. Jabbar. *External-Memory algorithms for state space exploration in model checking and action planning*. PhD thesis, Faculty of Computer Science, Dortmund University of Technology, 2004.
- [21] G. Jayachandran, V. Vishal, and V. S. Pande. Using massively parallel simulations and Markovian models to study protein folding: Examining the Villin head-piece. *Journal of Chemical Physics*, 124(6):164 903–164 914, 2006.
- [22] Donald E. Knuth. *The Art of Computer Programming*. Addison-Wesley, 1973.

- [23] R. Korf. Best-first frontier search with delayed duplicate detection. In *AAAI'04*, pages 650–657. AAAI Press / The MIT Press, 2004.
- [24] R. Korf. Minimizing disk I/O in two-bit breath-first search. In *AAAI*, 2008. To appear.
- [25] R. Korf and A. Felner. Recent progress in heuristic search: A case study of the Four-Peg Towers of Hanoi problem. In *IJCAI*, pages 2334–2329, 2007.
- [26] R. Korf and P. Schultze. Large-scale parallel breadth-first search. In *AAAI*, pages 1380–1385, 2005.
- [27] P. Lamborn and E. Hansen. Layered duplicate detection in external-memory model checking. In *SPIN*, 2008. To appear.
- [28] K. Munagala and A. Ranade. I/O-complexity of graph algorithms. In *SODA*, pages 687–694, 1999.
- [29] W. Myvold and F. Ruskey. Ranking and unranking permutations in linear time. *Information Processing Letters*, 79:281–284, 2001.
- [30] R. Pagh and F. F. Rodler. Cuckoo hashing. In *European Symposium on Algorithms (ESA)*, pages 121–133, 2001.
- [31] R. Pelánek. BEEM: Benchmarks for Explicit Model Checkers. In *SPIN*, volume 4595 of *LNCIS*, pages 263–267. Springer, 2007.
- [32] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kale, and K. Schulten. Scalable molecular dynamics with NAMD. *Journal of Computational Chemistry*, 26:1781–1802, 2005.
- [33] V. Schuppan and A. Biere. 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.
- [34] U. Stern and D. L. Dill. Combining state space caching and hash compaction. In *Methoden des Entwurfs und der Verifikation digitaler Systeme, 4. GI/ITG/GME Workshop*, pages 81–90. Shaker Verlag, Aachen, 1996.

- [35] R. Zhou and E. A. Hansen. Structured duplicate detection in external-memory graph search. In *AAAI*, pages 683–689, 2004.