Caching, Hashing, and Garbage Collection for Distributed State Space Construction

Ming-Ying Chung [1] Gianfranco Ciardo [2]
Radu I. Siminiceanu [3]

Abstract

The *Saturation* algorithm for symbolic state-space generation is a recent advance in exhaustive verification of complex systems, in particular globally-asynchronous/locally-synchronous systems. The distributed version of Saturation uses the overall memory available on a network of workstations (NOW) to efficiently spread the memory load during its highly irregular exploration. A crucial factor in limiting the memory consumption in symbolic state-space generation is the ability to perform *garbage collection* to free up the memory occupied by dead nodes. However, garbage collection over a NOW requires a nontrivial communication overhead. In addition, operation cache policies become critical while analyzing large-scale systems using a symbolic approach. In this paper, we develop a garbage collection scheme and several operation cache policies to help the analysis of complex systems. Experiments show that our schemes improve the performance of the original distributed implementation, SMARTNOW, in terms of both time and memory efficiency.

Keywords: state-space generation, decision diagrams, distributed computing, hashing, cache policy.

1 Introduction

Formal verification techniques such as model checking and theorem proving have received wide adoption in industry for quality assurance, as they can be used to detect errors early in the design lifecycle. State-space generation, also called reachability analysis, is an essential and often memory-intensive step in model checking. The increasing complexity of system designs stresses the limits of most model checkers. Even though symbolic state-space encodings based on *binary decision diagrams* (BDDs) [2] and *multiway decision diagrams* (MDDs) [20] have helped cope with the inherent state-space explosion

⋆ This work was supported in part by the National Aeronautics and Space Administration under the cooperative agreement NCC-1-02043

[1] Email:chung@cs.ucr.edu University of California, Riverside, CA 92521
[2] Email:ciardo@cs.ucr.edu University of California, Riverside, CA 92521
[3] Email:radu@nianet.org National Institute of Aerospace, Hampton, VA 23666

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

of discrete-state systems, the analysis of many industrial-size models may still rely on the use of virtual memory.

A natural way to deal with the excessive memory consumption of symbolic reachability analysis is to use parallel and distributed approaches. Most of the research in this area has focused on *vertical* slicing schemes to parallelize BDD manipulations, by decomposing boolean functions and distributing the computation over a NOW [18,21,25]. This allows algorithms to overlap the *image computation* (the application of the next-state function to a set of substates encoded by a decision diagram node), but the distributed state-space generation is still synchronous, consisting of interleaved rounds of computation and communication, in which the fastest or the most lightly loaded workstation must wait for the heavily loaded ones at the end of each round. Such a periodic global synchronization is a major obstacle to scalability. To overcome this drawback, Grumberg et al. [17] introduced an asynchronous version of the vertical slicing approach which not only performs image computation and message passing concurrently, but also incorporates an adaptive mechanism that takes into account the available computational power to split the workload.

In [5], we use instead MDDs and partition them *horizontally* onto a NOW, so that each workstation exclusively owns a contiguous range of MDD levels. Therefore, the memory required for state-space encoding is mutually exclusively partitioned onto workstations. This horizontally distributed state-space generation does not create any redundant or duplicate work. Furthermore, it uses only peer-to-peer communication between neighboring orkstations, avoiding global synchronizations and allowing for good scalability. However, this approach comes with a severe tradeoff. Given the highly optimized nature of Saturation, which was designed as a sequential algorithm, only one workstation is active at any time, hence the distributed computation is virtually sequentialized. This leaves only limited opportunities for speed-up. For example, in [6,7], we introduced a way to speed-up this scheme by using workstations' idle time to speculatively perform image computations.

Also, during distributed state-space generation on a NOW, performing *garbage collection* for dead MDD nodes requires a nontrivial communication overhead. In addition, MDD cache policies become relatively critical in a large-scale symbolic reachability analysis. Thus, in this paper, we develop a garbage collection scheme and several operation cache policies to help solving extremely complex systems.

The paper is organized as follows. Section 2 gives the necessary background on state-space generation, decision diagrams, Kronecker encoding, and the evolution of the Saturation algorithm. Section 3 details our new garbage collection scheme tailor-made for distributed state-space generation. Section 4 discusses several operation cache policies which might help solving complex systems. Section 5 shows experimental results. Section 6 draws conclusions and discusses future research directions.

2 Background

A discrete-state model is a triple $(\widehat{\mathcal{S}}, \mathbf{s}^{init}, \mathcal{N})$, where $\widehat{\mathcal{S}}$ is the set of *potential states* of the model, $\mathbf{s}^{init} \in \widehat{\mathcal{S}}$ is the *initial state*, and $\mathcal{N} : \widehat{\mathcal{S}} \to 2^{\widehat{\mathcal{S}}}$ is the *next-state function* specifying the states reachable from each state in a single step. We assume that the model is composed of K *submodels*. Thus, a (*global*) state $\mathbf{i}$ is a K-tuple $(\mathbf{i}_K, ..., \mathbf{i}_1)$, where $\mathbf{i}_k$ is the *local* state of submodel k, $K \geq k \geq 1$, and $\widehat{\mathcal{S}} = \mathcal{S}_K \times \cdots \times \mathcal{S}_1$ is the cross-product of K *local state spaces*. This allows us to use techniques targeted at exploiting system structure, in particular, symbolic techniques to store the state space based on decision diagrams.

Since we target globally-asynchronous locally-synchronous systems, we decompose $\mathcal{N}$ into a disjunction of next-state functions [4]: $\mathcal{N}(\mathbf{i}) = \bigcup_{e \in \mathcal{E}} \mathcal{N}_e(\mathbf{i})$, where $\mathcal{E}$ is a finite set of *events* and $\mathcal{N}_e$ is the next-state function associated with event e. We then seek to build the *(reachable) state space* $\mathcal{S} \subseteq \widehat{\mathcal{S}}$, the smallest set containing $\mathbf{s}^{init}$ and closed with respect to $\mathcal{N}$, that is:

$$\mathcal{S} = \{\mathbf{s}^{init}\} \cup \mathcal{N}(\mathbf{s}^{init}) \cup \mathcal{N}(\mathcal{N}(\mathbf{s}^{init})) \cup \cdots = \mathcal{N}^*(\mathbf{s}^{init}),$$

where "*" denotes reflexive and transitive closure and $\mathcal{N}(\mathcal{X}) = \bigcup_{\mathbf{i} \in \mathcal{X}} \mathcal{N}(\mathbf{i})$.

2.1 Symbolic encoding of $\mathcal{S}$

In the sequel, we assume that each $\mathcal{S}_k$ is known a priori. In practice, the local state spaces $\mathcal{S}_k$ can actually be generated "on-the-fly" by interleaving symbolic global state-space generation with explicit local state-space generation [12]. We then use the mappings $\psi_k : \mathcal{S}_k \to \{0, 1, \ldots, n_k{-}1\}$, with $n_k = |\mathcal{S}_k|$, identify local state $\mathbf{i}_k$ with its index $i_k = \psi_k(\mathbf{i}_k)$, thus $\mathcal{S}_k$ with $\{0, 1, \ldots, n_k{-}1\}$, and encode any set $\mathcal{X} \subseteq \widehat{\mathcal{S}}$ in a *(quasi-reduced ordered) MDD* over $\widehat{\mathcal{S}}$. Formally, an MDD is a directed, acyclic, edge-labeled multi-graph where:

- Each node p belongs to a *level* $k \in \{K, ..., 1, 0\}$, denoted $p.lvl$.
- There is a single *root* node r at level K.
- Level 0 can only contain the two *terminal* nodes *Zero* and *One*.
- A node p at level $k > 0$ is *nonterminal* with n_k outgoing edges, labeled from 0 to $n_k{-}1$. The edge labeled by i_k points to a node q at level $k{-}1$; we write $p[i_k] = q$.
- Given nodes p and q at level k, if $p[i_k] = q[i_k]$ for all $i_k \in \mathcal{S}_k$, then $p = q$, i.e., there are no *duplicates*.

The MDD encodes a set of states $\mathcal{B}(r)$, defined by the recursive formula:

$$\mathcal{B}(p) \;=\; \begin{cases} \bigcup_{i_k \in \mathcal{S}_k} \{i_k\} \times \mathcal{B}(p[i_k]) & \text{if } \; p.lvl = k > 1 \\ \{i_1 : p[i_1] = One\} & \text{if } \; p.lvl = 1 \end{cases}.$$

For example, box 10 at the bottom of Figure 1 shows an MDD with $K = 3$ and five non-terminal nodes, encoding four global states: (0,0,2), (0,1,1), (0,2,0),

and (1,0,0). In our MDDs, the arcs indices are written in a box in the node from where the arc originates; the terminal nodes *Zero* and *One* and nodes p such that $\mathcal{B}(p) = \emptyset$, as well as any arc pointing to them, are omitted.

Compared with BDDs, MDDs have the disadvantage of resulting in larger and fewer shareable nodes when the variable domains $\mathcal{S}_k$ are very large. On the other hand, MDDs have several advantages. First, many real-world models (e.g., non-safe Petri nets, software) have variable domains with a priori unknown or large upper bounds. These bounds must then be discovered "on the fly" during the symbolic iterations [12,14], and MDDs are preferable to BDDs when using this approach, due to the ease with which MDD nodes and variable domains can be extended. A second advantage, related to the present paper, is that our chaining heuristics applied to the MDD state variables more closely reflect structural information of the model behavior, which is instead spread on multiple levels in a BDD.

2.2 Symbolic encoding of $\mathcal{N}$

For $\mathcal{N}$, we adopt a representation used for Markov chains [3] when the partition into submodels is *Kronecker consistent* [10,11]. Each $\mathcal{N}_e$ is conjunctively decomposed into K local next-state functions $\mathcal{N}_{k,e}$, for $K \geq k \geq 1$, such that in any global state $(i_K, ..., i_1) \in \widehat{\mathcal{S}}$, $\mathcal{N}_e(i_K, ..., i_1) = \mathcal{N}_{K,e}(i_K) \times \cdots \times \mathcal{N}_{1,e}(i_1)$. Using $K \cdot |\mathcal{E}|$ matrices $\mathbf{N}_{k,e} \in \{0,1\}^{n_k \times n_k}$, with $\mathbf{N}_{k,e}[i_k, j_k] = 1 \Leftrightarrow j_k \in \mathcal{N}_{k,e}(i_k)$, we encode $\mathcal{N}_e$ as a Kronecker product: $\mathbf{j} \in \mathcal{N}_e(\mathbf{i}) \Leftrightarrow \bigotimes_{K \geq k \geq 1} \mathbf{N}_{k,e}[i_k, j_k] = 1$, where state $\mathbf{i}$ is interpreted as a *mixed-based* index in $\widehat{\mathcal{S}}$ and $\bigotimes$ denotes Kronecker product of matrices. For standard Petri nets, any partition of the places into K subsets results in a Kronecker-consistent decomposition, and the $\mathbf{N}_{k,e}$ matrices are extremely sparse, each row containing at most one nonzero entry.

For example, the middle of Figure 1 shows the Kronecker encoding of $\mathcal{N}$ according to events (a, b, c, d) and levels (x, y, z), listing only the nonzero entries, for example,

$$\mathbf{N}_{y,b} = \left\{ \begin{array}{c} 1 \rightarrow 0 \\ 2 \rightarrow 1 \end{array} \right\} \quad \text{means} \quad \mathbf{N}_{y,b} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

and $\mathbf{N}_{y,b}[1, 0] = 1$ indicates that if the local state at level y is 1, event b is locally enabled and firing b, if globally possible, moves the local state from 1 to 0.

2.3 Saturation-based iteration strategy

In addition to efficiently representing $\mathcal{N}$, the Kronecker encoding allows us to recognize *event locality* [10,22] and employ the *Saturation* algorithm [11]. Event e is *independent* of level k if $\mathbf{N}_{k,e} = \mathbf{I}$, the identity matrix. Let $Top(e)$ and $Bot(e)$ be the highest and lowest levels for which $\mathbf{N}_{k,e} \neq \mathbf{I}$. An MDD node

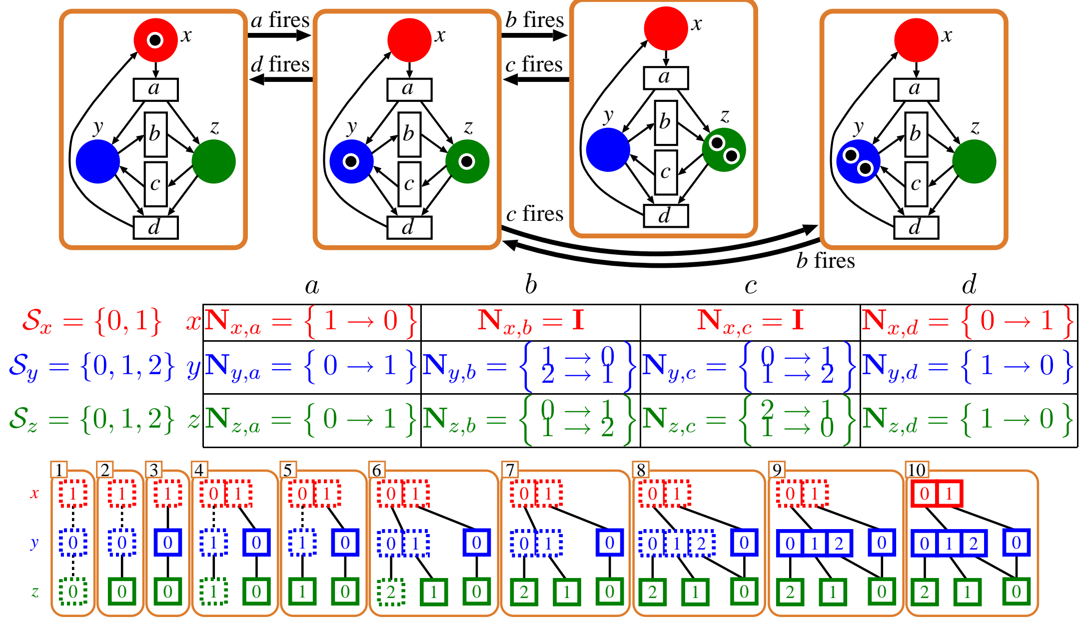

Fig. 1. Reachability graph, $\mathcal{S}_x, \mathcal{S}_y, \mathcal{S}_z$, and $\mathcal{N}$, evolution of the MDD,

p at level k is said to be *saturated* if it is a fixed point with respect to all $\mathcal{N}_e$ such that $Top(e) \leq k$, i.e., $\mathcal{S}_K \times \cdots \times \mathcal{S}_{k+1} \times \mathcal{B}(p) \supseteq \mathcal{N}_{\leq k}(\mathcal{S}_K \times \cdots \times \mathcal{S}_{k+1} \times \mathcal{B}(p))$, where $\mathcal{N}_{\leq k} = \bigcup_{e:Top(e) \leq k} \mathcal{N}_e$. After saturating its descendants, we saturate node p by updating it in place until it also encodes $\mathcal{N}_{k,e} \times \cdots \times \mathcal{N}_{1,e}(\mathcal{B}(p))$, for any event e such that $Top(e) = k$. If this creates new MDD nodes below p, we saturate them immediately, prior to completing the saturation of p.

If we start with the MDD encoding the initial state $\mathbf{s}^{init}$ and saturate its nodes bottom up, at the end the root r will encode $\mathcal{S} = \mathcal{N}^*(\mathbf{s}^{init})$, because $\mathcal{N}^*(\mathbf{s}^{init}) \supseteq \mathcal{B}(r) \supseteq \{\mathbf{s}^{init}\}$, since we only add states through legal event firings, and because $\mathcal{B}(r) \supseteq \mathcal{N}_{\leq K}(\mathcal{B}(r)) = \mathcal{N}(\mathcal{B}(r))$, since r is saturated.

The top of Figure 1 shows the reachability graph of a three-place Petri net. A global state is described by the local state of place x, y, and z, in that order, with local states indexed by the number of tokens in the corresponding place. The bottom of Figure 1 shows the Saturation-based state-space generation of this model (solid MDD nodes are saturated, dashed ones are not).

1 **Initial configuration:** Set up the MDD encoding the global state (1,0,0).
2 **Saturate node $\boxed{0}$ at level z:** No action is required, since there is no event e with $Top(e) = z$. The node is saturated by definition.
3 **Saturate node $\boxed{0}$ at level y:** $Top(b) = Top(c) = y$, but neither b nor c are enabled at both levels y and z. Thus, no firing is possible, and the node is saturated.
4 **Saturate node $\boxed{1}$ at level x:** $Top(a) = x$ and a is enabled for all levels, thus event a must be fired on the node. Since, by firing event a, local state 1 is reachable from 0 for both levels y and z, node $\boxed{1}$ at level y and node $\boxed{1}$ at level z are created (not yet saturated). This also implies that the new global state (0,1,1) is discovered.

5 **Saturate node $\boxed{1}$ at level z:** Again, no action is required as the node is saturated by definition.
6 **Saturate node $\boxed{1}$ at level y:** $Top(b) = y$ and b is enabled for all levels, thus event b must be fired on the node. Since, by firing event b, local state 0 is reached from 1 at level y and local state 2 is reached from 1 at level z, node $\boxed{1}$ at level y is extended to $\boxed{0}\boxed{1}$ and node $\boxed{2}$ at level z is created. This also implies that the new global state (0,0,2) is discovered.
7 **Saturate node $\boxed{2}$ at level z:** Again, no action is required, as the node is saturated by definition.
8 **Saturate node $\boxed{01}$ at level y:** $Top(c) = y$ and c is enabled for all levels, thus event c must be fired on the node. Since, by firing event c, local state 2 is reachable from 1 at level y and local state 0 is reachable from 1 at level z, node $\boxed{01}$ at level y is extended to $\boxed{012}$ and node $\boxed{0}$ at level z, which has been created and saturated previously, is referenced. This also implies that the new global state (0,2,0) is discovered.
9 **Saturate node $\boxed{012}$ at level y:** After exploring all possible firings, the node is saturated.
10 **Saturate node $\boxed{01}$ at level x:** Since no firing can find new global states, the root is saturated.

Saturation consists of many "lightweight" nested "local" fixed-point image computations and is completely different from the traditional breadth-first approach that employs a single "heavyweight" global fixed-point image computation. Results in [8,11,12,13,14] consistently show that Saturation outperforms the breadth-first approach (even if improved by *chaining*) for symbolic state-space generation by several orders of magnitude in both memory and time, making it arguably the most efficient state-space generation algorithm for globally-asynchronous locally-synchronous discrete event systems.

2.4 Saturation NOW

In [5] we described a message-passing algorithm, *Saturation NOW*, that distributes the MDD nodes encoding states over a NOW, to study large models where a single workstation would have to rely on virtual memory to explore the state space. On a NOW with $W \leq K$ workstations numbered from W down to 1, each workstation w has two *neighbors*: one "below", $w-1$ (unless $w = 1$), and one "above", $w+1$ (unless $w = W$). Initially, we evenly allocate the K MDD levels to the W workstations accordingly, by assigning the ownership of levels $\lfloor w \cdot K/W \rfloor$ through $\lfloor (w-1) \cdot K/W \rfloor + 1$ to workstation w. Local variables $mytop_w$ and $mybot_w$ indicate the highest and lowest levels owned by workstation w, respectively.

For distributed state-space generation, each workstation w first generates the Kronecker matrices $\mathbf{N}_{k,e}$ for those events and levels where $\mathbf{N}_{k,e} \neq \mathbf{I}$ and $mytop_w \geq k \geq mybot_w$. Then, the sequential Saturation algorithm begins, except that, when workstation $w > 1$ would normally issue a recursive call to level $mybot_w - 1$, it instead sends workstation $w-1$ a request to perform this

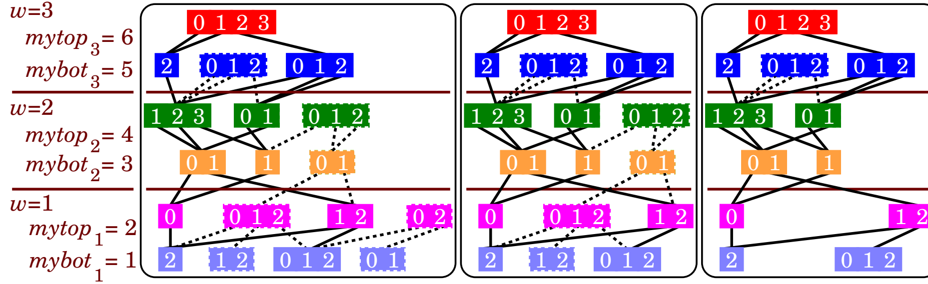

Fig. 2. Garbage collection example

operation, and waits for a reply. A linear organization of the workstations suffices, as each workstation only needs to communicate with its neighbors.

3 Distributed Garbage Collection

The garbage collection in SMART uses a cleanup procedure based on reference counts, where each MDD node has a counter that records the number of the incoming arcs. A cleanup invocation deletes any node whose reference count is zero, and removes its reference from the unique table and the operation caches. Then, a garbage collection call recycles the frees space using level-based recycling pools. When SMART uses the *strict* garbage collection policy, any dereference may proceed recursively down to the bottom level of the decision diagram. This can free up more memory, but it can be more costly in terms of runtime and, in a distributed setting, it also requires message passing between workstations. To avoid a heavy communication overhead, the policy can be relaxed by delaying the garbage collection until the number of dead nodes reaches some given threshold.

Distributed cleanup is then performed by freezing state-space exploration temporarily to deal with the disconnected nodes at runtime. Without the up-to-date reference counts, the distributed cleanup issued at the kth level needs to scan through the outgoing arcs of the nodes at level $k+1$, to determine the referencing information. To overlap the distributed cleanup and referencing information retrieval, it is useful to clean up several consecutive levels at a time in a top-down fashion. Thus, our distributed garbage collection triggers a series of distributed cleanups starting at the kth level to recycle disconnected nodes at any level equal to or lower than k.

The left of Figure 2 shows a runtime snapshot of a six-level decision diagram distributed over three workstations where each workstation manages two levels of the MDD. The dashed boxes and lines indicate disconnected nodes. The middle of Figure 2 shows the decision diagram resulting when the two bottom workstations ($w = 2$ and 1) perform distributed cleanup on the decision diagram shown in the left of Figure 2 starting at level 2. In this case, one node at level 2 and one node at level 1 have been cleaned out. The right of Figure 2 shows the decision diagrams resulting when three workstations per-

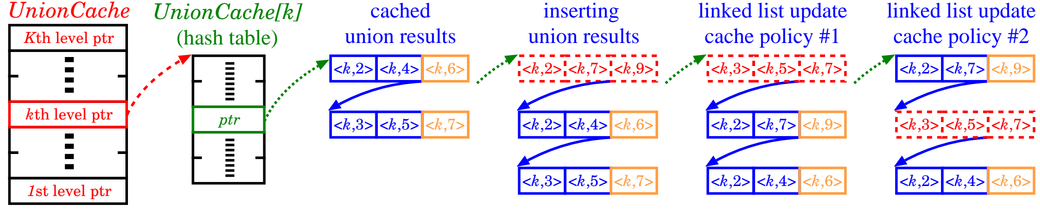

Fig. 3. Union caches

form distributed cleanup on the same decision diagram but starting at level 4 instead. In this case, four nodes have been claned out.

4 Operation cache policies

The main reason why symbolic model checking can outperform the explicit approach is that the *implicit* state-space construction allows nodes to share not only their children, roots of isomorphic decision diagrams, providing memory efficiency, but also the computation corresponding to each of those children, providing time efficiency. To efficiently share the computation during symbolic state-space generation, *hashing* is considered to be one of the most effective way to cache computed data dynamically. However, a good hashing can still create identical hash values for distinct entries. To cope with this hash *collisions*, we use collision-tolerant hash tables: multiple entries having identical hash value are stored in a linked list.

For each MDD level, we use a collision-tolerant hash table for each kind of operation (union, fire) and rehash each of the tables whenever the projected number of collisions becomes too large. Figure 3 shows the data structure that we use to cache the results of the union operation during distributed state-space generation. The fourth column of Figure 3 shows an example of caching a newly computed result. The union of $\langle k,2\rangle\langle k,7\rangle = \langle k,9\rangle$, has the same hash value as the the union of $\langle k,2\rangle\langle k,4\rangle$ and the union of $\langle k,3\rangle\langle k,5\rangle$. In this case, three entries are stored in the linked list for that hash value.

4.1 Bounded-rehashing and bounded-collision caches

We developed a *bounded-rehashing* and *bounded-collision* cache policy to facilitate all cache-related operations, a hybrid technique between the unbounded list hashing and fixed-size lists.

For each hash table, we keep track of *MaxCollision*, the maximal size of any linked list used in the table, indicating that the time to retrieve any entry is $\mathcal{O}(MaxCollision)$. Whenever *MaxCollision* exceeds a given threshold, *MaxAllowedCollisions*, we double the size of the table and rehash all its entries. To prevent esxcessive memory usage for caching computed results, we stop rehashing the table when its size reaches a maximum, *RehashThreshold*, while still enforcing the limit of *MaxAllowedCollisions* on the size of the linked lists, to preserve the data retrieval time. In other words, the maximum number

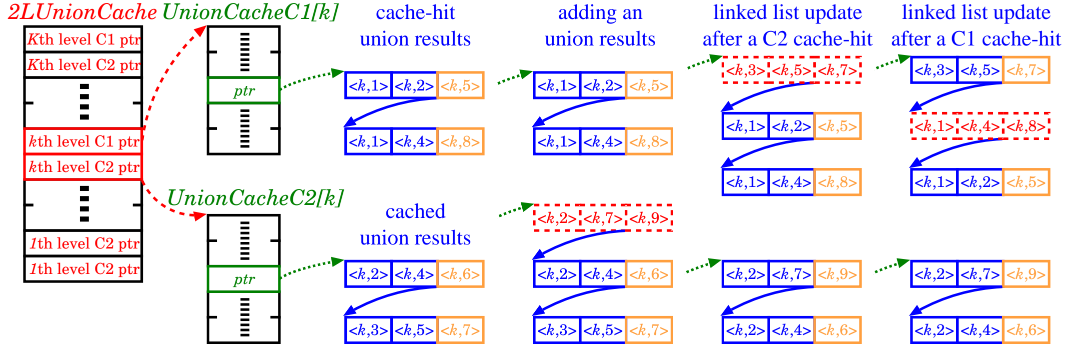

Fig. 4. Two-level union caches

of entries with identical hash value cannot exceed *MaxAllowedCollisions*, no matter whether the size of the hash table has exceeded *RehashThreshold* or not. This way, the time to retrieve a cached entry is $\mathcal{O}(MaxAllowedCollisions)$. An entry is said to be a *cache-hit* if it is retrieved by a hash table lookup.

Once the list is full, the insertion of a new element at the front of the list causes the removal of an element according to the cache replacement policy. To take the potential usefulness of entries into account, we need cache policies that preserve the frequently hit entries, which means keeping them in the first *MaxAllowedCollisions* elements of the corresponding linked list.

4.2 Single-level cache policies

The Last Recently Used (LRU) policy moves every newly accessed element to the front of the corresponding list. Yet, this policy treats every cache-hit entry equally no matter how often each of them has been retrieved. A possibly better strategy is to switch a newly cache-hit entry with the one in front of it. The last two columns of Figure 3 show how a linked list is updated following these two policies after the result of the union of $\langle k,3\rangle\langle k,5\rangle \to \langle k,7\rangle$ is cache-hit. Using either of these two policies, the time to retrieve each cached entry is still $\mathcal{O}(MaxAllowedCollisions)$.

4.3 Two-level cache policies

Since the single-level cache policy mixes up the newly cached entries and the cache-hit entries, it hinders the idea of distinguishing the usefulness of each entry. Also, heavy element insertion might slow down the subsequent list operations corresponding to some key. Thus, we suggest a two-level (C1 and C2) operation cache. The C1 cache is used to store cache-hit entries, while the C2 cache is used to store newly cached entries. A newly computed result is initially cached in C2. When a C2-cached entry is hit, it is moved to the C1 cache. When a C1-cached entry is hit, its position is changed. Figure 4 shows the new structure we use to cache union operations. The three examples shown in the last three columns of Figure 4 are: caching a newly computed result, the union of $\langle k,2\rangle\langle k,7\rangle$ is $\langle k,9\rangle$, and this entry shares a key with the

union of $\langle k,2\rangle\langle k,4\rangle$ and the union of $\langle k,3\rangle\langle k,5\rangle$; the update following an C2 cache hit on the result of the union of $\langle k,3\rangle\langle k,5\rangle$; the update following an C1 cache hit on the result of the union of $\langle k,1\rangle\langle k,4\rangle$.

5 Results

5.1 Experiments for the bounded rehashing scheme

We evaluate the rehashing heuristic and different cache policies using the Saturation algorithm to generate the state space of several parameterized models:

- *Round robin mutex protocol* (**Robin**) [16] models the round robin solution to a mutual exclusion problem where N is the number of processes involved.
- *Flexible manufacturing system* (**FMS**) [22] models a manufacturing system with three machines to process three different types of parts where N is the number of each type of part.
- *Slotted ring network protocol* (**Slot**) [23] models a local area network protocol where N is the number of nodes in the network.
- *Leader election protocol* (**Leader**) [19] models a protocol for designating a unique processor as the leader by sending messages along a unidirectional ring of N processors.
- *Aloha network protocol* (**Aloha**) [9] models a local area network protocol where N is the number of nodes in the network.
- *Kanban manufacturing system* (**Kanban**) [26] models a manufacturing system authorizing production based on the consumption at the downstream stations where N is the admission threshold to each machine.
- *Bounded open queuing network* (**BQ**) [15] models an open queuing network where the capacity of the queue is bounded by N.
- *Knights problem* (**Knight**) models the problem of swapping $(N^2-1)/2$ black knights with $(N^2-1)/2$ white knights placed on an $N \times N$ chessboard with one empty square in the middle, where $N \geq 5$ is odd.
- *Queens puzzle* (**Queen**) models the game of placing N non-attacking queens on an $N \times N$ chessboard.
- *Runway safety monitor* (**RIPS**) [24] models an avionics system monitoring T targets with S speeds on a grid represented as a $X \times Y \times Z$ grid.

Table 1 shows the experimental study performed on a Pentium IV 3GHz workstation with 1GB of RAM. The first two columns show the model names and the corresponding parameters used for this evaluation. The third and fourth columns indicate the initial size (*init*) of hash tables and the maximal size (*max*) allowed for rehashing. The next six columns list the time and memory consumption of three different approaches: fixed-size hashing using *init*, fixed-size hashing using *max*, dynamic hashing using *init* and then allowing to rehash until *max* is reached, denoted *FHI*, *FHM*, and *DH*, respectively. The

Model	param	Cache		Time (sec)			Memory (MB)			Rehash
		init	*max*	*FHI*	*FHM*	*DH*	*FHI*	*FHM*	*DH*	(times)
Robin	600	100	50K	41	34	37	326	555	349	2677
FMS	250	100	100K	174	55	56	103	240	108	62
Slot	150	100	100K	122	97	104	210	324	233	1302
Leader	7	100	100K	123	12	15	147	205	172	684
Aloha	70	100	500K	85	12	16	227	502	255	699
Kanban	60	100	1M	124	16	16	60	175	69	143
BQ	40	100	1M	102	58	60	97	120	106	43
Knight	6	100	1M	160	9	12	69	336	123	370
Queen	12	1000	1M	22	10	11	65	149	72	35
RIPS	1,4,4,4,4	1000	10M	567	113	127	168	854	516	124

Table 1
Experimental results of rehashing.

Model	param	Memory (MB)	Time (sec)		
			DoNothing	*MoveFirst*	*MoveForward*
Robin	600	285	49.74	49.68	49.94
FMS	200	60	67.89	39.62	64.07
Slot	150	229	146.79	146.93	146.43
Leader	7	132	88.19	74.86	83.11
Aloha	70	241	160.22	161.16	158.92
Kanban	60	57	135.91	73.08	128.58
BQ	40	97	78.39	69.78	70.05
Knight	6	73	168.26	130.02	158.82
Queen	12	51	33.48	33.35	33.28
RIPS	1,2,3,3,3	31	42.09	17.08	20.50

Table 2
Experimental results for two-level caching.

last column shows the number of rehashings that have been performed.

In Table 1, we can see that *FHI* is always the most memory efficient approach and *FHM* is always the most time efficient one. However, in all cases, the memory consumption of *DH* can be as low as *FHI*'s, while the runtime of *DH* is very close to *FHM*'s. Note that the tradeoff in performing dynamical rehashing, shown in the last column of Table 1, is the runtime difference between *FHM* and *DH*, which is insignificant. Additionally, the high memory consumption of *DH* implies that excessive unbounded rehashing can be expensive. In conclusion, the experiment shows that rehashing works well in many cases making the bounded rehashing idea more practical.

5.2 Experiments for the two-level cache policies

In Table 2, the first three columns show the model names, the corresponding parameters used and the total memory consumed for the evaluation respec-

tively. The next three columns present the runtime for the two-level cache schemes using three different policies for a cache hit on a C1-cached entry: its position will not be changed (*DoNothing*), its position will be moved to the front of the linked list (*MoveFirst*), or moved forward one position (*MoveForward*). To perform a fair comparison among these cache policies, we use fixed-size unbounded hashing with $init = 100$ for all experiments.

In Table 2, we can see that our two-level cache policies do significantly improve the runtime in some cases, especially the models with a constant or linear number of MDD levels. When there is an improvement, *MoveFirst* always outperforms *MoveForward* and *DoNothing*. In the worst case, neither *MoveFirst* nor *MoveForward* slow down the runtime with respect to *DoNothing*.

In a nutshell, by comparing column 5 in Table 1 with column 4 in Table 2 (for runtime) and column 8 in Table 1 with column 3 in Table 2 (for memory consumption), we observe that the overhead of the two-level cache is not negligible in comparison to the one-level cache. However, for those benchmarks where the cache policies can improve the runtime, this is a good tradeoff.

5.3 Experiments for the message-passing implementation

This experimental study of SMART and SMARTNOW on **Robin** and **Slot** was performed on the Sciclone [1] cluster at the College of William and Mary consisting of many different heterogeneous subclusters. We used the Whirlwind (homogeneous) subcluster containing 64 single-CPU Sun Fire V120 nodes (UltraSPARC IIi+ 650 MHz, 1 GB RAM) connected by Myrinet and running Solaris 9 with LAM/MPI on TCP/IP. Three parameters selected for both models represent the small, medium, and large cases of symbolic state-space generation, where the sequential program requires $\approx$ 100MB, $\approx$ 500MB and $\approx$ 900MB, respectively. For each case, we run SMARTNOW on W (1, 2, 4, 8, 16, 32, or 64) workstations and record the runtime in seconds, with dashed lines in Figure 5 and 6, and the total NOW memory usage in MB, filled and dashed boxes corresponding to the right axes. To compute the actual overhead of the distributed algorithm, we use the memory information reported by the operating system (retrieved from /proc). Also, we use fixed-size hashing to test all cases, even though, in comparison to SMART, running SMARTNOW on a NOW has more memory available for rehashing to accelerate the computation

Figure 5 and 6 show that, when the RAM of a single workstation is sufficient to run the test case, the runtime of SMART is better than that of SMARTNOW on multiple workstations. The message-passing overhead, while not huge, is not negligible either. For the small test cases, the runtime of SMARTNOW is a few times larger than that of SMART. However, the difference diminishes as the model size grows, even much before memory swapping becomes an issue. Significant improvements are noticeable even when comparing SMARTNOW to itself for different values of W. Considering the case of **Robin** $N = 1000$, the optimal number W_{opt} of workstations to use is 4. In the right of Figure 5,

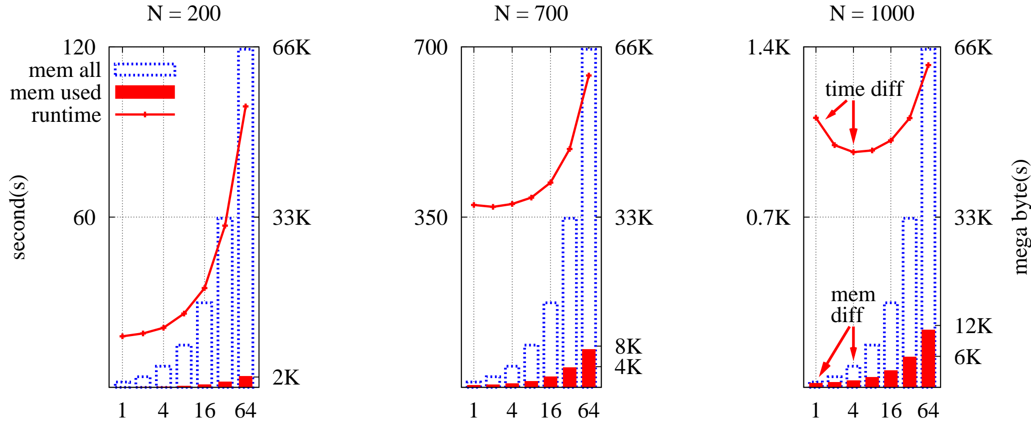

Fig. 5. Experiments on **Robin**

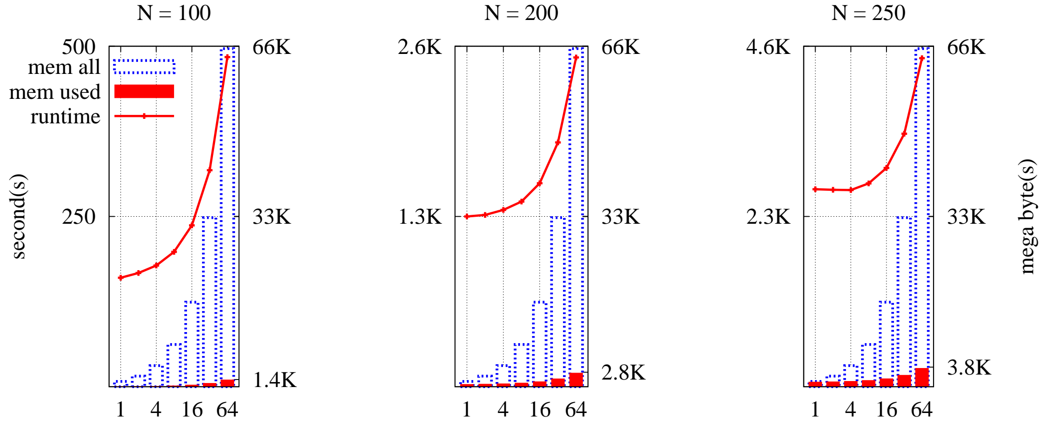

Fig. 6. Experiments on **Slot**

the runtime difference and the memory consumption difference is indicated by solid arrows. SMART rarely triggers memory swapping when 862MB of memory is used on a single machine and SMARTNOW does not use virtual memory at all when 1026MB of NOW memory is used over four network-connected machines.

In addition, while such W_{opt} cannot be known a priori, the results clearly show that using too many workstations affects the runtime only by a small factor, while using too few, or just one, results in very large penalties, if the algorithm completes at all. Considering the case of **Robin** $N = 1000$ again, the runtime penalty of using $W = 1 < W_{opt}$ is higher than that of using $W = 32$. Furthermore, even though the runtime penalty of using $W > W_{opt}$ is not trivial, the difference between the filled boxes and the dashed boxes indicates the large amount of NOW memory left over, which could be used for model checking or some *anticipation* heuristic [6,7] to accelerate the computation.

Robin and **Slot** are two typical problems for our distributed algorithm.

However, the scalability of our current distributed Saturation implementation is restricted by the number of partitions of the input model: the number W of workstations cannot exceed the number K of MDD levels. In the other words, the ownership of each MDD level is exclusive among workstations, although this ownership can be transferred between workstations. Thus, these two models are selected simply because they have more than 64 MDD levels. This limitation can be overcome by allowing multiple workstations to manage the same MDD level, but then additional communication overhead is required to maintain canonicity of the MDD nodes over the workstations. Last but not least, the experiments for the two-level cache policies show an improvement in comparison to our original implementation.

6 Conclusion

We designed and implemented a new version of the distributed symbolic state-space generator, SMART_NOW, whose level-based node allocation scheme achieves excellent memory distribution and scalability over NOWs. Thanks to the ever increasing network speed, our approach effectively provides the large amounts of memory needed when studying large systems, although it offers no theoretical speed-up. Also, the cache heuristics speed up our algorithm and make the performance of the distributed approach more convincing.

References

[1] Sciclone cluster project. http://www.compsci.wm.edu/sciclone/.

[2] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comp.*, 35(8):677–691, August 1986.

[3] P. Buchholz, G. Ciardo, S. Donatelli, and P. Kemper. Complexity of memory-efficient Kronecker operations with applications to the solution of Markov models. *INFORMS J. Comp.*, 12(3):203–222, 2000.

[4] J. Burch, E. Clarke, and D. Long. Symbolic model checking with partitioned transition relations. *Intl. Conf. on VLSI*, p. 49–58, Edinburgh, Scotland, August 1991. IFIP Transactions.

[5] M. Chung and G. Ciardo. Saturation NOW. In *Proc. QEST*, p. 272–281, Enschede, The Netherlands, September 2004. IEEE Press.

[6] M. Chung and G. Ciardo. A pattern recognition approach for speculative firing prediction in distributed saturation state-space generation. *PDMC 2005*, ENTCS, p. 65–79, Lisbon, Portugal, July 2005.

[7] M. Chung and G. Ciardo. A dynamic firing speculation to speedup distributed symbolic state-space generation. In *Proc. IPDPS*, Rhodes, Greece, April 2006.

[8] M. Chung, G. Ciardo, and J. Yu. A fine-grained fullness-guided chaining heuristic for symbolic reachability analysis. In *Proc. Automated Technology for Verification and Analysis (ATVA)*, LNCS, Beijing, China, October 2006.

[9] G. Ciardo and Y. Lan. Faster discrete-event simulation through structural caching. In *Proc. PMCCS*, p. 11–14, Monticello, IL, USA, September 2003.

[10] G. Ciardo, G. Lüttgen, and R. Siminiceanu. Efficient symbolic state-space construction for asynchronous systems. *Proc. 21th Int. Conf. on Applications and Theory of Petri Nets*, LNCS 1825, p. 103–122, Aarhus, June 2000.

[11] G. Ciardo, G. Lüttgen, and R. Siminiceanu. Saturation: An efficient iteration strategy for symbolic state space generation. *Proc. TACAS 2001*, LNCS 2031, p. 328–342, Genova, Italy, April 2001.

[12] G. Ciardo, R. Marmorstein, and R. Siminiceanu. Saturation unbound. *Proc. TACAS 2003*, LNCS 2619, p. 379–393, Warsaw, Poland, April 2003.

[13] G. Ciardo and R. Siminiceanu. Structural symbolic CTL model checking of asynchronous systems. *Computer Aided Verification (CAV'03)*, LNCS 2725, p. 40–53, Boulder, CO, USA, July 2003.

[14] G. Ciardo and J. Yu. Saturation-based symbolic reachability analysis using conjunctive and disjunctive partitioning. *Proc. CHARME*, LNCS 3725, p. 146–161, Saarbrücken, Germany, October 2005.

[15] P. Fernandes, B. Plateau, and W. Stewart. Efficient descriptor-vector multiplication in stochastic automata networks. *J. ACM*, 45(3):381–414, 1998.

[16] S. Graf, B. Steffen, and G. Lüttgen. Compositional minimisation of finite state systems using interface specifications. *Formal Aspects of Comp.*, 8(5):607–616.

[17] O. Grumberg, T. Heyman, N. Ifergan, and A. Schuster. Achieving speedups in distributed symbolic reachability analysis through asynchronous computation. In *Proc. CHARME*, LNCS 3725, p. 129–145, Saarbrücken, Germany, Oct. 2005.

[18] T. Heyman, D. Geist, O. Grumberg, and A. Schuster. Achieving scalability in parallel reachability analysis of very large circuits. *Computer-Aided Verification (CAV), LNCS* vol 1855, p. 20–35, Chicago, IL, USA, 2000.

[19] A. Itai and M. Rodeh. Symmetry breaking in distributed networks. In *22th Symp. on Foundations of Computer Science*, p. 150–158, October 1981.

[20] T. Kam, T. Villa, R. Brayton, and A.S. Vincentelli. Multi-valued decision diagrams: theory and applications. *Multiple-Valued Logic*, 4(1–2):9–62, 1998.

[21] K. Milvang-Jensen and A. Hu. BDDNOW: A parallel BDD package. *Formal Methods in Computer-Aided Design*, LNCS 1522, p. 501–507, 1998.

[22] A. Miner and G. Ciardo. Efficient reachability set generation and storage using decision diagrams. *Proc. Int. Conf. on Applications and Theory of Petri Nets*, LNCS 1639, p. 6–25, Williamsburg, VA, USA, June 1999.

[23] E. Pastor, O. Roig, J. Cortadella, and R. Badia. Petri net analysis using boolean manipulation. *Proc. 15th Int. Conf. on Applications and Theory of Petri Nets*, LNCS 815, p. 416–435, Zaragoza, Spain, June 1994.

[24] R. Siminiceanu and G. Ciardo. Formal verification of the NASA Runway Safety Monitor. *Proc. AVoCS'04*, ENTCS, London, UK, September 2004.

[25] A. Stornetta. Implementation of an Efficient Parallel BDD Package. M.S. thesis, University of California, Santa Barbara, 1995.

[26] M. Tilgner, Y. Takahashi, and G. Ciardo. SNS 1.0: Synchronized Network Solver. *Int. Workshop on Manufacturing and Petri Nets*, p. 215–234, Osaka, Japan, June 1996.