I/O Efficient Accepting Cycle Detection*

Jiri Barnat, Lubos Brim, and Pavel Šimeček

Department of Computer Science, Faculty of Informatics
Masaryk University Brno, Czech Republic

Abstract. We show how to adapt an existing non-DFS-based accepting cycle detection algorithm OWCTY [10,15,29] to the I/O efficient setting and compare its I/O efficiency and practical performance to the existing I/O efficient LTL model checking approach of Edelkamp and Jabbar [14]. The new algorithm exhibits similar I/O complexity with respect to the size of the graph while it avoids quadratic increase in the size of the graph. Therefore, the number of I/O operations performed is significantly lower and the algorithm exhibits better practical performance.

1 Introduction

Model checking became one of the standard technique for verification of hardware and software systems even though the class of systems that can be fully verified is fairly limited due to the well known *state explosion problem* [12]. The automata-theoretic approach [33] to model checking finite-state systems against linear-time temporal logic (LTL) reduces to the detection of reachable accepting cycles in a directed graph. Due to the state explosion problem, the graph tends to be extremely large and its size poses real limitations to the verification process. Many more-or-less successful techniques have been introduced [12] to reduce the size of the graph advancing thus the frontier of still tractable systems. Nevertheless, for real-life industrial systems these techniques are not efficient enough to fit the data into the main memory. An alternative solution is to increase the computational resources available to the verification process. The two major approaches include the usage of clusters of workstations and the usage of external memory devices (disks).

Regarding external memory devices, the goal is to develop algorithms that minimize the number of I/O operations an algorithm has to perform to complete its task. This is because the access to information stored on an external device is orders of magnitude slower than the access to information stored in the main memory. Thus the complexity of I/O efficient algorithms is measured in the number of I/O operations [1].

A lot of effort has been put into research on I/O efficient algorithms working on explicitly stored graphs [11,20,24,25]. For an explicitly stored graph, an I/O efficient algorithm typically has to perform a random access operation every

* This work has been partially supported by the Grant Agency of Czech Republic grant No. 201/06/1338 and the Academy of Sciences grant No. 1ET408050503.

W. Damm and H. Hermanns (Eds.): CAV 2007, LNCS 4590, pp. 281–293, 2007.

time it needs to enumerate edges incident with a given vertex. However, in model checking, the graphs are often given implicitly which means that the edges incident with a given vertex are computed on demand from the vertex itself. Thus, an algorithm working on an implicitly given graph may save up to $|V|$ random access operations, which may have significant impact on the performance of the algorithm in practice.

A distinguished technique that allows for an I/O efficient implementation of a graph traversal procedures is the so called *delayed duplicate detection* [21,22,26,32]. A traversal procedure has to maintain a set of visited vertices to prevent their re-exploration. Since the graphs are large, the set cannot be completely kept in the main memory and must be stored on the external memory device. When a new vertex is generated it is checked against the set to avoid its re-exploration. The idea of the delayed duplicate detection technique is to postpone the individual checks and perform them together in a group for the price of a single scan operation.

Unfortunately, the delayed duplicate detection technique is incompatible with the depth-first search (DFS) of a graph [14]. Therefore, most approaches to I/O efficient (LTL) model checking suggested so far, have focused on the state space generation and verification of safety properties only. The first I/O efficient algorithm for state space generation has been implemented in Murφ [32]. Later on, several heuristics for the state space generation were suggested and implemented in various verification tools [16,18,23]. The first attempt to verify more than safety properties was described in [19], however, the suggested approach uses the random search to find a counterexample to a given property. Therefore, it is incomplete in the sense that it is not able to prove validity of the property.

To the best of our knowledge, the only *complete* I/O efficient LTL model checker was suggested by Edelkamp and Jabbar in [14] where the problematic DFS-based algorithm was avoided by the reduction of the accepting cycle detection problem to the reachability problem [7,31] whose I/O efficient solution was further improved by using the directed (A^*) search and parallelism. The algorithm works in the on-the-fly manner meaning that only a part of the state space is constructed, which is needed in order to check the desired property. The reduction transforms the graph so that the size of the graph after the transformation is asymptotically quadratic with respect to the original one. More precisely, the size of the resulting graph is $|F| \times |G|$, where $|G|$ is the size of the original graph and $|F|$ is the number of accepting vertices. As the external memory algorithms are meant to be applied to large scale graphs, the quadratic increase in the size of the graph is significant and, according to our experience, it often aborts due to the lack of space. This is especially the case when the model is valid and the entire graph has to be traversed to prove the absence of an accepting cycle. The approach is thus mainly useful for finding counterexamples in the case a standard verification tool fails due to the lack of memory. However, completeness is a very important aspect of LTL model checking as well. A typical scenario is that if

the system is invalid and the counterexample found, the system is corrected and the property verified again. In the end, the graph must be traversed completely anyway.

Since DFS-based algorithms cannot be used for I/O efficient solution to the accepting cycle detection, a non-DFS algorithm is required. The situation very much resembles a similar one encountered in cluster-based approach to LTL model checking [2]. The main problem of the approach is that the optimal sequential algorithm (e.g. Nested DFS [17]) is inherently sequential and hence difficult to be parallelized [30]. Consequently, several new parallel algorithms that do not build on top of the depth-first search have been introduced [3,4,8,9,10].

In this paper we show how to adapt a parallel enumerative version of the *One Way Catch them Young Algorithm* (OWCTY) [10,15,29] to the I/O efficient setting and compare its I/O efficiency and practical performance with the I/O efficient LTL model checking algorithm by Edelkamp and Jabbar [14].

2 I/O Efficient OWCTY Algorithm

As discussed above, an I/O efficient solution to LTL model checking has to build upon a non-DFS algorithm. A particularly suitable algorithm for enumerative LTL model checking was described in [10]. The goal of the algorithm is to compute the set of vertices that are reachable from a vertex on an accepting cycle. If the set is empty, there is no accepting cycle in the graph, otherwise the presence of an accepting cycle is ensured [15,29].

The algorithm repeatedly computes approximations of the target set until a fixpoint is reached. All reachable vertices are inserted into the approximation set (*ApproxSet*) within the procedure Initialize-ApproxSet. After that, vertices violating the condition are gradually removed from the approximation set using procedures Elim-No-Accepting and Elim-No-Predecessors. Procedure Elim-No-Accepting removes those vertices from the approximation set that have no accepting ancestors in the set, i.e. vertices that lie on leading non-accepting cycles. Procedure Elim-No-Predecessors removes vertices that have no ancestors at all, i.e. leading vertices lying outside a cycle. The pseudo-code is given as Algorithm 1.

Algorithm 1. DetectAcceptingCycle

Require: Implicit definition of G=(V,E,ACC)
1: Initialize-ApproxSet()
2: $oldSize \leftarrow \infty$
3: **while** $(ApproxSet.size \neq oldSize) \wedge (ApproxSet.size > 0)$ **do**
4: $\quad oldSize \leftarrow ApproxSet.size$
5: $\quad$ Elim-No-Accepting()
6: $\quad$ Elim-No-Predecessors()
7: **return** $ApproxSet.size > 0$

The approximation set induces an approximation graph. The in-degree of a vertex in the approximation graph corresponds to the number of its immediate predecessors in the approximation set. To identify vertices without ancestors in the approximation set, the in-degree is maintained for every vertex of the approximation graph. Procedure Elim-No-Predecessors then works as follows. All vertices from the set with a zero in-degree are moved to a queue from where they are dequeued one by one. Dequeued vertices are eliminated from the set, and the in-degrees of its descendants are updated. If an in-degree drops to zero, the corresponding vertex is inserted into the queue to be eliminated as well. The procedure eliminates vertices in a topological order and hence the queue becomes empty as soon as all vertices preceding a cycle are eliminated.

Procedure Elim-No-Accepting works as follows. If a vertex has an accepting ancestor in the approximation set, it has to be reachable from some accepting vertex in the set. Therefore, the procedure first removes all non-accepting vertices from the set and sets the numbers of predecessors of all vertices remaining in the set to zero. Then a forward search is performed starting from the vertices remaining in the set. During the search all visited vertices are re-inserted to the approximation set and the numbers of immediate predecessors of vertices in the set are re-counted.

There are three major data structures used by the algorithm. These are *Candidates*, *ApproxSet*, and *Open*. *Candidates* is the set of vertices strictly kept in memory that is used for the delayed duplicate detection technique. It keeps vertices that have been processed and are waiting to be checked against the set of vertices stored on the external device. *ApproxSet* is the set of vertices belonging to the current approximation set. It is implemented as a linear list and stored externally. Together with *Candidates*, it is used as the set of vertices already visited during the forward exploration of the graph in procedure Elim-No-Accepting. For that purpose, both *Candidates* and *ApproxSet* data structures are modified to keep not only vertices, but also the corresponding numbers of relevant immediate predecessors. The number associated with a particular vertex s is referred to as the *appendix* of the vertex and is set and read with methods *setAppendix(s)* and *getAppendix(s)*, respectively. Finally, the data structure *Open* is a queue of vertices. It is used to keep open vertices during the breadth-first exploration of the graph within procedure Elim-No-Accepting, and vertices to be eliminated (vertices without any predecessors) during the execution of procedure Elim-No-Predecessors. The data structure *Open* is stored in the external memory, the vertices are, however, inserted into and taken from it in a strict FIFO manner. Thus, a possible I/O overhead could be minimized using an appropriate buffering mechanism.

In some of its phases, the algorithm performs a *scan* through the externally stored set of vertices (*ApproxSet*) and decides about every vertex if it should be removed from the set or not. To preserve the I/O efficiency of such an operation, a temporary external data structure *ApproxSet'* is introduced. In particular, vertices that should remain in the set are copied to the temporary structure.

Algorithm 2. MERGE

```
 1: if mode = Elim-No-Accepting then
 2:     for all s ∈ ApproxSet do
 3:         if s ∈ Candidates then
 4:             app ← Candidates.getAppendix(s)
 5:             app' ← ApproxSet.getAppendix(s)
 6:             Candidates ← Candidates \ {s}
 7:             ApproxSet.setAppendix(s, app + app')
 8:     for all s ∈ Candidates do
 9:         Open.pushBack(s)
10:         ApproxSet ← ApproxSet ∪ {s}
11: else
12:     ApproxSet' ← ∅
13:     for all s ∈ ApproxSet do
14:         app' ← ApproxSet.getAppendix(s)
15:         if s ∈ Candidates then
16:             app ← Candidates.getAppendix(s)
17:             if (app + app') = 0 then
18:                 Open.pushBack(s)
19:             else
20:                 ApproxSet' ← ApproxSet' ∪ {s}
21:                 ApproxSet'.setAppendix(s, app + app')
22:         else
23:             ApproxSet' ← ApproxSet' ∪ {s}
24:             ApproxSet'.setAppendix(s, app')
25:     ApproxSet ← ApproxSet'
26: Candidates ← ∅
```

Once the scan is complete, the content of the original *ApproxSet* is discarded and replaced with the content of the temporary structure *ApproxSet'*.

Having described the data structures we are ready to introduce several auxiliary subroutines. The most important one is procedure MERGE that is responsible for merging information about vertices stored in the internal memory (*Candidates*) and vertices stored externally (*ApproxSet*). The procedure can operate in two different modes according to the value of the variable *mode*. The two modes correspond to the top most procedures ELIM-NO-ACCEPTING and ELIM-NO-PREDECESSORS. In the mode **Elim-No-Accepting**, vertices from set *Candidates* are merged with vertices from *ApproxSet* and the result is stored externally to *ApproxSet*. For already visited vertices the corresponding appendices are just combined and stored externally. Moreover, newly discovered vertices are inserted into the queue of vertices to be further processed (*Queue*). In the mode **Elim-No-Predecessors**, no new vertices are discovered, hence only the appendices are combined. Vertices with zero in-degree are removed from the external memory and in-degree of their immediate descendants is appropriately decreased. For the details see Algorithm 2.

Algorithm 3. StoreOrCombine

```
Require: s, app
 1: if s ∈ Candidates then
 2:       app' ← Candidates.getAppendix(s)
 3:       Candidates.setAppendix(s, app+app')
 4: else
 5:       Candidates ← Candidates ∪ {s}
 6:       Candidates.setAppendix(s, app)
 7:       if MemoryIsFull() then
 8:            Merge()
```

Another auxiliary procedure is procedure StoreOrCombine whose purpose is to insert a vertex into the candidate set if the vertex is not yet present in the set, or update the corresponding appendix of the vertex, otherwise. Once the main memory becomes full, vertices from the candidate set are processed and the candidate set is emptied by procedure Merge.

Algorithm 4. OpenIsNotEmpty

```
1: if Open.isEmpty() then
2:       Merge()
3: return ¬Open.isEmpty()
```

The last auxiliary function is a function for checking the emptiness of the queue of vertices to be processed (*Open*). If the queue is empty, procedure OpenIsNotEmpty calls procedure Merge to perform the delayed duplicate detection. The procedure returns **False**, if *Open* is empty and merging has not brought any new vertices to be processed.

Algorithm 5 and Algorithm 6 give pseudo-codes of the two main procedures. Note that algorithm DetectAcceptingCycle uses functions GetInitialVertex, GetSuccessors, and IsAccepting to traverse the graph and to check whether a vertex is accepting or not. These functions are part of the implicit definition of the graph. Procedure Elim-No-Accepting has actually two goals. First, to eliminate those vertices from the approximation set that are unreachable from accepting vertices in the set, and second, to properly count the in-degrees in the approximation graph. Procedure Elim-No-Predecessors employs the in-degrees to recursively remove vertices without predecessors from the approximation set.

An important observation is that it is not necessary to initialize the approximation set with the set of all vertices. Since the first procedure in the very first iteration of the while loop performs forward exploration of the graph starting from accepting vertices in the set, it is enough to initialize the set with "leading" accepting vertices only, i.e. those accepting vertices that have no accepting ancestors. Such vertices can be identified with a simple forward traversal that is

Algorithm 5. ELIM-NO-ACCEPTING

```
 1: mode ← Elim-No-Accepting
 2: ApproxSet' ← ∅
 3: for all s ∈ ApproxSet do
 4:       if IsAccepting(s) then
 5:             Open.pushBack(s)
 6:             ApproxSet' ← ApproxSet' ∪ {s}
 7:             ApproxSet'.setAppendix(s, 0)
 8: ApproxSet ← ApproxSet'
 9: while OpenIsNotEmpty() do
10:       s ← Open.popFront()
11:       for all t ∈ GetSuccessors(s) do
12:             StoreOrCombine(t, 1)
```

Algorithm 6. ELIM-NO-PREDECESSORS

```
 1: mode ← Elim-No-Predecessors
 2: ApproxSet' ← ∅
 3: for all s ∈ ApproxSet do
 4:       if ApproxSet.getAppendix(s) = 0 then
 5:             Open.pushBack(s)
 6:       else
 7:             ApproxSet' ← ApproxSet' ∪ {s}
 8: ApproxSet ← ApproxSet'
 9: while OpenIsNotEmpty() do
10:       s ← Open.popFront()
11:       for all t ∈ GetSuccessors(s) do
12:             StoreOrCombine(t, −1)
```

Algorithm 7. INITIALIZE-APPROXSET

```
 1: mode ← Elim-No-Accepting
 2: Candidates ← ∅
 3: s ← GetInitialVertex()
 4: ApproxSet ← {s}
 5: if ¬ IsAccepting(s) then
 6:       Open.pushBack(s)
 7:       while OpenIsNotEmpty() do
 8:             s ← Open.popFront()
 9:             for all t ∈ GetSuccessors(s) do
10:                   if IsAccepting(t) then
11:                         ApproxSet ← ApproxSet ∪ {t}
12:                   else
13:                         StoreOrCombine(t, 0)
```

allowed to explore descendants of non-accepting vertices only. See the pseudo-code given as Algorithm 7.

3 Complexity Analysis

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 I/O algorithm is measured in terms of the numbers of external I/O operations only. This is motivated by the fact that a single I/O operation is by approximately six orders of magnitude slower than a computation step performed in the main memory [34]. 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 an I/O algorithm in the model of Aggarwal and Vitter is further parametrized by M, B, and D, where M denotes the number of items that fits into the internal memory, B denotes the number of items that can be transferred in a single I/O operation, and D denotes the number of blocks that can be transferred in parallel, i.e. the number of independent parallel disks available. The abbreviations $sort(n)$ and $scan(n)$ stand for $\theta(N/(DB)log_{M/B}(N/B))$ and $\theta(N/(DB))$, respectively. In this section we give the I/O complexity of our algorithm and compare it with the complexity of the algorithm from [14].

We use the following notation. *BFS tree* is a tree given by the graph traversal from the initial set of vertices in the breadth-first order. Its height h_{BFS} is called *BFS height*, its levels are called *BFS levels*. *SCC graph* is a directed acyclic graph, whose vertices are maximal strongly connected components of the graph and the edges are given according to the reachability relation between the components. Let l_{SCC} denote the length of the longest path in the SCC graph. The I/O complexity of the algorithm is given in Theorem 1. The proof of the complexity can be found in the full version of the paper [6].

Theorem 1. *The I/O complexity of algorithm* DetectAcceptingCycle *is*

$$\mathcal{O}(l_{SCC} \cdot (h_{BFS} + |p_{max}| + |E|/M) \cdot scan(|V|)),$$

where p_{max} is the longest path in the graph going through trivial strongly connected components (without self-loops).

For the purpose of comparison we denote our new algorithm as *DAC* and the algorithm of Edelkamp and Jabbar [14] as *EJ*. Theorem 1 of [14] claims that *EJ* is able to detect accepting cycles with I/O complexity $\mathcal{O}(sort(|F||E|) + l \cdot scan(|F||V|))$, where $|F|$ is the number of accepting states and l is the length of the shortest counterexample.

The complexity of *EJ* is not easy to compare with our results, because the two algorithms use different ways to maintain the set of candidates. The candidate set can be either stored externally (*EJ*) or internally (*DAC*). In the case that the candidate set is stored externally, it is possible to perform the merge operation on a BFS level independently of the size of the main memory. Therefore, this

approach is suitable for those cases where memory is small or the graph is by orders of magnitude larger. The disadvantage of the approach is that it needs *sort* operations and it cannot be combined with heuristics, such as bit-state hashing and a lossy hash table [16]. Fortunately, both *EJ* and *DAC* are modular enough to be able to work in both modes. Table 1 gives I/O complexities of all four variants, where *EJ'* denotes algorithm *EJ* modified so that the candidate set is kept in the internal memory, and *DAC'* denotes algorithm *DAC* modified so that the candidate set is stored externally.

Table 1. I/O complexity of algorithms for both modes of storage of the candidate set

Candidate set in the main memory:

EJ'	$\mathcal{O}((l + \|F\|\|E\|/M) \cdot scan(\|F\|\|V\|))$
DAC	$\mathcal{O}(l_{SCC} \cdot (h_{BFS} + \|p_{max}\| + \|E\|/M) \cdot scan(\|V\|))$

Candidate set in the external memory:

EJ	$\mathcal{O}(l \cdot scan(\|F\|\|V\|) + sort(\|F\|\|E\|))$
DAC'	$\mathcal{O}(l_{SCC} \cdot ((h_{BFS} + \|p_{max}\|) \cdot scan(\|V\|) + sort(\|E\|)))$

In the worst case the values of l_{SCC}, $|p_{max}|$, and h_{BFS} are equal to $|V|$. Thus the worst case I/O complexity of *DAC* is better than that of *EJ'* and the worst case I/O complexity of *DAC'* is equal to that of *EJ*, provided that $l = |V|$ and $|F| = |V|$.

Note that for graphs of verified systems the numbers l_{SCC}, $|p_{max}|$, and h_{BFS} are typically smaller by several orders of magnitude than the number of vertices. l_{SCC} (giving the upper bound to the number of iterations of the loop of Algorithm 1) usually ranges from 1 to 20 [15]. h_{BFS} is not proportional to the size of the state space and oscillates around several hundreds [27], so the $|p_{max}|$ according to our own measurements. However, the number of accepting vertices (F) is quite often in the same order of magnitude as the number of vertices. Therefore, *EJ'* and *EJ* suffer from the graph blow-up and perform much more I/O operations compared to *DAC* and *DAC'*, respectively. On the other hand, *EJ'* and *EJ* work on-the-fly and can thus outperform *DAC* and *DAC'* on the graphs with small number of accepting vertices and short counterexamples. Nevertheless, short counterexamples are also easy to find using on-the-fly internal memory model checkers which outperform both external memory approaches.

Regarding space complexity, *DAC* is more space efficient than *EJ*. Since *EJ'* needs to remember all visited pairs of vertices, where a pair consists of one accepting and one arbitrary vertex, the space complexity of the algorithm is $\mathcal{O}(|F||V|)$, i.e. asymptotically quadratic in the size of the graph. On the other hand, the space complexity of *DAC* is $\mathcal{O}(|V|)$, as it only maintains the approximation set, queue and the candidate set whose sizes are always bounded by the number of vertices. The same holds for the pair *EJ* and *DAC'*.

4 Experimental Evaluation

In order to obtain experimental evidence about how our algorithm behaves in practice, we implemented both algorithms and compared them mutually as well as with the model checker SPIN with all the default reduction techniques (including partial order) turned on.

Algorithm *DetectAcceptingCycle* (*DAC*) has been implemented upon DiVinE Library [5], providing the state space generator, and STXXL Library [13], providing the necessary I/O primitives. Algorithm *EJ* was implemented as a procedure that performs the graph transformation as suggested in [14] and then employs I/O efficient breadth-first search to check for the counterexample. Note that our implementation of [14] does not have the A^* heuristics and so it can be less efficient in the search for the counterexample. The procedure is referred to as *Liveness as Safety with BFS* (*LaS-BFS*).

We have measured run times and a memory consumption of SPIN, *LaS-BFS* and *DAC* on a collection of systems and their LTL properties taken from the BEEM project [28]. The models were selected so that the state spaces generated by SPIN and DiVinE were exactly of the same size. The experimental results are listed in Table 2. Note that just before the unsuccessful termination of *LaS-BFS* due to exhausting the disk space the size of BFS levels exhibited growing

Table 2. Run times and memory consumption on a single workstation with 2 GB of RAM and 60 GB of available hard disk space. The time is given in `hh:mm:ss` format.

		SPIN		LaS-BFS		DAC	
	States	Time	RAM	Time	Disk	Time	Disk
Phils(16,1),P3	61,230,206	Out of memory		Out of disk space		02:01:11	5.5 GB
MCS(5),P4	119,663,657	Out of memory		Out of disk space		03:32:41	8 GB
Szymanski(5),P4	419,183,762	Out of memory		Out of disk space		44:49:36	32 GB
Elevator2(16),P4	76,824,540	Out of memory		Out of disk space		11:37:57	9.2 GB
Leader Fil.(7),P2	431,401,020	00:01:35	1369 MB	Out of disk space		32:03:52	42 GB

Valid properties on large models.

		SPIN		LaS-BFS		DAC	
	States	Time	RAM	Time	Disk	Time	Disk
Lamport(3),P4	56,377	00:00:01	18 MB	00:55:34	799 MB	00:00:19	6,1 MB
Anderson(4),P2	58,205	00:00:01	20 MB	00:11:11	153 MB	00:00:18	6,1 MB
Peterson(4),P4	2,239,039	00:00:08	85 MB	Out of disk space		00:04:44	159 MB

Valid properties on small models.

		SPIN		LaS-BFS		DAC	
	States	Time	RAM	Time	Disk	Time	Disk
Bakery(5,5),P3	506,246,410	00:00:01	16 MB	01:34:13	5,4 GB	69:27:58	38 GB
Szymanski(4),P2	4,555,287	00:00:01	18 MB	00:59:00	203 MB	00:19:55	205 MB
Elevator2(7),P5	43,776	00:00:01	17 MB	00:01:15	121 MB	00:00:18	6,1 MB

Invalid properties.

tendency. This suggests that the computation would last substantially longer if sufficient disk space was available. For the same input graphs, algorithm *DAC* manage to perform the verification using a few GBs of space only.

Measurements on large systems with valid formulas demonstrate that *DAC* is able to successfully prove the correctness of systems, on which SPIN and *LaS-BFS* fail. However, there are systems and valid formulas, which take a long time to verify by our algorithm, but can be verified quickly using SPIN (e.g. model *Leader Filters*). This is due to the partial order reduction technique, which is extraordinarily efficient in this case. Results on small systems show the state space blow-up in case of *LaS-BFS*. E.g. on the model *Lamport*, 6,1 MB of disk space is enough for *DAC* to store the entire state space while *LaS-BFS* needs 799 MB. As for systems with invalid formulas, the new algorithm is slow, since it does not work on-the-fly. Nevertheless, it is able to finish if the state space fits in the external memory. Moreover, it is faster than *LaS-BFS* on systems with long counterexamples as the space space blow-up takes effect when *LaS-BFS* has to traverse a substantial part of the state space (e.g. model *Elevator2*).

In summary, the new algorithm is especially useful for verification of large systems with valid formulas where SPIN fails due to the limited size of the main memory and *LaS-BFS* runs out of the available external memory because of a large amount of accepting states. On systems with invalid formulas, algorithm *DAC* finishes if the state space fits in the external memory, but it may take quite a long time as it does not work on-the-fly.

5 Conclusions and Future Work

In this paper we presented a new I/O efficient algorithm for accepting cycle detection on implicitly given graphs. The algorithm exhibits linear space complexity while preserving practically reasonable I/O complexity. Another indirect contribution of the paper is that it introduces an I/O efficient procedure to compute the topological sort on implicitly given graphs (procedure Elim-No-Predecessors).

Our experimental evaluation confirmed that the new algorithm is able to fully solve instances of the LTL model checking problem that cannot be solved either with the standard LTL model checker SPIN or using so far the best I/O efficient approach of Edelkamp and Jabbar [14]. The approach of [14] fails especially if the verified formula is valid, which is because after the transformation, the graph becomes too large to be kept even in the external memory.

On the other hand, unlike SPIN and the approach of [14] our algorithm does not work on-the-fly. The on-the-fly algorithms are particularly successful if the property is violated and the counterexample can be found early during the state space exploration.

As our algorithm is based on the algorithm which can be easily parallelized [10], it is straightforward to develop a parallel version of the algorithm that can further speed up verification of large systems. It also seems promising to design I/O efficient variants of other BFS-based verification algorithms [3,4,8,9,10].

Some of them work on-the-fly and hence could outperform both the new algorithm and the algorithm of Edelkamp and Jabbar.

An open problem for which we still do not know a practically good solution, is the inefficiency of the delayed duplicate detection technique as used in procedure Elim-No-Predecessors. Since procedure Merge is called every time a BFS level is explored, merging a small level into a large set can slow down the exploration speed of a few vertices per minute. The question is, if this can be avoided.

References

1. Aggarwal, A., Vitter, J.S.: The Input/Output Complexity of Sorting and Related Problems. Communications of the ACM 31(9), 1116–1127 (1988)
2. Barnat, J.: Distributed Memory LTL Model Checking. PhD thesis, Faculty of Informatics, Masaryk University Brno (2004)
3. Barnat, J., Brim, L., Chaloupka, J.: Parallel Breadth-First Search LTL Model-Checking. In: Automated Software Engineering (ASE'03), pp. 106–115. IEEE Computer Society Press, Los Alamitos (2003)
4. Barnat, J., Brim, L., Stříbrná, J.: Distributed LTL Model-Checking in SPIN. In: Dwyer, M.B. (ed.) Model Checking Software. LNCS, vol. 2057, pp. 200–216. Springer, Heidelberg (2001)
5. Barnat, J., Brim, L., Černá, I., Šimeček, P.: DiVinE – The Distributed Verification Environment. In: PDMC'05, pp. 89–94 (2005)
6. Barnat, J., Brim, L., Šimeček, P.: LTL Model Checking with I/O-Efficient Accepting Cycle Detection. Technical Report FIMU-RS-2007-01, Faculty of Informatics, Masaryk University (2007)
7. Biere, A., Artho, C., Schuppan, V.: Liveness Checking as Safety Checking. Electr. Notes Theor. Comput. Sci. 66(2) (2002)
8. Brim, L., Černá, I., Moravec, P., Šimša, J.: Accepting Predecessors are Better than Back Edges in Distributed LTL Model-Checking. In: Hu, A.J., Martin, A.K. (eds.) FMCAD 2004. LNCS, vol. 3312, pp. 352–366. Springer, Heidelberg (2004)
9. Brim, L., Černá, I., Krčál, P., Pelánek, R.: Distributed LTL Model Checking Based on Negative Cycle Detection. In: Hariharan, R., Mukund, M., Vinay, V. (eds.) FST TCS 2001: Foundations of Software Technology and Theoretical Computer Science. LNCS, vol. 2245, pp. 96–107. Springer, Heidelberg (2001)
10. Černá, I., Pelánek, R.: Distributed Explicit Fair Cycle Detection. In: Ball, T., Rajamani, S.K. (eds.) SPIN'03. LNCS, vol. 2648, pp. 49–73. Springer, Heidelberg (2003)
11. Chiang, Y., Goodrich, M., Grove, E., Tamassia, R., Vengroff, D., Vitter, J.: External-Memory Graph Algorithms. In: SODA'95, pp. 139–149. Society for Industrial and Applied Mathematics (1995)
12. Clarke Jr, E., Grumberg, O., Peled, D.: Model Checking. MIT press, Cambridge (1999)
13. 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)
14. Edelkamp, S., Jabbar, S.: Large-Scale Directed Model Checking LTL. In: SPIN'06, pp. 1–18 (2006)

15. Fisler, K., Fraer, R., Kamhi, G., Vardi, M.Y., Yang, Z.: Is There a Best Symbolic Cycle-Detection Algorithm? In: Margaria, T., Yi, W. (eds.) ETAPS 2001 and TACAS 2001. LNCS, vol. 2031, pp. 420–434. Springer, Heidelberg (2001)
16. Hammer, M., Weber, M.: To Store Or Not To Store Reloaded: Reclaiming Memory On Demand. In: FMICS 2006 and PDMC 2006. LNCS, vol. 4346, pp. 51–66. Springer, Heidelberg (2006)
17. Holzmann, G.J., Peled, D., Yannakakis, M.: On Nested Depth First Search. In: The SPIN Verification System, pp. 23–32. American Mathematical Society (1996)
18. Jabbar, S., Edelkamp, S.: I/O Efficient Directed Model Checking. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 313–329. Springer, Heidelberg (2005)
19. Jones, M., Mercer, E.: Explicit State Model Checking with Hopper. In: Graf, S., Mounier, L. (eds.) SPIN'04. LNCS, vol. 2989, pp. 146–150. Springer, Heidelberg (2004)
20. Katriel, I., Meyer, U.: Elementary Graph Algorithms in External Memory. In: Algorithms for Memory Hierarchies, pp. 62–84 (2002)
21. Korf, R.: Best-First Frontier Search with Delayed Duplicate Detection. In: AAAI'04, pp. 650–657. AAAI Press / The MIT Press, Cambridge, MA (2004)
22. Korf, R., Schultze, P.: Large-Scale Parallel Breadth-First Search. In: AAAI'05, pp. 1380–1385. AAAI Press / The MIT Press, Cambridge, MA (2005)
23. Kristensen, L., Mailund, T.: Efficient Path Finding with the Sweep-Line Method Using External Storage. In: Dong, J.S., Woodcock, J. (eds.) ICFEM 2003. LNCS, vol. 2885, pp. 319–337. Springer, Heidelberg (2003)
24. Kumar, V., Schwabe, E.: Improved Algorithms and Data Structures for Solving Graph Problems in External Memory. In: 8th IEEE Symposium on Parallel and Distributed Processing (SPDP'96), IEEE Computer Society Press, Los Alamitos (1996)
25. Mehlhorn, K., Meyer, U.: External-Memory Breadth-First Search with Sublinear I/O. In: Möhring, R.H., Raman, R. (eds.) ESA 2002. LNCS, vol. 2461, pp. 723–735. Springer, Heidelberg (2002)
26. Munagala, K., Ranade, A.: I/O-Complexity of Graph Algorithms. In: SODA'99, pp. 687–694, Philadelphia, PA, USA, Society for Industrial and Applied Mathematics (1999)
27. Pelánek, R.: Typical Structural Properties of State Spaces. In: Graf, S., Mounier, L. (eds.) SPIN'04. LNCS, vol. 2989, pp. 5–22. Springer, Heidelberg (2004)
28. Pelánek, R.: BEEM: BEnchmarks for Explicit Model checkers (February 2007) `http://anna.fi.muni.cz/models/index.html`
29. Ravi, K., Bloem, R., Somenzi, F.: A Comparative Study of Symbolic Algorithms for the Computation of Fair Cycles. In: Johnson, S.D., Hunt Jr., W.A. (eds.) FMCAD 2000. LNCS, vol. 1954, pp. 143–160. Springer, Heidelberg (2000)
30. Reif, J.H.: Depth-First Search is Inherrently Sequential. Information Processing Letters 20(5), 229–234 (1985)
31. 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)
32. Stern, U., Dill, D.L.: Using Magnetic Disk Instead of Main Memory in the Murphi Verifier. In: CAV'98, pp. 172–183 (1998)
33. Vardi, M., Wolper, P.: An Automata-Theoretic Approach to Automatic Program Verification. In: Logic in Computer Science (LICS'86), pp. 332–344. IEEE Computer Society Press, Los Alamitos (1986)
34. Vitter, J.: External Memory Algorithms and Data Structures: Dealing with Massive Data. ACM Comput. Surv. 33(2), 209–271 (2001)