An I/O Efficient Model Checking Algorithm for Large-Scale Systems

Lijun Wu, Huijia Huang, Kaile Su, Shaowei Cai, and Xiaosong Zhang

Abstract—**Model checking is a powerful approach for the formal verification of hardware and software systems. However, this approach suffers from the state space explosion problem, which limits its application to large-scale systems due to space shortage. To overcome this drawback, one of the most effective solutions is to use external memory algorithms. In this paper, we propose an I/O efficient model checking algorithm for large-scale systems. To lower I/O complexity and improve time efficiency, we combine three new techniques: 1) a linear hash-sorting technique; 2) a cached duplicate detection technique; and 3) a dynamic path management technique. We show that the new algorithm has a lower I/O complexity than state-of-the-art I/O efficient model checking algorithms, including detect accepting cycle, maximal accepting predecessors, and iterative-deepening depth-first search. In addition, the experiments show that our algorithm obviously outperforms these three algorithms on the selected representative benchmarks in terms of performance.**

Index Terms—**Duplicate detection, dynamic search path management, linear hash-sorting, model checking, state space explosion.**

I. Introduction

MODEL checking is a powerful approach for the formal verification of hardware and software systems. When applicable, it automatically checks whether or not a system satisfies a given specification via detection of counterexamples. There have been a lot of efforts applying model checking in hardware verification [1]–[4]. However, this approach severely suffers from the state space explosion problem, which renders it inapplicable to large-scale systems due to space shortage [5].

Practical model checking algorithms mainly fall into two types: 1) internal memory algorithms; and 2) external algorithms. To overcome the state space explosion problem, internal memory algorithms focus on reducing system size or representation. To this end, many techniques are introduced, such as partial order reduction [6], symmetry reduction [7], abstraction [8], compositional approach [9], symbolic model checking [10], symbolic trajectory evaluation (STE), automata theory [11], and bounded model checking [12]. Nevertheless, due to the internal memory limitation, internal memory algorithms become inapplicable to real-life industrial systems with large scale.

Compared with internal memory, external memory devices (disks) can provide much larger space. In addition, in the past few years, there have been enormous increase in the capacity of magnetic disks, with little increase in their cost, resulting in dramatic reductions in the cost per byte. Magnetic disk is about two and a half orders of magnitude cheaper than the semiconductor memory [13], which suggests the idea of using external memory in model checking large-scale systems. Because external memory access is orders of magnitude slower than internal memory access [14], [15], the main concern for external memory algorithms is to reduce the number of I/O operations so as to improve their time efficiency.

In this paper, we propose an I/O efficient model checking algorithm for large-scale systems based on nested depth-first search [16], called IOEMC. To lower I/O complexity and improve time efficiency, we combine three new techniques: 1) a linear hash-sorting algorithm denoted by LHS; 2) a cached duplicate detection technique denoted by CDD; and 3) a dynamic path management technique denoted by DPM. The algorithm LHS aims to quickly locate a record in a hash table on disk by the hash value of a state. Whenever the hash table storing visited states in the internal memory is full, it merges the hash table into the sorted hash table in external memory and sorts the new hash table in external memory by a special technique. The I/O complexity of this algorithm is linear in the size of the two hash tables together. The CDD technique allows almost all duplicate detections to be performed in internal memory by efficient management of visited states. With LHS together, CDD significantly reduces the cost of duplicate detection. The scheme DPM makes two stacks of the nested depth-first search, dynamically share the same memory section, and solves the memory dithering problem by efficient management of stacks and states, where the memory dithering refers to the phenomenon that states are frequently moved into and out of the internal memory (see Section V-D), which may significantly increase the number of I/O operations and thus reduce the efficiency of an algorithm.

For demonstrating the effectiveness of IOEMC, we compare it with state-of-the-art I/O efficient linear temporal logic (LTL)

Manuscript received November 22, 2013; revised May 4, 2014; accepted June 9, 2014. Date of publication July 8, 2014; date of current version April 22, 2015. This work was supported in part by the National Natural Science Foundation of China under Grants 61370072 and 61073033 and in part by the Australian Research Council Discovery under Grant DP110101211.

L. Wu is with the School of Computer Science and Engineering, University of Electronic Science and Technology of China, Chengdu 611731, China and also with the School of Information Technology and Electrical Engineering, University of Queensland, Brisbane, Qld. 4072, Australia (e-mail: wljuestc@gmail.com).

H. Huang and X. Zhang are with the School of Computer Science and Engineering, University of Electronic Science and Technology of China, Chengdu 611731, China (e-mail: gzzsudocwlj@yahoo.com; xiaosongzhanguestc@sina.com).

K. Su is with the Institute for Integrated and Intelligent Systems, Griffith University, Brisbane, Qld. 4072, Australia (e-mail: kailesuuestc@sina.com).

S. Cai is with the State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China and the Queensland Research Laboratory National ICT Australia, Brisbane, Australia (e-mail: shaoweicaiuestc@sina.com).

Digital Object Identifier 10.1109/TVLSI.2014.2330061

model checking algorithms, including detect accepting cycle (DAC) [14], maximal accepting predecessors (MAP) [17], and iterative-deepening depth-first search (IDDFS) [18]. The complexity comparisons show that IOEMC has lower I/O complexity than DAC, MAP, and IDDFS. Furthermore, the experiments show that the time efficiency of IOEMC is obviously better than its competitors.

The rest of this paper is organized as follows. We describe a running example in Section II. Section III provides some necessary knowledge used in this paper. In Section IV, we introduce the related work. And then, we propose an I/O efficient LTL model checking algorithm for large-scale systems based on LHS, CDD, and DPM in Section V. In Sections VI and VII, we compare our algorithm's I/O complexity and practical performance with that of DAC, MAP, and IDDFS. Section VIII discusses the parameter used in the dynamic path management technique DPM and state number limit. Finally, we conclude this paper in Section IX.

II. Example

In order to illustrate the related concepts and our algorithm, we use $ITC'99, b15(std)$ as a running example, which is also a benchmark in the experiment of this paper. $ITC'99, b15(std)$ is a standard 80386 processor (subset) that has 671 VHDL (VHSIC Hardware Description Language) lines, three processes, 8922 gates, 36 primary inputs, 70 primary outputs, 449 flip-flops, 21 logic-zero, nine logic-one, and 53 018 faults in complete fault list, where VHSIC is the abbreviation of very high speed integrated circuit. Its RT (register transfer) level VHDL description can be found in [19]. Two properties to be verified for $ITC'99, b15(std)$ are as follows.

1) $P1$: $AG(\text{reset}='1' \wedge Pro0@s_3)$, meaning that whenever 80386 processor is reset, the system process $Pro0$ will forever stay at the third state s_3.
2) $P2$: $EF(\text{reset}='1' \wedge Pro0@s_3)$, meaning that there is one path such that the system process $Pro0$ will eventually research the third state s_3 along the path when 80386 processor is reset.

III. Preliminaries

In this section, we provide a brief introduction of some necessary notions used in this paper. Please refer to [5] and [20] for more details.

A. Model Checking and Automata

Model checking is a technique that checks if a system satisfies the given specification, where the specification is the property that the system needs to satisfy, which is expressed by a logical formula. For example, the verified properties of the benchmark $ITC'99, B15(std)$ are expressed by $AG(\text{reset}='1' \wedge Pro0@s_3)$ and $EF(\text{reset}='1' \wedge Pro0@s_3)$. The automata-theoretic approach is one of the most efficient model checking techniques.

Formally, a finite automaton (over finite words) $\mathcal{M}$ is a five tuple $(\sum, Q, \Delta, Q^0, F)$ such as follows.

1) $\sum$ is the finite alphabet. The letters in $\sum$ are used for transition labels.
2) Q is the finite set of states.
3) $\Delta \subseteq Q \times \sum \times Q$ is the transition relation, $(q_1, a, q_2) \in \Delta$ means that state q_1 transits to state q_2 through the edge labeled with letter a. The a is called a transition label.
4) $Q^0 \subseteq Q$ is the set of initial states.
5) F is the set of final states (or accepting states).

We use $\mathcal{L}(\mathcal{M})$ to denote the language accepted by $\mathcal{M}$. Suppose the specification (or property) that the system needs to satisfy is expressed by LTL formula φ, the negation $\neg\varphi$ of the specification is translated into automaton $\mathcal{S} = (\sum, Q_2, \Delta_2, Q_2^0, F_2)$, and the system to be verified is translated into automaton $\mathcal{A} = (\sum, Q_1, \Delta_1, Q_1^0, F_1)$. According to [5] and [14], model checking is to check whether or not there is an accepting cycle accessible from some initial state in the intersection automaton of $\mathcal{A}$ and $\mathcal{S}$ which is denoted by $\mathcal{A} \cap \mathcal{S}$, where a cycle is a path that first vertex (state) and last vertex of the path are the same, and an accepting cycle is a cycle going through some accepting vertex. If there exists such a cycle, then the path consisting of the accepting cycle and the path from some initial state to the cycle is a counterexample; otherwise, the system satisfies the given specification. Thus, transition labels can be ignored, which does not affect the verification results.

The intersection of automata of $\mathcal{A}$ and $\mathcal{S}$ is an automaton that accepts language $\mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{S})$. Note that intersection of automata here is different from that of sets. Because accepting states from both automata may appear together only finitely many times even if they appear individually infinitely often [5], setting $F = F_1 \times F_2$ does not work. Hence, we build $\mathcal{A} \cap \mathcal{S}$ following the method in [5]. Namely, $\mathcal{A} \cap \mathcal{S} = (\sum, Q_1 \times Q_2 \times \{0, 1, 2\}, \Delta, Q_1^0 \times Q_2^0 \times 0, Q_1 \times Q_2 \times 2)$, where $\sum$ is the alphabet, $Q_1 \times Q_2 \times \{0, 1, 2\}$ is the state set, Δ is the transition relation, $Q_1^0 \times Q_2^0 \times 0$ is the initial state set, $Q_1 \times Q_2 \times 2$ is the accepting state set of $\mathcal{A} \cap \mathcal{S}$, respectively. The transition relation Δ of $\mathcal{A} \cap \mathcal{S}$ is defined as following: let $(r_i, q_j, x), (r_m, q_n, y) \in Q_1 \times Q_2 \times \{0, 1, 2\}$ be two states. $((r_i, q_j, x), a, (r_m, q_n, y)) \in \Delta$ if and only if the following conditions hold:

1) $(r_i, a, r_m) \in \Delta_1$ and $(q_j, a, q_n) \in \Delta_2$, that is, the local components agree with the transitions of $\mathcal{A}$ and $\mathcal{S}$;
2) the third component is affected by the accepting conditions of $\mathcal{A}$ and $\mathcal{S}$;
3) if $x = 0$ and $r_m \in F_1$, then $y = 1$;
4) if $x = 1$ and $q_n \in F_2$, then $y = 2$;
5) if $x = 2$ then $y = 0$;
6) otherwise, $y = x$.

Fig. 1 shows an automaton that has a counterexample, where s_1 is an initial state marked with an incoming arrow, s_5 is an accepting state marked with a double circle, the path $s_3s_4s_5s_6s_3$ is an accepting cycle, and the path $s_1s_2s_3s_4s_5s_6s_3$ is a counterexample.

B. I/O Complexity Model

Because the access to information stored on an external memory device is orders of magnitude slower than the access

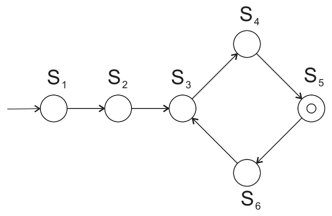


Fig. 1. Automaton with a counterexample.

to information stored in the internal memory [14], [15], complexities of external memory algorithms are usually measured in terms of the number of I/O operations. Here, an I/O operation is a transfer of data from a disk to internal memory or from internal memory to a disk. For example, for the benchmark $ITC'99$, $B15(std)$, $P1$, our algorithm costs in total 210 I/O operations when finding one counterexample.

For complexity analysis of external memory algorithms, a widely used model is the model of Aggarwal and Vitter [20]. In the model, the number of I/O operations is usually described by $O(scan(N))$ and $O(sort(N))$, standing for $O(N/B)$ and $O(N/B \cdot \log_{M/B}(N/B))$, respectively, where N denotes the total number of states of system, M denotes the number of states that fits into the internal memory, B denotes the number of states that can be transferred in a single I/O operation, and $O(N/B)$ denotes the same order as N/B.

IV. Related Work

Different I/O efficient algorithms for LTL model checking have been proposed. Most algorithms are based on nondepth-first search (non-DFS) which include breadth-first search (BFS) and A* [14], [21]–[27], because they utilize the delayed duplicate detection technique which is incompatible with DFS [25]. The technique needs to maintain a set of visited states on disk to prevent them from being reexplored. It is based on the observation [21] that a newly generated state does not need to be checked against the state table immediately; one can postpone the checking until an entire level of the search has been explored and then check all states in the level together by linearly reading the table from the disk.

To the best of our knowledge, in the last few years, among this kind of algorithms, DAC [14], MAP [27], and IDDFS [18] achieve state-of-the-art performance and represent the most recent advances.

The algorithm DAC [14] adapts an existing non-DFS-based accepting cycle detection algorithm one way catch them young [28] to the I/O efficient setting. The algorithm first inserts all reachable vertices into an approximation set. After that, it repeatedly reduces the approximation set until a fixpoint is reached. In detail, vertices violating the condition are gradually removed from the approximation set using two procedures. One procedure removes those vertices from the approximation set that lie outside any cycle. The other removes vertices lying on nonaccepting cycles. Finally, if the approximation set is empty, there is no accepting cycle in the graph, otherwise the presence of an accepting cycle is ensured. The algorithm is especially useful for verification of large systems with valid properties. But, it needs to create the whole state space.

Since DAC does not work on-the-fly, Barnat *et al.* [14], the same authors of DAC, further proposed an on-the-fly algorithm: MAP algorithm [17], [27], which is a revisiting resistant algorithm for I/O efficient LTL model checking. Revisiting resistant graph algorithms are those that significantly reduce the number of expensive I/O operations at the price of reexploration of edges (or vertices) in internal memory. They are actually an improvement on the delayed duplicate detection technique. The idea of the delayed duplicate detection technique is to postpone the duplicate check of single vertex against disk and perform them together in a group, for the reduction of the number of I/O operations. In the case of BFS traversal, the group (also called candidate set) consists typically of a single BFS level. However, if the level is small, the utility of delaying duplicate detection drops down. A possible solution is to maximize the group by exploring more BFS levels at once which will lead to revisiting of vertices due to cycles. However, even though vertex revisits result in performing more (cheap) operations in internal memory, it might significantly reduce the number of expensive I/O operations. Thus, revisiting resistant algorithms are expected to be more I/O efficient than nonresistant ones in practice. The main idea behind the MAP algorithm is based on the fact that each accepting vertex lying on an accepting cycle is its own accepting predecessor. Instead of expensive computing and storing of all accepting predecessors for each (accepting) vertex, the algorithm computes and stores a single representative accepting predecessor for each vertex, namely the maximal one in a suitable ordering of vertices. Experiments showed the algorithm outperformed previous I/O efficient algorithms on invalid LTL properties.

The IDDFS is a 5-bit semiexternal LTL model checking algorithm proposed in [18]. Semiexternal graph algorithms are algorithms, in which the vertices but not the edges fit in memory [29]. The IDDFS uses heuristic EPH to construct a minimal perfect hash function from the vertex set stored on disk, which allows compressing V to $5|V|$ bits, and only needs to store the $5|V|$ bits but not V into internal memory, where V is the vertex set of a graph. Thus, the algorithm can handle spaces that are orders of magnitudes larger than internal memory. However, IDDFS still has a limitation on the size of the graph because it needs 5 bits of internal memory for every vertex. This algorithm works on-the-fly by applying iterative-deepening strategy.

V. I/O Efficient Model Checking for Large-Scale Systems

In this section, we propose an I/O efficient LTL model checking algorithm based on the nested depth-first search [16], namely IOEMC. The IOEMC incorporates three key ideas: 1) a new linear hash-sorting algorithm LHS; 2) a new duplicate detection technique CDD; and 3) a dynamic search path management scheme DPM. By using linear hash-sorting algorithm and new duplicate detection technique, IOEMC can significantly reduce the time cost of duplicate detection of states. In addition, IOEMC also solves the memory dithering problem by using DPM technique.

We first present related data structures and memory usage. And then, LHS, CDD, and DPM techniques are proposed. Finally, we describe the model checking algorithm IOEMC, which is based on the above techniques. In the following, $\#(Q)$ expresses the number of elements in Q, and we assume that the algorithm can read or write B records from or into external memory in each I/O operation.

A. Data Structures and Memory Usage

The IOEMC needs a database DB on disk and two stacks $stack_1$ and $stack_2$ in the internal memory for two DFSs, respectively. There are four tables in DB, explained as follows.

1) *Tables $tableDD_1$ and $tableDD_2$:* They are used for duplicate detection of states and have the same data structure that consists of two fields: $state$ field and $hash$ field, which store those visited states and their hash values, respectively. Both $tableDD_1$ and $tableDD_2$ are sorted in nondecreasing order by hash values. Those records with the same hash value are located together.
2) *Tables $tableP_1$ and $tableP_2$:* They store the states on the path in the first DFS and ones on the path in the second DFS, respectively.

In our algorithm, the internal memory space is divided into code segment and data segment. The data segment is divided into two equal parts: T_1 and T_2. The first one is further divided into two subparts T_{11} and T_{12}, which are of the same size and store two hash tables H_1 and H_2 for two DFSs, respectively. Each element in H_1 and H_2 is a tuple (h, s), where s is a visited state, and h is the hash value of the state, and all elements in H_1 or H_2 are stored in time sequence. The other part T_2 is shared by $stack_1$ and $stack_2$ in a dynamic way.

Note that the aim of using tuples is to accelerate the search of disk tables $tableDD_1$ and $tableDD_2$, and to avoid hash collision. By using tuples, we cannot only search the disk tables sorted by hash values quickly, but also differentiate two different states even if they share the same hash value.

B. Linear Hash-Sorting Algorithm

In this section, we propose a linear hash-sorting algorithm LHS, which is used to reduce I/O complexity and improve the practical performance of our model checking algorithm.

The hash-sorting problem we consider in this paper is described as follows.

1) *Problem Instance:* There are m tuples in Q and n records in $tableDD$, where Q consists of the first $(\#(H) \cdot \rho_1)$ tuples of H, H may be H_1 or H_2, ρ_1 is a parameter with $0 < \rho_1 < 1$, and $tableDD$ may be $tableDD_1$ or $tableDD_2$, sorted in nondecreasing order by hash values. The records with the same hash value in $tableDD$ are located together.
2) *Goal:* Q is empty, and the m tuples are appended into $tableDD$, and the $(n + m)$ records in $tableDD$ are sorted in nondecreasing order by hash values.

The hash-sorting algorithm is outlined in Algorithm 1 and described as follows.

Algorithm 1 Linear Hash-Sorting Algorithm

```
input $Q$: a tuple set stored in the internal memory;
$tableDD$: database table in $DB$;
output $tableDD$: database table;
function $Merge\text{-}sort()$
var $t$, $s$: hash value; $Q_1$, $Q_2$: tuple set; $u$, $v$: state; $i$, $j$: integer;
begin
$Append(m$ empty records, $tableDD)$;
$i := n + m$;
$j := 0$;
repeat
  $Q_1 := read((n-(j+1)B+1), (n-jB), tableDD)$;
  $Empty((n-(j+1)B+1), (n-jB), tableDD)$;
  $t := min\{s|(s,u) \in Q_1\}$;
  $Q_2 := \{(s,u)|(s,u) \in Q, s \geq t\}$;
  $Store(Q_1 \cup Q_2, tableDD, i - |Q_1 \cup Q_2| + 1, i)$;
  $Delete(Q_1 \cup Q_2)$;
  $i := i - |Q_1 \cup Q_2|$;
  $j := j + 1$;
until ($Q$ is empty) or $(j = n/B)$
end
```

1) The algorithm appends m empty records into $tableDD$.
2) The algorithm reads B records from the $(n - B + 1)$th to the nth in $tableDD$ to the internal memory by invoking the function $read()$, and replaces these records in $tableDD$ with B empty records by calling the function $empty()$. Every record corresponds to a tuple. This needs only one I/O operation. Suppose t is the smallest hash value in the B records. If there are k_1 tuples in Q whose hash values are larger than or equal to t, then the algorithm moves $(B + k_1)$ tuples, including the k_1 tuples and the B tuples read before, into $tableDD$ to cover the records from the $(n + m - B - k_1 + 1)$th to the $(n+m)$th in turn by hash value's nondecreasing order, by invoking the function $store()$. After that, the algorithm deletes the $(B + k_1)$ tuples from the internal memory, by calling the function $Delete()$.
3) The algorithm continues to read B records from the $(n - 2B + 1)$th to the $(n - B)$th in $tableDD$ to the internal memory and replaces these records in $tableDD$ with B empty records, then does like (2).
4) This goes on until Q is empty or the first record in $tableDD$ is dealt with.

Note that we move only the first $(\#(H) \cdot \rho_1)$ tuples of H into external memory in order to improve the efficiency of duplicate detection (see Section V-C).

Lemma 1: The total number of I/O operations that the hash-sorting algorithm executes is smaller than or equal to $(m + 2n)/B$.

Proof: From the process described in Algorithm 1, we can observe that the ith loop mainly executes two types of operations: one is to read B records from disk to the internal memory; the other is to move $(B + k_i)$ tuples to disk, where $1 \leq i \leq n/B$. Thus, the ith loop needs $((B + k_i)/B + 1)$ I/O

TABLE I
STATES BEFORE MERGING. (a) STATES IN INTERNAL MEMORY. (b) STATES ON DISK

(a)
3561
3563
...
4402
4410
...
5811
5834

(b)
2101
2104
...
4406
4409
...
5832
5833
− − −

TABLE II
STATES AFTER MERGING. (a) STATES IN MEMORY. (b) STATES ON DISK. (c) FINAL RESULT

(a)
3561
3563
...
4402

(b)
2101
2104
...
4406
− − −
4409
4010
...
5811
5832
5833
5834

(c)
2101
2104
...
3561
3563
...
4402
...
4406
4409
4010
...
5811
5832
5833
5834

operations in total. Thus, the overall number of I/O operations the algorithm executes is $((B + k_1)/B + 1) + ((B + k_2)/B + 1) + \cdots + ((B + k_{n/B})/B + 1)$. Because $k_1 + k_2 + \cdots + k_{n/B}$ is equal to m, the total number of I/O operations is smaller than or equal to $(m + 2n)/B$. Thus, the I/O complexity of this algorithm is $O(n + m)$, linear in the input size. ■

In the following, we still use the example $ITC'99$, $B15(std)$ to illustrate how the hash storing algorithm is working. Note that every state is denoted by its hash value. Table I shows the states in the internal memory and the states in table on disk before merging, which are sorted in nondecreasing order by hash values. Our aim is to merge those in the internal memory into a table on disk. The last line "− − −" in Table I(b) means 1000 empty records are appended, where 1000 is equal to the number of the states in the internal memory. The 100 states can be transferred in a single I/O operation. After performing sequentially the following operations: 1) moving the last 100 states in Table I(b) (which are from 4409 to 5833) into the internal memory; 2) sorting them in the internal memory; and 3) moving the states in the internal memory whose hash value are greater than or equal to 4409 into Table I(b), the corresponding result is shown in Table II(a) and (b). This goes on until all records in Table I(b) are handled. The final result is shown in Table II(c) on the disk.

C. Cached Duplicate Detection Technique

In this section, we propose a cached duplicate detection technique CDD, which can significantly improve the performance of IOEMC.

In our duplicate detection technique, the visited states are divided into two groups: 1) the recent states; and 2) the historical states. The recent states are the ones generated most recently and stored in the hash table H in the internal memory, and the historical states are the ones stored in the hash table $tableDD$ on disk, where H may be H_1 or H_2, and $tableDD$ may be $tableDD_1$ or $tableDD_2$. When H is full, CDD invokes LHS to move only the first $(\#(H) \cdot \rho_1)$ tuples in H into $tableDD$ and sorts the new $tableDD$. Thus, the corresponding states become historical states.

In the following, we describe how to perform duplicate detection of a state. When a state is generated, CDD first checks by the hash value of the state whether or not it is in H in the internal memory, which costs no I/O operation. If this is the case, then the state was visited before. Otherwise, CDD further checks by the hash value of the state whether or not the state is in $tableDD$ in external memory. If this is the case, then the state was visited before; otherwise, the state is a new state.

In general, if we select $\rho_1 < 0.05$, then duplicate detections of almost all states are carried out in the internal memory.

We give an intuitive explanation as follows. Suppose the data segment is allocated 2G memory and every state needs 500 bits of internal memory and every hash value needs 12 bits. Then, H_1 should be 0.5G of size and can hold 2^{20} tuples nearest generated. When H_1 is full and the algorithm moves the first $(\#(H) \cdot \rho_1)$ tuples of the hash table H into external memory, the tuple number of H_1 is still close to 2^{20} because $\rho_1 < 0.05$. In general, the probability that the currently generated state goes back beyond the 2^{20} states nearest generated is extremely small. Therefore, for most states, their duplicate detections are executed in the internal memory.

To verify our argument, we conduct some experiments to figure out the proportion of states whose duplicate detections are performed in internal memory. The selected benchmarks and experimental environment are the same as that of Section VII. The experimental results are listed in Table III. The experiments show that for all models, the rates of the number of states whose duplicate detections are carried out in internal memory to that of all generated states are at least 90%. In addition, we also observe that setting $\rho_1 = 0.02$ yields the best performance for most large-scale models. The main cause is as follows. There is a tradeoff about time consumption in the setting of ρ_1. As the value of ρ_1 decreases, the number of visited states stored in internal memory increases, which saves time for duplicate detection, but this also results in more frequent movement of state blocks from the internal memory to the disk, leading to more time of state management, and vice versa.

The above analysis and experimental evidence show CDD can significantly reduce the cost of duplicate detection due to the fact that almost all duplicate detections of IOEMC are carried out in internal memory and $tableDD$ is sorted by hash values.

TABLE III
RATES OF DUPLICATE DETECTIONS IN INTERNAL MEMORY

benchmark	size	$\rho_1 = 0.01$	$\rho_1 = 0.02$	$\rho_1 = 0.03$	$\rho_1 = 0.04$	$\rho_1 = 0.05$
$Elevator2(16)$,P4	$173,916,122$	95%	97%	97%	95%	90%
$MCS(5)$,P4	$119,663,657$	93%	96%	95%	94%	91%
$Phils(16,1)$,P3	$61,230,206$	90%	99%	96%	94%	90%
$Lamport(5)$,P4	$74,413,141$	91%	98%	93%	90%	91%
$Peterson(6)$,P4	$4,187,461,216$	91%	96%	95%	92%	91%
$ITC'99, b15(std)$,P2	$163,840$	100%	100%	100%	100%	100%
$Szyman.(6)$,P4	$6,521,314,436$	92%	99%	95%	91%	90%
$Bakery(5,5)$,P3	$506,246,410$	96%	98%	95%	93%	91%
$Elevator2(16)$,P5	$173,916,122$	97%	98%	96%	95%	92%
$Szyman.(4)$,P2	$4,555,287$	100%	100%	100%	100%	100%
$ITC'99, b15(std)$,P2	$163,840$	100%	100%	100%	100%	100%
$Elevator2(7)$,P5	$43,776$	100%	100%	100%	100%	100%
$Lifts(7)$,P4	$73,473$	100%	100%	100%	100%	100%

D. Dynamic Search Path Management

The search path management includes the static and dynamic ones. The static management means the algorithm allocates fixed internal memory sections to two stacks of the nested depth-first search, while the dynamic one means two stacks share the same internal memory section. Thus, the dynamic management can more efficiently make use of the internal memory. In this section, we introduce a scheme for dynamic search path management, called DPM.

During search, when T_2 is full and a new state is generated, in order to avoid memory overflow, we need to move states from the two stacks to DB. However, this may result in the phenomenon that states are frequently moved inside and outside the internal memory.

In the following, we analyze how this phenomenon occurs. Suppose we swap M_2 states at a time between T_2 and the disk, where M_2 is the number of states that T_2 can hold. When T_2 is full, if we transfer all states in T_2 to the disk, then T_2 becomes empty. In succession, if the algorithm needs to pop a state from $stack_1$ or $stack_2$ because of backtracking, then immediately those M_2 states just moved to the disk have to be moved back to the internal memory, and thus T_2 becomes full. Afterward, if another new state is generated and needs to be pushed into $stack_1$ or $stack_2$, then the M_2 states have to be moved outside the internal memory again to make space for the new state, and so on. We call such a phenomenon of frequent state movement memory dithering, which can significantly increase disk accesses and thus the algorithm's I/O complexity.

In order to avoid memory dithering, we present an efficient scheme which works as follows.

1) When T_2 is full, the algorithm moves only some states (not all) of $stack_1$ and $stack_2$ into the database to make memory space for new states. For $stack_1$, the algorithm moves $k_1(= \#(stack_1)\cdot\rho_2)$ states from the bottom of the stack into $tableP_1$ by invoking the function $Append()$, and releases the corresponding memory space, where ρ_2 is a parameter with $0 < \rho_2 < 1$. Along with this, the bottom pointer of $stack_1$ points to the $(k_1 + 1)$th state. For $stack_2$, the algorithm handles it in the same way. This procedure is implemented in the $Dmem\text{-}DB()$ function, as shown in Algorithm 2.
2) When $stack_1$ becomes empty, if $tableP_1$ is not empty, then the algorithm pushes k_1 states stored most recently in $tableP_1$ into $stack_1$ in turn by calling the function $Push()$, and deletes them from $tableP_1$ by invoking $Delete()$, where $k_1 = min(\#(tableP_1), (M_2 - \#(stack_2)\cdot\rho_2))$. When $stack_2$ is empty, the algorithm works similarly. The corresponding procedure is implemented in $DDB\text{-}mem()$ function and is outlined in Algorithm 3.

Algorithm 2 Dynamic Full-Memory Management

```
input $stack_1, stack_2 : stack$; $tableP_1, tableP_2$: database table;
output $stack_1, stack_2 : stack$; $tableP_1$, $tableP_2$: database table;
function $Dmem\text{-}DB()$
var $Q$: state set;
begin
$Q$:=$\{s_1, \cdots, s_{k_1} | s_1, \cdots, s_{k_1}$ are $k_1$ states starting from bottom of $stack_1\}$;
$Append(Q, tableP_1)$;
$Btmpointer(stack_1, s_{k_1+1})$;
$Delete(Q, stack_1)$;
$Setfree(Q)$;

$Q$:=$\{t_1, \cdots, t_{k_2} | t_1, \cdots, t_{k_2}$ are $k_2$ states starting from bottom of $stack_2\}$;
$Append(Q, tableP_2)$;
$Btmpointer(stack_2, s_{k_2+1})$;
$Delete(Q, stack_2)$;
$Setfree(Q)$;
end
```

Note that by reserving some states in both stacks when T_2 is full, we ensure there are always some states in both stacks, which to some extend reduces the memory dithering phenomenon.

E. Model Checking Algorithm IOEMC

Based on LHS, CDD, and DPM, we design the IOEMC algorithm. The IOEMC is an I/O efficient on-the-fly model checking algorithm based on the nested depth-first search.

We use V_0 to denote the initial state set and F to denote the accepting state set. Also, we assume the state transition graph of the automata is implicitly given from the function $successor(x)$ that generates all successors of the state x.

Algorithm 3 Dynamic Empty-Stack Management

```
input $stack_i$ : $stack$; $tableP_i$: database table;
/*$i$ may be 1 or 2 */
output $stack_i$ : $stack$; $tableP_i$: database table;
function $DDB$-$mem()$
var $Q$: state set;
begin
$Q$:={$s_1, \cdots, s_k | s_1, \cdots, s_k$ are the last $k$ states stored in $tableP_i$ };
/* $k$ is equal to $min(\#(tableP_i), (M_2 - \#(stack_j) \cdot \rho_2))$ where $j$ is equal to 1 if $i = 2$, or 2 if $i = 1$ */
$Push(Q, stack_i)$;
$Delete(Q, tableP_i)$;
end
```

The first DFS, outlined in Algorithm 4, is to search for a path from an initial state to some accepting state by postorder traversal. In each while loop, for the current state x, the algorithm first performs duplicate detection for successor s by using CDD technique. If s is a new state, then the algorithm puts s into $stack_1$ and puts the tuple $(hash(s), s)$ into H_1. Before these two operations, the algorithm needs to check whether or not $stack_1$ and T_{11} are full. If the former is full, then it invokes the function $DMem\text{-}DB()$ to handle the stack; if the latter is full, then it puts a part of tuples of H_1 into $tableDD_1$ and sorts the new $tableDD_1$ by invoking the function $Merge\text{-}sort()$. The number of tuples moved into $tableDD_1$ is determined by the parameter ρ_1. If all successors of state x have been visited, then the algorithm pops out x from $stack_1$; if x is an accepting state, then the algorithm goes into the second DFS.

The second DFS is to detect an accepting cycle and finally return a counterexample if the system does not satisfy the given specification. The counterexample consists of an accepting cycle and a path to the cycle from some initial state. The second DFS works similarly as the first DFS, and is outlined in Algorithm 5.

Algorithm 4 First DFS Based on Quick Hash-Sorting Algorithm

```
input $q$: an initial state in $V_0$;
output without output;
function $DFS_1()$
var $Q$, $T$: tuple set; $t$: tuple;
begin
$stack_1 = \emptyset; stack_2 = \emptyset; H_1 := \emptyset; H_2 := \emptyset; Q := \emptyset$;
/*$\emptyset$ denotes empty set*/
Push $q$ into $stack_1$;
Put $(hash(q), q)$ into $H_1$;

while $stack_1 \neq \emptyset$ do
  $x := top(stack_1)$;
  $Q := successor(x)$;
  if $\exists s \in Q$ and $(hash(s), s) \notin H_1$ then
    if $check(s, tableDD_1) = 0$ then
      if $stack_1$ is full then
        $Dmem$-$DB()$;
      end if
      Push $s$ into $stack_1$;
      if $T_{11}$ is full then
        $T$=:{$t | t$ is in the first $(\#(H_1) \cdot \rho_1)$ tuples of $H_1$};
        $Merge$-$sort(T, tableDD_1)$;
      end if
      Put $(hash(s), s)$ into $H_1$;
    end if
  else
    Pop $x$ from $stack_1$;
    if $stack_1$ =$\emptyset$ and $tableP_1 \neq \emptyset$ then
      $DDB$-$mem(stack_1, tableP_1)$;
    end if
    if $x \in F$ then
      $DFS_2(x)$;
    end if
  end if
end while
end
```

VI. Complexity Analysis

In this section, we analyze the I/O complexity of IOEMC. Note that the correctness of IOEMC can be easily proved by following the similar proof line in [16]. In the following, we let N express the number of states of the verified system and M be the number of states that the internal memory can hold.

A. Complexity of Algorithm IOEMC

In this section, we estimate the I/O complexity of IOEMC.

Lemma 2: The total I/O operations that dynamic search path management needs in the nested depth-first search is $O(scan(N))$.

Proof: The worst case is the one with the most number of I/O operations. Thus, the algorithm should traverse all states and pop them out from the two stacks, and the number of states moved from the internal memory to the disk is equal to the number of states moved from the disk to the internal memory. Thus, when T_2 is full, $(M_2 \cdot \rho_2)$ states are moved from the internal memory to the disk, and then $(M_2 \cdot (1 - \rho_2))$ states are continuously popped out from the two stacks one by one and T_2 becomes empty, and then $(M_2 \cdot \rho_2)$ states have to be transferred to the internal memory from the disk again; afterward, the algorithm pushes continuously $(M_2 \cdot (1 - \rho_2))$ states into the stacks one by one and T_2 becomes full, and then it moves $(M_2 \cdot \rho_2)$ states from the internal memory to the disk again, and this goes on until all states of system are traversed. In this process, the block of states is moved from the internal memory to the disk $((N - M_2 \cdot \rho_2)/(M_2 \cdot (1 - \rho_2)) - 1)$ times. Because every movement of a block of states needs $(M_2 \cdot \rho_2/B)$ I/O operations and the number of states moved from the internal memory to the disk is equal to that from the disk to the internal memory, the whole process costs $2((N - M_2 \cdot \rho_2)/$

Algorithm 5 Second DFS Based on Quick Hash-Sorting Algorithm

```
input $x$: an accepting state in $F$;
output a counterexample ( if the system does satisfy the given specification);
function $DFS_2()$
var $Q$, $T$: tuple set; $t$: tuple;
begin
Push $x$ into $stack_2$;
Put$(hash(x), x)$ into $H_2$;

while $stack_2 \neq \emptyset$ do
  $v := top(stack_2)$; $Q := successor(v)$;
  if $x \in Q$ then
    $return(tableP_1 + stack_1 + tableP_2 + stack_2)$;
    /* the operator '+' expresses the concatenation of two state sequences */
  end if
  if $\exists s \in Q$ and $(hash(s), s) \notin H_2$ then
    if $check(s, tableDD_2) = 0$ then
      if $stack_2$ is full then
        $Dmem\text{-}DB()$;
      end if
      Push $s$ into $stack_2$;
      if $T_{12}$ is full then
        $T$=:$\{t|t$ is in the first $(\#(H_2) \cdot \rho_1)$ tuples of $H_2\}$;
        $Merge\text{-}sort(T,\ tableDD_2)$;
      end if
      Put$(hash(s), s)$ into $H_2$;
    end if
  else
    Pop $v$ from $stack_2$;
    if $stack_2 = \emptyset$ and $tableP_2 \neq \emptyset$ then
      $DDB\text{-}mem(stack_2,\ tableP_2)$;
    end if
  end if
end while
end
```

$(M_2 \cdot (1 - \rho_2)) - 1) \cdot (M_2 \cdot \rho_2 / B)$ I/O operations. Therefore, the total I/O operations that dynamic search path management needs in the nested depth-first search is $O(\rho_2/(1 - \rho_2) \cdot scan(N))$, i.e., $O(scan(N))$. ■

Theorem 1: The I/O complexity of algorithm IOEMC is $O(N/M \cdot scan(N) + sort(|E|))$.

Proof: From Section V-E, we can see the I/O complexity of IOEMC is determined by the functions $Merge\text{-}sort()$, $check()$, $Dmem\text{-}DB()$, and $DDB\text{-}mem()$. For $Merge\text{-}sort(H_1, tableDD_1)$, in the worst case, when the while loop ends, $Merge\text{-}sort(H_1, tableDD_1)$ is invoked N/M_{11} times, where M_{11} is the number of the states that T_{11} can hold. From Lemma 1, the total number of I/O operations that it executes is at most $(M_{11} + (M_{11} + 2M_{11}) + (M_{11} + 2 \cdot 2M_{11}) + \cdots + (M_{11} + 2(N/M_{11} - 1) \cdot M_{11}))/B$ which is smaller than $N/B + N^2/(B \cdot M_{11})$, namely, $(1 + N/M_{11}) \cdot scan(N)$. Similarly, when the while loop ends, the total number of I/O operations executed by Merge-$sort(H_2, tableDD_2)$ is at most $(1+N/M_{12}) \cdot scan(N)$, where M_{12} is the number of the states that T_{12} can hold. From Algorithm 4, we can see that on the hand, the number of check operations (namely, $check(s, tableDD_1)$) is equal to at most the number $|E|$ of edges; on the other hand, every check operation can cause that CDD checks by the hash value of a state whether or not the state is in the sorted $tableDD_1$ in external memory, thus, the number of I/O operations in every check is $\log_B |E|$. Thus, the overall number of I/O operations invoked by $check(s, tableDD_1)$ is $O(|E| \cdot \log_B N) = O(\log_B(M/B) \cdot sort(|E|))$=$O(sort(|E|))$. For $check(s, tableDD_2)$, we have the same result as $check(s, tableDD_1)$. From Lemma 2, we have the total number of the I/O operations that invoking of $Dmem\text{-}DD()$ and $DDB\text{-}mem()$ needs is $O(scan(N))$. Because T_1 and T_2 are of the same size and T_{11} and T_{12} are of the same size, the I/O complexity of algorithm IOEMC is $O(N/M \cdot scan(N) + sort(|E|))$. ■

B. Complexity Comparison

In this section, we compare IOEMC with DAC, MAP, and IDDFS, in terms of I/O complexity.

The DAC is an I/O efficient algorithm for accepting cycle detection proposed in [14]. The [14, Th. 1] claims the I/O complexity of DAC is $O(l_{SCC} \cdot (h_{\text{BFS}} + |pmax| + |E|/M) \cdot scan(N))$, where l_{SCC} denotes the length of the longest path in the SCC graph, and $pmax$ is the longest path in the graph going through trivial strongly connected components (without self-loops), and h_{BFS} is the BFS height, and $|E|$ is the number of edges. Because both $N/M \cdot scan(N)$ and $sort(|E|)$ are clearly smaller than $|E|/M \cdot scan(N)$, IOEMC has much lower I/O complexity than DAC.

For MAP, I/O complexity is $O(|F|((d + |E|/M + |F|) scan(N) + sort(N)))$ in the case for candidate set in RAM, and $O(|F|((d+|F|)scan(N)+sort(|F| \cdot |E|)))$ in the case for candidate set on disk, where d is diameter of the graph [27]. Because $|E|$ is larger than N, I/O complexity of IOEMC is much lower than that of MAP in the case for candidate set in RAM. In the case for candidate set on disk, from Section III-A, we can observe that $|F|$ is equal to $N/3$ for the intersection of automata $\mathcal{A}$ and $\mathcal{S}$. It follows that I/O complexity of MAP is $O(N^3)$. Thus, IOEMC outperforms MAP in terms of I/O complexity.

The IDDFS is a semiexternal algorithm proposed in [18]. Its I/O complexity is $O(\epsilon_s \cdot sort(N) + sort(|E|))$, where $\epsilon_s = max\{\delta(s, v)|v \in V\}$ [18] is the maximal BFS level and $\delta(s, v)$ is the length of a shortest path from s to v. Thus, IDDFS is especially useful for the model checking problem of systems with small number of BFS levels. Assume that each state needs k bits of internal memory space, then for 5-bit semiexternal search on a computer with m GB RAM, IDDFS cannot solve the model checking problem for the systems with larger size than $(m \cdot 2^{30} \cdot 8/5)$ states [18]. Thus, for the systems that IDDFS can verify, N/M is smaller than $(m \cdot 2^{30}/5)/(m \cdot 2^{30}/k) = k/5$. Hence, the corresponding I/O complexity of IOEMC is $O((k/5) \cdot scan(N) + sort(|E|))$,

TABLE IV
EXPERIMENTAL RESULT ON MODELS WITH VALID PROPERTIES

		DAC		MAP		$IDDFS$		$IOEMC$	
benchmark	size	time	disk	time	disk	time	disk	time	disk
$Elevator2(16)$,P4	$173,916,122$	11:21:13	9.20 GB	07:46:42	16 GB	08:01:15	10 GB	03:01:23	10.85 GB
$MCS(5)$,P4	$119,663,657$	03:35:45	8 GB	04:58:51	10 GB	03:57:05	6.2 GB	01:45:32	6.87 GB
$Phils(16,1)$,P3	$61,230,206$	02:00:35	5.5 GB	02:30:21	7.8 GB	02:52:19	6.7 GB	01:02:35	7.48 GB
$Lamport(5)$,P4	$74,413,141$	03:45:52	3.51 GB	03:16:25	5.7 GB	02:25:21	3.31 GB	01:10:57	4.12 GB
$ITC'99, b15(std)$,P2	$163,840$	00:05:40	66 MB	00:04:36	53 MB	00:04:13	47 MB	00:02:01	78 MB
$Peterson(6)$,P4	$4,187,461,216$	32:26:16	136.32 GB	30:54:32	124.32 GB	cannot handle		12:02:34	136.65 GB
$Szyman.(6)$,P4	$6,521,314,436$	37:54:41	165.21 GB	34:33:41	151.32 GB	cannot handle		15:08:23	166.53 GB

TABLE V
EXPERIMENTAL RESULT ON MODELS WITH INVALID PROPERTIES

		DAC		MAP		$IDDFS$		$IOEMC$	
benchmark	size	time	disk	time	disk	time	disk	time	disk
$Bakery(5,5)$,P3	$506,246,410$	68:31:45	39 GB	00:00:18	16 MB	00:01:25	71 MB	00:00:07	11 MB
$Elevator2(16)$,P5	$173,916,122$	00:02:10	125 MB	00:01:22	69 MB	00:01:16	65 MB	00:00:24	30 MB
$Szyman.(4)$,P2	$4,555,287$	00:21:15	221 MB	00:00:12	2 MB	00:00:52	8 MB	00:00:03	3 MB
$ITC'99, b15(std)$,P1	$163,840$	00:01:12	26 MB	00:01:10	26 MB	00:00:56	21 MB	00:01:25	30 MB
$Elevator2(7)$,P5	$43,776$	00:00:26	61 MB	00:00:11	2 MB	00:00:35	6 MB	00:00:41	9 MB
$Lifts(7)$,P4	$73,473$	00:23:17	536 MB	00:00:26	4.6 MB	00:06:48	239 MB	00:00:12	8 MB

TABLE VI
RUN TIMES WITH DIFFERENT VALUES OF THE PARAMETER ρ_2

benchmark	size	$\rho_2 = 0.75$	$\rho_2 = 0.80$	$\rho_2 = 0.85$	$\rho_2 = 0.90$	$\rho_2 = 0.95$
Models with valid property						
$Elevator2(16)$,P4	$173,916,122$	07:51:35	05:32:42	05:12:45	**03:01:23**	05:45:51
$MCS(5)$,P4	$119,663,657$	01:36:38	**01:10:51**	01:16:39	01:45:32	02:18:22
$Phils(16,1)$,P3	$61,230,206$	02:21:54	01:26:13	01:32:47	**01:02:35**	02:26:27
$Lamport(5)$,P4	$74,413,141$	02:16:53	01:26:54	**01:08:45**	01:10:57	02:38:53
$Peterson(6)$,P4	$4,187,461,216$	15:15:46	13:35:41	13:56:43	**12:52:34**	15:32:11
$Szyman.(6)$,P4	$6,521,314,436$	19:37:15	17:16:45	**15:00:12**	15:08:23	18:41:28
Models with invalid property						
$Bakery(5,5)$,P3	$506,246,410$	00:00:48	00:00:17	**00:00:05**	00:00:07	00:00:51
$Elevator2(16)$,P5	$173,916,122$	00:00:41	00:00:35	00:00:32	**00:00:24**	00:00:29

namely $O(sort(|E|))$. Therefore, IOEMC has a lower I/O complexity than IDDFS.

VII. EXPERIMENT

In this section, we compare runtime and allocated disk space of IOEMC with that of DAC, MAP, and IDDFS.

A. Benchmarks

In order to compare the performance of IOEMC with that of DAC, MAP, and IDDFS, we selected benchmarks from [14], [18], [19] and added models $Peterson(6)$, $P4$ and $Szyman.(6)$, $P4$. The two models are to show the limitation of scale of systems IDDFS can verify. All selected benchmarks are from the BEEM project [30], which include models with valid properties and models with invalid properties, ranging from less than 50 000 states to more than 6 000 000 000 states. They are typical ones in the literatures and serve as a good test bed to justify the efficiency and performance of model checking algorithms.

B. Experimental Setup

The four algorithms have been implemented on top of the DiVine library [31], providing the state space generator, and the STXXL library [32], providing the I/O primitives. For IOEMC, we set the parameters $\rho_1 = 0.02$ and $\rho_2 = 0.90$.

All experiments were run on a PC with CPU $P4$ 2.4 G, memory 2G, disk space 400 GB, and Linux 9.0 operation system. For each instance, each algorithm is performed 100 runs. For each algorithm on each instance, we report the average runtime (time) and average disk consumption (disk). The time format is hh:mm:ss (the elapsed hours, minutes, and seconds).

C. Experimental Results

Experimental results on models with valid properties are presented in Table IV. As is clear in Table IV, IOEMC verifies these valid models significantly faster than other algorithms. On the five benchmarks, namely $Elevator2(16)$, P4, $MCS(5)$, P4, $Phils(16, 1)$, P3, $Lamport(5)$, P4, and $ITC'99$, $b15(std)$, P2, for which all the algorithms verify the validity within 10 h, IOEMC does this two to three times faster than other algorithms. For the two hard benchmarks $Peterson(6)$, P4 and $Szyman.(6)$, P4, IDDFS fails to handle them due to internal memory shortage, as it is a semiexternal algorithm which needs five extra bits of internal memory for every state. In addition, both DAC and MAP need more than 30 h to verify each of these two benchmarks, while IOEMC only needs 12 and 15 h for $Peterson(6)$, P4 and $Szyman.$ (6), P4, respectively. Nevertheless, as IOEMC needs to store not only state, but also its hash value on disk for every state, it has a bit more space consumption than other algorithms on models with valid properties.

Experimental results on models with invalid properties are reported in Table V. An obvious observation from Table V is that all the algorithms but DAC can find a counterexample for these benchmarks very quickly (within several minutes). Notably, IOEMC dominates other algorithms on four benchmarks, namely, Bakery(5,5),P3, $Elevator2(16)$,P5, Szyman(4),P2, and Lifts(7),P4, in terms of time consumption. For these three benchmarks, IOEMC performs at least two times faster than the best of other algorithms. Admittedly, for two of small models, namely, $ITC'99, b15(std)$,P1, and $Lifts(7)$,P4, IOEMC is a bit slower than other three algorithm. This is because the three new techniques used in IOEMC is designed to reduce the number of I/O operations for large models, and has no effect for small models.

VIII. Discussion

A. ρ_2 Parameter

The ρ_2 parameter must be provided in order to execute IOEMC. It influences the performance of IOEMC by controlling the size of state blocks moved into or out of the internal memory.

To observe the parameter's impact on the performance of IOEMC, we run IOEMC with different values of the ρ_2 parameter for each of the used instances. The selected benchmarks and experimental environment are the same as that of Section VII except for models $ITC'99, b15(std)$,P2, $Szyman.(4)$,P2, $ITC'99, b15(std)$,P1, $Elevator2(7)$,P5, and $Lifts(7)$,P4, because small models do not need to use DPM technique. The experimental results are reported in Table VI.

From Table VI, we observe that the ρ_2 parameter has an obvious impact on the performance of IOEMC, and there are different optimal values of the parameter between different instances, and the optimal values mainly are between 0.85 and 0.90. The investigation about adjusting the ρ_2 parameter automatically is left for future work.

B. State Number Limit

In this section, we discuss the state number limit over which our approach cannot get a solution in reasonable time. We may regard 24 h as reasonable time, and assume the computer has a 7200-r/min desktop hard disk drive (HDD) and a Serial Advanced Technology Attachment (SATA) bus interface, and every state occupies 100 bits in our approach.

According to [33], as of 2010, a typical 7200-r/min desktop HDD has a disk-to-buffer data transfer rate up to 1030 Mb/s, and a widely used standard for the buffer-to-computer interface is 3.0-Gb/s SATA. Thus, the transfer rate between disk and computer is less than 772 Mb/s. In order to find a counterexample, if the state number to be searched is larger than $772 \times 1024 \times 1024 \times 24 \times 3600/100$ ($<6.68 \times 10^{10}$), then our approach cannot carry out this operation successfully in reasonable time (24 h). Thus, a state number limit to our approach is about 6.68×10^{10}.

IX. Conclusion

In this paper, we proposed and introduced a linear hash-sorting algorithm LHS, a cached duplicate detection technique CDD, and a dynamic search path management technique DPM. Based on the above techniques, we proposed an I/O efficient LTL model checking algorithm for large-scale systems. We have implemented our model checking algorithm, and carried out the experiments on selected representative benchmarks. The complexity analysis and the experimental results show IOEMC has lower I/O complexity and obviously better practical performance than state-of-the-art I/O efficient algorithms, including DAC, MAP, and IDDFS. The low I/O complexity and good performance indicate that IOEMC is very promising for verifying large-scale systems efficiently.

Acknowledgment

The authors would like to thank the anonymous reviewers for their helpful comments on earlier versions of this paper.

References

[1] K. L. McMillan, "A methodology for hardware verification using compositional model checking," *Sci. Comput. Program.*, vol. 37, nos. 1–3, pp. 279–309, 2000.

[2] J. Yang and C.-J. H. Seger, "Introduction to generalized symbolic trajectory evaluation," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 11, no. 3, pp. 345–353, Jun. 2003.

[3] H. Seger *et al.*, "An industrially effective environment for formal hardware verification," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 24, no. 9, pp. 1381–1405, Sep. 2005.

[4] M. Talupur, "Hardware model checking: Status, challenges, and opportunities," in *Proc. Int. Conf. Formal Methods Comput.-Aided Design (FMCAD)*, 2011, pp. 154–160.

[5] E. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*, 1st ed. Cambridge, MA, USA: MIT Press, 1999, pp. 121–140.

[6] P. Godefroid and D. Pirottin, "Refining dependencies improves partial-order verification methods," in *Proc. Comput. Aided Verificat. (CAV)*, 1993, pp. 438–449.

[7] D.-H. Chu and J. Jaffar, "A complete method for symmetry reduction in safety verification," in *Proc. Int. Conf. Comput. Aided Verificat. (CAV)*, 2012, pp. 616–633.

[8] E. M. Clarke, O. Grumberg, and D. E. Long, "Model checking and abstraction," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 5, pp. 1512–1542, 1993.

[9] J.-P. Krimm and L. Mounier, "Compositional state space generation from Lotos programs," in *Proc. Tools Algorithms Construct. Anal. Syst. (TACAS)*, 1997, pp. 239–258.

[10] G. Morbé, F. Pigorsch, and C. Scholl, "Fully symbolic model checking for timed automata," in *Proc. 23rd Int. Conf. Comput. Aided Verificat. (CAV)*, 2011, pp. 616–632.

[11] G. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*. Reading, MA, USA: Addison-Wesley, 2004, pp. 1–596.

[12] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," *Adv. Comput.*, vol. 58, no. 5, pp. 117–148, 2003.

[13] R. E. Korf, "Linear-time disk-based implicit graph search," *J. ACM*, vol. 55, no. 6, pp. 1–40, 2008.

[14] J. Barnat, L. Brim, and P. Simecek, "I/O efficient accepting cycle detection," in *Proc. Int. Conf. Comput. Aided Verificat. (CAV)*, 2007, pp. 281–293.

[15] S. Jabbar and S. Edelkamp, "Parallel external directed model checking with linear I/O," in *Proc. 7th Int. Conf. Verificat., Model Check., Abstract Interpret. (VMCAI)*, 2006.

[16] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis, "Memory efficient algorithms for the verification of temporal properties," *Formal Methods Syst. Design*, vol. 1, nos. 2–3, pp. 275–288, 1990.

[17] L. Brim, I. Cerna, P. Moravec, and J. Simsa, "Accepting predecessors are better than back edges in distributed LTL model-checking," in *Proc. 5th Int. Conf. Formal Methods Comput.-Aided Design (FMCAD)*, vol. 3312. 2004, pp. 352–366.

[18] S. Edelkamp, P. Sanders, and P. Šimeček, "Semi-external LTL model checking," in *Proc. Int. Conf. Comput. Aided Verificat. (CAV)*, 2008, pp. 530–542.
[19] (1999). *ITC'99* [Online]. Available: https://lis.ei.tum.de/projects/faultify/browser/hardware/benchmark_circuits/itc99/itc99-poli2
[20] A. Aggarwal and J. S. Vitter, "The input/output complexity of sorting and related problems," *Commun. ACM*, vol. 31, no. 9, pp. 1116–1127, 1988.
[21] U. Stern and D. Dill, "Using magnetic disk instead of main memory in the Mur φ verifier," in *Proc. 10th Int. Conf. Comput. Aided Verificat. (CAV)*, 1998, pp. 172–183.
[22] R. E. Korf, "Best-first frontier search with delayed duplicate detection," in *Proc. 19th Nat. Conf. Artif. Intell. (AAAI)*, 2004, pp. 650–657.
[23] R. E. Korf and P. Schultze, "Large-scale parallel breadth-first search," in *Proc. 20th Nat. Conf. Artif. Intell. (AAAI)*, 2005, pp. 1380–1385.
[24] K. Mehlhorn and U. Meyer, "External-memory breadth-first search with sublinear I/O," in *Proc. 10th Annu. Eur. Symp. Algorithms (ESA)*, 2002, pp. 723–735.
[25] S. Edelkamp and S. Jabbar, "Large-scale directed model checking LTL," in *Proc. SPIN*, 2006, pp. 1–18.
[26] J. Barnat, L. Brim, and J. Chaloupka, "Parallel breadth-first search LTL model-checking," in *Proc. 18th IEEE Int. Conf. Autom. Softw. Eng. (ASE)*, Oct. 2003, pp. 106–115.
[27] J. Barnat, L. Brim, P. Šimeček, and M. Weber, "Revisiting resistance speeds up I/O-efficient LTL model checking," in *Proc. 14th Int. Conf. Tools Algorithms Construct. Anal. Syst. (TACAS)*, vol. 4963. 2008, pp. 48–62.
[28] I. Cerna and R. Pelanek, "Distributed explicit fair cycle detection," in *Proc. SPIN*, vol. 2648. 2003, pp. 49–73.
[29] J. Abello, A. L. Buchsbaum, and J. R. Westbrook, "A functional approach to external graph algorithms," in *Proc. 6th Annu. Eur. Symp. Algorithms (ESA)*, 1998, pp. 332–343.
[30] R. Pelanek, "BEEM: Benchmarks for explicit model checkers," in *Proc. SPIN*, vol. 4595. 2007, pp. 263–267.
[31] J. Barnat, L. Brim, I. Cerna, P. Moravec, P. Rockai, and P. Šimeček, "DiVinE: A tool for distributed verification," in *Proc. 18th Int. Conf. Comput. Aided Verificat. (CAV)*, vol. 4144. 2006, pp. 278–281.
[32] R. Dementiev, L. Kettner, and P. Sanders, "STXXL: Standard template library for XXL data sets," in *Proc. 13th Ann. Eur. Symp. Algorithms (ESA)* vol. 3669. 2005, pp. 640–651.
[33] Wikipedia. (2013, Mar. 10). *Hard Disk Drive* [Online]. Available: http://en.wikipedia.org/wiki/Hard_disk_drive

Lijun Wu was born in 1965.

He is an Associate Professor with the University of Electronic Science and Technology of China, Chengdu, China. His current research interests include formal methods and artificial intelligence.

Huijia Huang was born in 1988. She is currently pursuing the Degree with the University of Electronic Science and Technology of China, Chengdu, China.

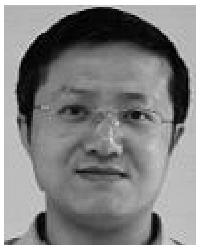
Her current research interests include formal methods and artificial intelligence.

Kaile Su was born in 1964.

He is a Professor with Griffith University, Brisbane, QLD, Australia. His current research interests include formal methods and artificial intelligence.

Shaowei Cai received the Ph.D. degree in computer science from Peking University, Beijing, China, in 2012.

His current research interests include optimization, heuristics, and randomized algorithms.



Xiaosong Zhang was born in 1966.

He is a Professor and Ph.D. Supervisor with the University of Electronic Science and Technology of China, Chengdu, China. His current research interests include network and information security.