A Pattern Recognition Approach for Speculative Firing Prediction in Distributed Saturation State-Space Generation

Ming-Ying Chung [1] and Gianfranco Ciardo [2]

University of California, Riverside, CA 92521

Abstract

The *saturation* strategy for symbolic state-space generation is particularly effective for globally-asynchronous locally-synchronous systems. A distributed version of saturation, *SaturationNOW*, uses the overall memory available on a network of workstations to effectively spread the memory load, but its execution is essentially sequential. To achieve true parallelism, we explore a *speculative firing prediction*, where idle workstations work on predicted future event firing requests. A naïve approach where all possible firings may be explored a priori, given enough idle time, can result in excessive memory requirements. Thus, we introduce a history-based approach for firing prediction that recognizes firing patterns and explores only firings conforming to these patterns. Experiments show that our heuristic improves the runtime and has a small memory overhead.

Keywords: state-space generation, decision diagrams, distributed systems, parallel and distributed computing, speculative computing, pattern recognition

1 Introduction

Formal verification techniques such as model checking [10] are widely used in industry for quality assurance, since they can be used to detect design errors early in the lifecycle. An essential step is an exhaustive, and very memory-intensive, state-space generation. Even though symbolic encodings like binary decision diagrams (BDDs) [2] help cope with the *state-space explosion*, the analysis of complex systems may still resort to the use of virtual memory.

Much research has then focused on parallel and distributed computing for this application. [1,20,25] use a network of workstations (NOW) for *explicit*

⋆ Work supported in part by the National Science Foundation (NSF) under Grants No. 0219745 and No. 0203971.

[1] Email:chung@cs.ucr.edu

[2] Email:ciardo@cs.ucr.edu

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

state-space exploration or model checking. [16] parallelizes BDD manipulation on a shared memory multiprocessor, while [21] uses distributed shared memory. [27] parallelizes BDD construction by sharing image computation among processors during Shannon expansion on shared and distributed shared memory platforms. [11] finds parallelism in breadth-first BDD traversals. [13,17,26] parallelize BDD manipulations by slicing image computation onto a NOW where a master workstation balances the memory load.

In [4], we presented a distributed version of the saturation algorithm [6], called *SaturationNOW*, to perform symbolic state-space generation on a NOW, where execution is strictly sequential but utilizes the overall NOW memory. As in [23], a *level-based horizontal "slicing"* scheme is employed to allocate decision diagram nodes to workstations, so that no additional node or work is created. In addition, we presented a heuristic that dynamically balances the memory load to help cope with the changing peak memory requirement of each workstation. However, the horizontal slicing scheme has two drawbacks. First, while it can evenly distribute the decision diagram with minimal time and space overhead, it does not facilitate parallelism (it corresponds to a sequentialization of the workstations, where most computations require a workstation to cooperate with its neighbors). Second, since a set of contiguous decision diagram levels is assigned to each workstation, models with few decision diagram levels impose a limit on the scalability of the approach. While assigning a single level to multiple workstations solves this problem, the cost of additional synchronizations would eliminate the major advantage of our horizontal slicing scheme.

In this paper, we tackle the first drawback, i.e., we improve the runtime of SaturationNOW, through the idea of using idle workstation time to speculatively fire events on decision diagram nodes, even if some of these event firings may never be needed. In a naïve approach, unrestrained speculation may cause an excessive increase in the memory consumption, to the point of being counter-productive. However, a history-based approach to predict which events should be fired based on past firing patterns is instead effective at reducing the runtime with only a small memory overhead.

Our paper is organized as follows. Sect. 2 gives background on reachability analysis, decision diagrams, Kronecker encoding, and saturation. Sect. 3 details our naïve and pattern recognition approaches to speculative firing prediction. Sect. 4 shows experimental results. Sect. 5 briefly survey related work, and Sect. 6 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. Since we target globally-asynchronous systems, we decompose $\mathcal{N}$ into a

disjunction of next-state functions [15]: $\mathcal{N}(\mathbf{i}) = \bigcup_{e\in\mathcal{E}} \mathcal{N}_e(\mathbf{i})$, where $\mathcal{E}$ is a finite set of *events* and $\mathcal{N}_e$ is the next-state function associated with event e.

The *reachable state space* $\mathcal{S} \subseteq \widehat{\mathcal{S}}$ is the smallest set containing $\mathbf{s}^{init}$ and closed with respect to $\mathcal{N}$: $\mathcal{S} = \{\mathbf{s}^{init}\} \cup \mathcal{N}(\mathbf{s}^{init}) \cup \mathcal{N}(\mathcal{N}(\mathbf{s}^{init})) \cup \cdots = \mathcal{N}^*(\mathbf{s}^{init})$, where "*" denotes reflexive and transitive closure and $\mathcal{N}(\mathcal{X}) = \bigcup_{\mathbf{i}\in\mathcal{X}} \mathcal{N}(\mathbf{i})$.

We assume a model composed of K *submodels*. Thus, a (*global*) state 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 $\widehat{\mathcal{S}} = \mathcal{S}_K \times \cdots \times \mathcal{S}_1$, 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.

2.1 Symbolic encoding of the state space $\mathcal{S}$ and next-state function $\mathcal{N}$

While not a requirement (the local state spaces $\mathcal{S}_k$ can be generated "on-the-fly" by interleaving symbolic global state-space generation with explicit local state-space generation [7]), we assume that each $\mathcal{S}_k$ is known a priori. We then use the mappings $\psi_k : \mathcal{S}_k \to \{0, 1, ..., n_k{-}1\}$, 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 $\{0, 1, ..., n_k{-}1\}$, and encode any set $\mathcal{X} \subseteq \widehat{\mathcal{S}}$ in a *quasi-reduced ordered multiway decision diagram* (MDD) over $\widehat{\mathcal{S}}$. Formally, an MDD is a directed acyclic edge-labeled multi-graph where:

- Each node p belongs to a *level* 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 $k > 0$ has 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 $p = q$.

The MDD encodes a set of states $\mathcal{B}(r)$, defined by the recursive formula: $\mathcal{B}(p) = \bigcup_{i_k\in\mathcal{S}_k} \{i_k\} \times \mathcal{B}(p[i_k])$ if $p.lvl = k > 1$, $\mathcal{B}(p) = \{i_1 : p[i_1] = One\}$ if $p.lvl = 1$.

To adopt a Kronecker representation of $\mathcal{N}$ inspired by work on Markov chains [3], we assume a *Kronecker consistent* model [5,6] where $\mathcal{N}_e$ is conjunctively decomposed into K local next-state functions $\mathcal{N}_{k,e}$, for $K \geq k \geq 1$, satisfying $\forall (i_K, ..., i_1) \in \widehat{\mathcal{S}}, \mathcal{N}_e(i_K, ..., i_1) = \mathcal{N}_{K,e}(i_K) \times \cdots \times \mathcal{N}_{1,e}(i_1)$. By defining $K \cdot |\mathcal{E}|$ 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 (boolean) 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 a state $\mathbf{i}$ is interpreted as a *mixed-based* index in $\widehat{\mathcal{S}}$ and "$\bigotimes$" indicates the Kronecker product of matrices. Note that the $\mathbf{N}_{k,e}$ matrices are extremely sparse: for standard Petri nets, each row contains at most one nonzero entry.

2.2 Saturation-based iteration strategy

In addition to efficiently representing $\mathcal{N}$, the Kronecker encoding allows us to recognize and exploit *event locality* [5] and employ *saturation* [6]. We say that event e is *independent* of level k if $\mathbf{N}_{k,e} = \mathbf{I}$, the identity matrix. Let $Top(e)$

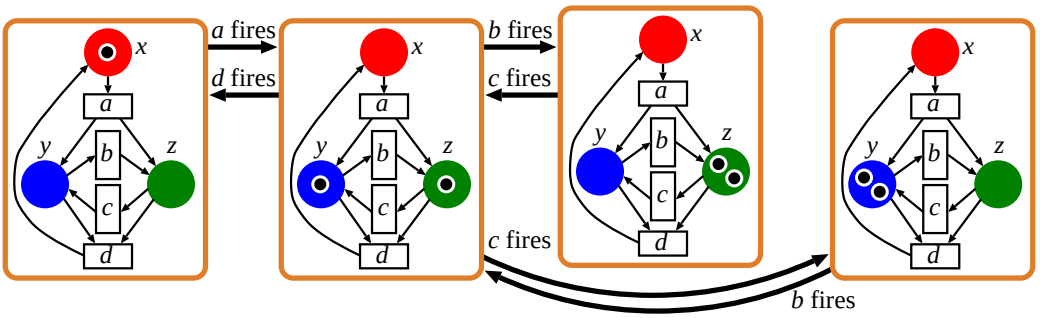

Fig. 1. The reachability graph.

and $Bot(e)$ denote the highest and lowest levels for which $\mathbf{N}_{k,e} \neq \mathbf{I}$. A node 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., $\mathcal{B}(p) = \mathcal{B}(p) \cup \mathcal{N}_{\leq k}(\mathcal{B}(p))$, where $\mathcal{N}_{\leq k} = \bigcup_{e:Top(e)\leq k} \mathcal{N}_e$. To saturate node p once all its descendants have been saturated, we *update it in place* so that it encodes also any state in $\mathcal{N}_{k,e} \times \cdots \times \mathcal{N}_{1,e}(\mathcal{B}(p))$, for any event e such that $Top(e) = k$. This can create new nodes at levels below k, which are saturated immediately, prior to completing the saturation of p.

If we start with the MDD that encodes the initial state $\mathbf{s}^{init}$ and saturate its nodes bottom up, the root r will encode $\mathcal{S} = \mathcal{N}^*(\mathbf{s}^{init})$ at the end, because: (1) $\mathcal{N}^*(\mathbf{s}^{init}) \supseteq \mathcal{B}(r) \supseteq \{\mathbf{s}^{init}\}$, since we only add states, and only through legal event firings, and (2) $\mathcal{B}(r) \supseteq \mathcal{N}_{\leq K}(\mathcal{B}(r)) = \mathcal{N}(\mathcal{B}(r))$, since r is saturated.

In other words, saturation consists of many "lightweight" nested "local" fixed-point image computations, and is completely different from the traditional breath-first approach employing a single "heavyweight" global fixed-point image computation. Results in [6,7,8] consistently show that saturation greatly outperforms breath-first symbolic exploration 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. Thus, it makes sense to attempt its parallelization, while parallelizing the less efficient breadth-first approaches would not offset the enormous speedups and memory reductions of saturation.

2.3 An example of saturation

The reachability graph of a three-place Petri net is shown in Fig. 1. Each global state is described by the local states for place x, y, and z, in that order, and we index local states by the number of tokens in the corresponding place. The reachability graph shows that three global states, (0,1,1), (0,0,2), and (0,2,0), are reachable from the initial state (1,0,0). The Kronecker description of the next-state function is shown in Fig. 2.

For instance, the matrix $\mathbf{N}_{y,b}$ of the Kronecker description indicates that firing event b will decrease the number of tokens in place y, either from 2 to 1 or from 1 to 0. Then, the saturation-based state-space generation on this model can be performed as follow (see Fig. 3).

	a	b	c	d
x	$1 \to 0$	**I**	**I**	$0 \to 1$
y	$0 \to 1$	$1 \to 0$ $2 \to 1$	$0 \to 1$ $1 \to 2$	$1 \to 0$
z	$0 \to 1$	$0 \to 1$ $1 \to 2$	$2 \to 1$ $1 \to 0$	$1 \to 0$

Fig. 2. Kronecker description of the next-state function $\mathcal{N}$.

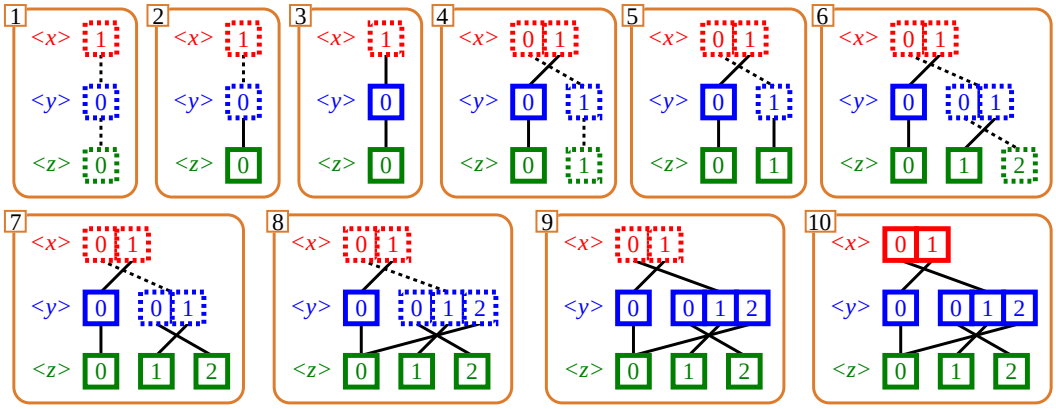

Fig. 3. Saturation example (solid nodes are saturated, dashed nodes are not).

1 **Initial configuration:** Setup the initial global state (1,0,0).

2 **Saturate node** $\boxed{0}$ **at level** z**:** No firing needs to be done, since there is no event with $Top(event) = z$. 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, Therefore, no firing needs to be done, and the node is thus saturated.

4 **Saturate node** $\boxed{0}$ **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, two nodes, $\boxed{1}$ at level y and node $\boxed{1}$ at level z, are created (not yet saturated), This also implies that a new global state, (0,1,1), is discovered.

5 **Saturate node** $\boxed{1}$ **at level** z**:** No firing needs to be done, since there is no event with $Top(event) = z$. Again, 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 $\boxed{01}$ and node $\boxed{2}$ at level z is created. This also implies that a new global state, (0,0,2), is discovered.

7 **Saturate node** $\boxed{2}$ **at level** z**:** No firing needs to be done, since there is

no event with $Top(event) = z$. Again, 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 a new global state, (0,2,0), is discovered.

9 **Saturate node** $\boxed{012}$ **at level** y**:** Since all possible event firings have been done, the node is saturated.

10 **Saturate node** $\boxed{01}$ **at level** x**:** Since no event firing can find new global states, the root node is then saturated.

2.4 Distributed version of saturation

[4] presents *SaturationNOW*, a message-passing algorithm that distributes the state space on a NOW to study large models where a single workstation would have to rely on virtual memory. 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. In each workstation w, local variables $mytop_w$ and $mybot_w$ indicate the highest- and lowest-numbered levels it owns, respectively ($mytop_W = K$, $mybot_1 = 1$ and $mytop_w \geq mybot_w$ for any w). We stress that, in this distributed saturation algorithm, we use a cluster to increase the amount of available memory, not to achieve parallelism in the computation.

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 $mytop_w \geq k \geq mybot_w$, without any synchronization. This is a simplification made possible by the fact that these matrices require little space and can be generated in isolation. Then, the sequential saturation algorithm begins, except that, when workstation $w > 1$ would normally issue a recursive call to level $mybot_w - 1$, it must instead send a request to perform this operation in workstation $w-1$ and wait for a reply. The linear organization of the workstations suffices, since each workstation only needs to communicate with its neighbors.

To cope with dynamic memory requirements, [4] uses a nested approach to reassign MDD levels, i.e., changing the $mybot_w$ and $mytop_{w-1}$ of two neighbors. Since memory load balancing requests can propagate, each workstation can effectively rely on the overall NOW memory, not just that in its neighbors, without the need for global synchronization or broadcasting. With our horizontal slicing scheme, *even an optimal static allocation of levels to workstations could still be inferior to a good, but sub-optimal, dynamic approach.* This is because, the number of nodes at a given MDD level usually increases and decreases dramatically during execution. Workstation w might be using

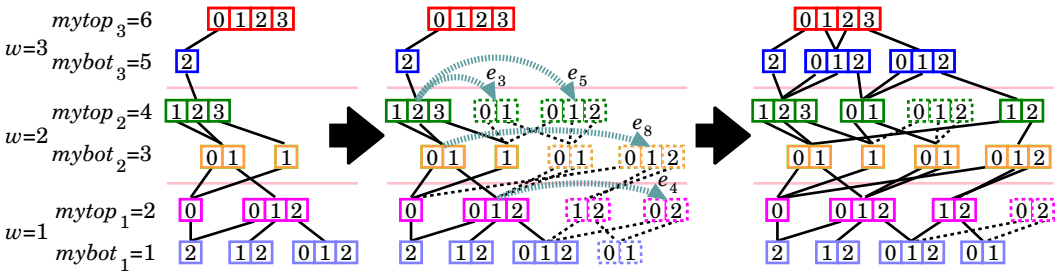

Fig. 4. Firing anticipation.

much less memory than w' at some point in time, while the reverse might occur later on. By dynamically reallocating levels between the two, such dynamic peak requirements can be better accommodated. Of course, this reallocation does not affect canonicity, since it preserves the MDD structure.

3 Speculative firing prediction

The distributed approach of [4] effectively partitions the memory load over the workstations, but it is strictly sequential. We now explore the idea of an idle workstation firing events e with $Top(e) > k$ on saturated nodes p at level k *a priori*, in the hope to reduce the time required to saturate nodes above p.

As explained in Sect. 2, an MDD node p at level k is saturated if any event e with $Top(e) = k$ has been fired exhaustively on p. However, events e with $Top(e) = l > k \geq Bot(e)$ will still need to be fired on p, if there is a path $(i_l, ..., i_{k+1})$ from a node q at level l to p, such that e is "locally enabled", i.e., $\mathcal{N}_{l,e}(i_l) \neq \emptyset, ..., \mathcal{N}_{k+1,e}(i_{k+1}) \neq \emptyset$. To accelerate the time required to saturate such hypothetical node q, our speculative prediction creates the (possibly disconnected) MDD node p' corresponding to the saturation of the result of firing e on p, and caches the result. Later on, any firing of e on p will immediately return the result p' found in the cache. Fig. 4 shows speculative firing prediction at work. In the middle, workstations 2 and 1 have predicted and computed firings for e_3 and e_5 at level 4, e_8 at level 3, and e_4 at level 2 (hence the disconnected "dashed" nodes). On the right, the nodes resulting from firing e_3 or e_8 are now connected, as they were actually needed and found in the cache: speculative prediction was effective in this case.

We stress that the MDD remains canonical, although with additional disconnected nodes. Also, even if workstation w might know a priori that event e satisfies $Top(e) > mytop_w = k \geq Bot(e) \geq mybot_w$, firing e on node p at level k can nevertheless require computation in workstation $w-1$ below, since the result p' must be saturated, causing work to propagate at levels below unless the cache can avoid it. In other words, as it is not known in advance whether the saturation of an event firing can be computed locally, consecutive idle workstations might need to perform speculative event firing together.

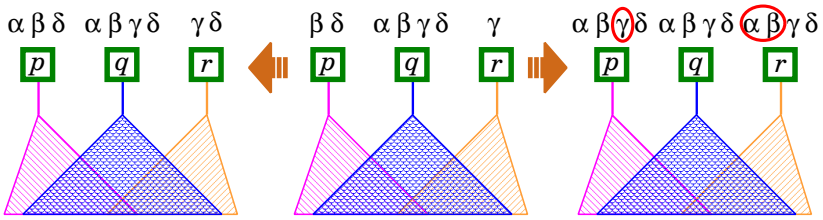

Fig. 5. History-based approach of firing prediction.

3.1 History-based approaches to speculative firing prediction

Since we do not know a priori whether event e will be fired on a node p at level k during state-space generation, the most naïve speculative firing prediction lets idle workstations exhaustively compute *all* possible firings starting "above" each node p of the MDD for $\mathcal{S}$, i.e., $\mathcal{E}_{all}(p) = \{e : Top(e) > k \geq Bot(e)\}$.

Obviously, this is effective only when $|\mathcal{E}|$ is small with respect to K, since, then, exhausting all possible firings over few events is relatively inexpensive in terms of time and space. However, for most models, this approach introduces too many nodes that never become connected to the state-space MDD.

We now motivate a more informed prediction based on firing *patterns*. For each node p at level k, let $\mathcal{E}_{patt}(p)$ be the set of events e that will be fired on p after p has been saturated, thus, $Top(e) > k$ and $\mathcal{E}_{patt}(p) \subseteq \mathcal{E}_{all}(p)$. We can then partition the nodes at level k according to their patterns, i.e., nodes p and q are in the same class if and only if $\mathcal{E}_{patt}(p) = \mathcal{E}_{patt}(q)$. Unfortunately, $\mathcal{E}_{patt}(p)$ is only known *a posteriori*, but it should be observed that most models exhibit clear firing patterns during saturation, i.e., most classes contain many nodes and most patterns contain several events.

Our goal is to *predict* the pattern of a given node p based only on the *history* of the events fired on p so far, $\mathcal{E}_{hist}(p) \subseteq \mathcal{E}_{patt}(p)$. The key idea is that, if $\emptyset \subset \mathcal{E}_{hist}(p) \subset \mathcal{E}_{hist}(q)$, we can speculate that the events in $\mathcal{E}_{hist}(q) \setminus \mathcal{E}_{hist}(p)$ will eventually need to be fired on p as well, i.e., that $\mathcal{E}_{patt}(p) = \mathcal{E}_{patt}(q)$ at the end. Fig. 5 shows an example where p, q, and r are saturated nodes at the same MDD level. The middle of Fig. 5 shows the current event firing history of these nodes at some point during runtime: $\mathcal{E}_{hist}(p) = \{\beta, \delta\}$, $\mathcal{E}_{hist}(q) = \{\alpha, \beta, \gamma, \delta\}$, and $\mathcal{E}_{hist}(r) = \{\gamma\}$. The left of Fig. 5 shows the actual event firing history of these nodes after the state space is generated, i.e., their true patterns: $\mathcal{E}_{patt}(p) = \{\alpha, \beta, \delta\}$, $\mathcal{E}_{patt}(q) = \{\alpha, \beta, \gamma, \delta\}$, and $\mathcal{E}_{patt}(r) = \{\gamma, \delta\}$. Applying our history-based approach, instead, will result in the firings on the right of Fig. 5: since $\mathcal{E}_{hist}(p) \subset \mathcal{E}_{hist}(q)$ and $\mathcal{E}_{hist}(r) \subset \mathcal{E}_{hist}(q)$, the workstation owning this MDD level will fire β and γ on p and α, β, and δ on q in advance, if it is idle. Thus, the useless firings of γ on p and of α and β on q will be speculatively computed (these are highlighted with circles in Fig. 5).

Of course, we do not want to be too aggressive in our prediction. We might have that $\mathcal{E}_{hist}(p) \subset \mathcal{E}_{hist}(q)$ for several different nodes q whose histories have few common elements in addition to $\mathcal{E}_{hist}(p)$. If, for each of these nodes q, we fire each e in $\mathcal{E}_{hist}(q) \setminus \mathcal{E}_{hist}(p)$ on p, many of these predicted firings may be

```
FirePredict (Requests : stack, Class : array)
 while Requests ≠ ∅ do
  (e, p) ← Pop(Requests);                          • for each event firing has been done
  Enqueue(e, E_hist(p));                           • records event firing history
  q ← Class_p.lvl[e];                              • the representative node for e
  if E_hist(p) ⊃ E_hist(q) then                    • E_hist(p) is a better pattern
   Class_p.lvl[e] ← p;                             • p becomes the new representative
   foreach e' ∈ E_hist(p) \ E_hist(q) do
    fire e' on q and cache the result;
  else if E_hist(p) ⊂ E_hist(q) then               • predicts event firings with the pattern
   foreach e' ∈ E_hist(q) \ E_hist(p) do
    fire e' on p and cache the result;
```

Fig. 6. Firing prediction algorithm.

useless, i.e., they may not be actually requested because $e \notin \mathcal{E}_{patt}(p)$. On the other hand, any prediction based on history is guaranteed to be *useful* in the rare case where the patterns of the nodes at level k are *disjoint*: i.e., if, for any two nodes p and q, either $\mathcal{E}_{patt}(p) = \mathcal{E}_{patt}(q)$ or $\mathcal{E}_{patt}(p) \cap \mathcal{E}_{patt}(q) = \emptyset$.

3.2 An efficient implementation of our history-based approach

In addition to being useful, our heuristic also needs to be inexpensive in terms of memory and time overhead. Our technique, then, uses only a subset of the history and an efficient array-based method for prediction requiring $O(1)$ time per lookup and $O(K \cdot |\mathcal{E}|)$ memory overall. Each workstation w:

- stores only the c (e.g., 10) most recent elements of $\mathcal{E}_{hist}$ for its nodes.
- maintains a list $Requests_w$ containing satisfied firing request (e, p).
- uses an array $Class_k$ of size $\{e : Top(e) > k \geq Bot(e)\}$ for each level k of the MDD it owns. An element $Class_k[e]$ is a pointer to a node, initially null.

Normally, workstation w is in *saturation* mode: it computes the result of firing requests (e, p) with $Top(e) > p.lvl = k$, and records (e, p) in $Requests_w$. When w becomes idle, it turns to *prediction* mode: it removes an element (e, p) from $Requests_w$, adds e to the (truncated) history $\mathcal{E}_{hist}(p)$, and examines $Class_k(e)$. If $Class_k(e) =$ null, we set $Class_k(e)$ to p; if $Class_k(e) = q$ and $\mathcal{E}_{hist}(p) \supset \mathcal{E}_{hist}(q)$, we set $Class_k(e)$ to p, and we speculatively fire the events in $\mathcal{E}_{hist}(p) \setminus \mathcal{E}_{hist}(q)$ on q; if $Class_k(e) = q$ and $\mathcal{E}_{hist}(p) \subset \mathcal{E}_{hist}(q)$, we leave $Class_k(e)$ unchanged and we speculatively fire the events in $\mathcal{E}_{hist}(q) \setminus \mathcal{E}_{hist}(p)$ on p (see Fig. 6). To minimize "real work" latency, a firing request from workstation $w + 1$ switches w back to *saturation* mode, aborting any speculative firing under way.

In other words, we use $Class_k(e)$ to predict which node has the "best history" among all nodes on which e has been fired so far, and use the history of this node as our speculative firing guide for any node on which e is subsequently fired. This heuristic may suffer from "inversions": if $Class_k(e) = q$

and $\mathcal{E}_{hist}(p) \supset \mathcal{E}_{hist}(q)$ when e is fired on p, we set $Class_k(e)$ to p; later on, further firings of q may result in $\mathcal{E}_{hist}(p) \subset \mathcal{E}_{hist}(q)$, but $Class_k(e)$ will never be set to q, since the firing of e on q is in the cache already and will never be requested again. Nevertheless, this heuristic has minimal bookkeeping requirements, especially in saturation mode, and fast lookup times; its memory requirements are also low, since, the more workstations are idle, the faster $Requests_w$ is emptied, while $Class_k$ and the truncated history use less memory than the nodes of the MDD in practice. Sect. 4 shows that this heuristic can reduce runtime on large models. Finally, we observe that our approach can be relaxed: if we fire e on p, $Class_k(e) = q$, and $\mathcal{E}_{hist}(q) \cup \{f\} \supset \mathcal{E}_{hist}(p)$ but $f \notin \mathcal{E}_{hist}(q)$, we can still decide to speculatively fire $\mathcal{E}_{hist}(q) \setminus \mathcal{E}_{hist}(p)$ on p; however, this aggressive approach often results in too many useless firings.

4 Experimental results

Our approach is implemented in SmArTNow [4], the MPICH-based distributed version of our tool SmArT [9]. We evaluate its performance by using saturation to generate the state space of following models.

- *Slotted ring network protocol* [22] models a protocol for local area networks where N is the number of nodes within the networks ($K = N$, $|\mathcal{S}_k| = 10$ for all k, $|\mathcal{E}| = 3N$).
- *Flexible manufacturing system* [18], models a manufacturing system with three machines to process three different types of parts where N is the number of each type of parts ($K = 19$, $|\mathcal{S}_k| = N + 1$ for all k except $|\mathcal{S}_{17}| = 4$, $|\mathcal{S}_{12}| = 3$, and $|\mathcal{S}_7| = 2$, $|\mathcal{E}| = 20$).
- *Round robin mutex protocol* [12] models the round robin version of a mutual exclusion algorithm where N is the number of processors involved ($K = N + 1$, $|\mathcal{S}_k| = 10$ for all k except $|\mathcal{S}_1| = N + 1$, $|\mathcal{E}| = 5N$).
- *Runway safety monitor* [24] models an avionics system to monitor T targets with S speeds on a $X \times Y \times Z$ runway ($K = 5(T+1)$, $|\mathcal{S}_{5+5i}| = 3$, $|\mathcal{S}_{4+5i}| = 14$, $|\mathcal{S}_{3+5i}| = 1+X(10+6(S-1))$, $|\mathcal{S}_{2+5i}| = 1+Y(10+6(S-1))$, $|\mathcal{S}_{1+5i}| = 1+Z(10+6(S-1))$, for $i = 0, ..., T$, except $|\mathcal{S}_{4+5T}| = 7$, $|\mathcal{E}| = 49+T(56+(Y-2)(31+(X-2)(13+4Z))+3(X-2)(1+YZ)+2X+5Y+3Z))$.

We run our implementation on this four models using a cluster of Pentium IV 3GHz workstations, each with 512MB RAM, connected by Gigabit Ethernet and running Red-Hat 9.0 Linux with MPI2 on TCP/IP. Table 1 shows runtimes, total memory requirements for the W workstations, and maximum memory requirements among the W workstations, for sequential SmArT (SEQ) and the original SmArTNow (DISTR), and the percentage change w.r.t. DISTR for the naïve (NAÏVE), and the history-based (HIST) speculative firing predictions; "*d*" means that dynamic memory load balancing is triggered, "*s*" means that, in addition, memory swapping occurs.

W	**Time** (sec)			**Total Memory** (MB)			**Max Memory** (MB)		
	DISTR	NAÏVE	HIST	DISTR	NAÏVE	HIST	DISTR	NAÏVE	HIST
Slotted ring network protocol									
$N = 200$ $\lvert\mathcal{S}\rvert = 8.38 \cdot 10^{211}$							SEQ completes in 108sec using 284MB		
2	119	−24%	−13%	286	+3%	+45%	197	+1%	+53%
4	139	−27%	−15%	286	+11%	+51%	127	+61%	+58%
8	182	−32%	−24%	286	+129%	+62%	69	+239%	+62%
$N = 300$ $\lvert\mathcal{S}\rvert = 8.38 \cdot 10^{211}$							SEQ does not complete in 5 hrs using 512MB		
2	^{s}552	^{s}+5%	^{s}−5%	962	+25%	+11%	562	+8%	+7%
4	^{d}490	> 5hrs	^{d}−16%	962	-	+34%	352	-	+12%
8	564	> 5hrs	−39%	962	-	+50%	252	-	+23%
Flexible manufacturing system									
$N = 300$ $\lvert\mathcal{S}\rvert = 3.64 \cdot 10^{27}$							SEQ completes in 55sec using 241MB		
2	79	−8%	−8%	243	+12%	+24%	121	+26%	+52%
4	91	^{d}+67%	−9%	243	+102%	+30%	119	+205%	+50%
8	260	-	−30%	243	-	+42%	103	-	+47%
$N = 450$ $\lvert\mathcal{S}\rvert = 6.90 \cdot 10^{29}$							SEQ does not complete in 5 hrs using 512MB		
2	^{s}257	^{s}+12%	^{s}−14%	826	+16%	+5%	512	+15%	+7%
4	^{d}311	> 5hrs	^{d}−18%	826	-	+33%	372	-	+6%
8	959	> 5hrs	−25%	826	-	+61%	343	-	+6%
Round robin mutex protocol									
$N = 800$ $\lvert\mathcal{S}\rvert = 1.20 \cdot 10^{196}$							SEQ completes in 27sec using 290MB		
2	29	+37%	+6%	293	+110%	+85%	215	+52%	+63%
4	36	+33%	+8%	293	+348%	+109%	130	+186%	+65%
8	51	+33%	+5%	293	+807%	+148%	73	+433%	+73%
$N = 1100$ $\lvert\mathcal{S}\rvert = 3.36 \cdot 10^{334}$							SEQ does not complete in 5 hrs using 512MB		
2	^{d}65	^{s}+62%	^{s}+18%	794	+46%	+6%	379	+79%	+30%
4	47	^{s}+131%	^{d}+10%	794	+119%	+38%	265	+104%	+40%
8	56	^{d}+164%	+7%	794	+299%	+50%	173	+126%	+38%
Runway safety monitor									
$Z = 2$ $\lvert\mathcal{S}\rvert = 1.51 \cdot 10^{15}$							SEQ completes in 236sec using 314MB		
2	731	> 10hrs	−2%	332	-	+39%	191	-	+48%
4	938	> 10hrs	−8%	332	-	+88%	190	-	+30%
8	1480	> 10hrs	−22%	332	-	+128%	173	-	+13%
$Z = 3$ $\lvert\mathcal{S}\rvert = 5.07 \cdot 10^{15}$							SEQ does not complete in 10 hrs using 512MB		
2	^{s}11280	> 10hrs	^{s}−1%	962	-	+10%	595	-	+16%
4	^{d}9762	> 10hrs	^{d}−15%	962	-	+31%	371	-	+8%
8	^{d}14101	> 10hrs	^{d}−17%	962	-	+58%	359	-	+6%

Table 1
Experimental results.

Even though the first two models have different characteristics (*slotted ring* has fixed-size nodes and numbers of levels K and events $|\mathcal{E}|$ linear in the parameter N; *FMS* has node size linear in N and fixed K and $|\mathcal{E}|$), both show that the pattern recognition approach improves the runtime of DISTR, more so as the number of workstations W increases, up to 39%. Indeed, NAÏVE and HIST are even faster than SEQ for *slotted ring* with $N = 200$ when $W = 2$. Furthermore, with HIST, the firing prediction is quite effective: mostly, only useful firing patterns are explored, resulting in a moderate increase in the memory requirements.

However, NAÏVE works well only if there is plenty of available memory, e.g., *slotted ring* with $N = 200$. Even then, though, increasing the number of workstations W can be counter-productive, because this increases their idle time, causing them to pursue an excess of speculative firings. This, in turn, can overwhelm the caches and the node memory and trigger expensive dynamic memory load balancing or even memory swapping, eventually slowing down the computation to levels below those of DISTR, as is the case when $N = 300$. Also, whenever W increases, the memory requirements for the most loaded workstation should decrease, as additional workstations should share the overall memory load. This holds for DISTR and HIST, but not for NAÏVE. This is even more evident for *FMS*.

Round robin mutex is a worst-case example for our approach, as no useful event firing pattern exists. We present it for fairness, but also to stress the resilience of our HIST approach. While the memory and time of NAÏVE increase dramatically because it explores many useless speculative firings, those of HIST increase only slightly, showing that HIST, being unable to help due to the lack of firing patterns, at least does not hurt much in terms of overhead.

Finally, the *RSM*, a real system being developed by National Aeronautics and Space Administration (NASA) [24], has $K = 15$, too close to W for our horizontal slicing scheme to work well. The results for SEQ are indeed much better than for any of the distributed versions, but only when SEQ can run. DISTR and HIST can still run for the second set of parameters, when SEQ fails due to excessive memory requirements. In this case, our HIST heuristic reduces the runtime with minimal additional memory overhead, confirming that event firing patterns exist in realistic models.

5 Symbolic state-space generation over a NOW

Most parallel or distributed work on symbolic state-space generation employs a vertical slicing scheme to parallelize BDD manipulations by decomposing boolean functions in breath-first fashion and distributing the computation over a NOW [14,17,26]. This allows the algorithm to overlap the image computation. However, if the slicing choice is poor, a substantial number of additional nodes is created, and it is generally agreed that finding a good slicing is not trivial [19]. Thus, some synchronization is required to minimize redundant

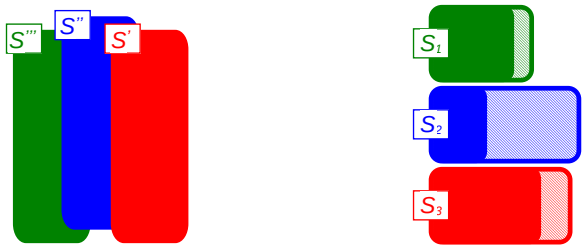

Fig. 7. Vertical slicing vs. horizontal slicing with firing prediction.

work, and this can reduce the scalability of this approach. [13] suggests to employ a host processor to manage the job queue for load-balance purposes and to reduce the redundancy in the image computation by slicing according to boolean functions that use an optimal choices of variables, in order to minimize the peak number of decision diagram nodes required, thus the maximum workload, among the workstations. However, no speedup is reported.

Instead, [4,23] partition the decision diagram horizontally onto a NOW, so that each workstation exclusively owns a contiguous range of decision diagram levels. Since the distributed image computation does not create any redundant work at all, synchronization is avoided. Also, with a horizontal slicing scheme. only peer-to-peer communication is required, so scalability is not an issue anymore. Yet, there is a tradeoff in that, to maintain canonicity of the distributed decision diagram, the distributed computation is sequentialized, which implies that there is no easy opportunity for speedup.

In fact, our pattern recognition approach for event firing prediction attacks this limitation while retaining the horizontal slicing scheme. However, just like the redundant work introduced by vertical slicing, our approach introduces some useless work. More precisely, even though the MDD remains canonical, additional disconnected MDD nodes can be generated. Fig. 7 shows the difference between these two approaches, where the solid boxes indicate the state space and the shaded boxes indicate the useless MDDs. Certainly, the vertical slicing approach can reorder the MDD variables to improve the node distribution, but the variable reordering operation is expensive and requires heavy synchronization. Instead, in our approach, each workstation can clean up disconnected MDD nodes at runtime without requiring any synchronization. Thus, our approach does not hurt the scalability, which is one of the advantages of a horizontal slicing scheme.

Our approach does not achieve a clear speedup with respect to the best sequential implementation. However, at least, it opens the possibility for speeding up symbolic state-space generation on a NOW in conjunction with a horizontal decision diagram slicing scheme.

6 Conclusions

We presented a pattern recognition approach to guide the speculative computation of event firings, and used it to improve the runtime of the distributed

saturation algorithm for state-space generation. Experiments show that recognizing event firing patterns at runtime during saturation is effective on some models, including that of a realistic system being developed by NASA.

We envision several possible extensions. First, while our idea is implemented for a saturation-style iteration, it is also applicable to the simpler breadth-first iteration needed in (distributed) CTL model checking. Second, having showed the potential of speculative firing prediction, we plan to explore more sophisticated, but still low-overhead, heuristics that improve the usefulness of the predicted events, while being more aggressive in the prediction when many workstation are idle. Finally, our heuristics should be augmented to include information about the current memory consumption.

7 Acknowledgment

We wish to thank Radu Siminiceanu and Christian Shelton for discussions on the anticipation approach and the referees for their helpful suggestion.

References

[1] A. Bell and B. Haverkort. Sequential and distributed model checking of Petri nets. *STTT*, 7(1):43–60, 2005.

[2] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comp.*, 35(8):677–691, 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] M.-Y. Chung and G. Ciardo. Saturation NOW. Proc. *QEST*, pp.272–281, 2004.

[5] G. Ciardo, G. Lüettgen, and R. Siminiceanu. Efficient symbolic state-space construction for asynchronous systems. Proc. *ICATPN*, pp.103–122, 2000.

[6] G. Ciardo, G. Lüttgen, and R. Siminiceanu. Saturation: An efficient iteration strategy for symbolic state space generation. Proc. *TACAS*, pp.328–342, 2001.

[7] G. Ciardo, R. Marmorstein, and R. Siminiceanu. Saturation unbound. Proc. *TACAS*, pp.379–393, 2003.

[8] G. Ciardo and R. Siminiceanu. Structural symbolic CTL model checking of asynchronous systems. Proc. *CAV*, pp.40–53, 2003.

[9] G. Ciardo, R. Jones, A. Miner, and R. Siminiceanu. Logical and stochastic modeling with SMART. *Performance Evaluation*, to appear.

[10] E. Clarke, O. Grumberg, and D. Peled. *Model checking*. 1999.

[11] S. Gai, M. Rebaudengo, and M. Sonza Reorda. A data parallel algorithm for boolean function manipulation. Proc. *FMPSC*, pp.28–36, 1995.

[12] S. Graf, B. Steffen, and G. Lüttgen. Compositional minimisation of finite state systems using interface specification. *Formal Asp. of Comp.*, 8(5):607–616, 1996.

[13] O. Grumberg, T. Heyman, and A. Schuster. A work-efficient distributed algorithm for reachability analysis. Proc. *CAV*, pp.54–66, 2003.

[14] T. Heyman, D. Geist, O. Grumberg, and A. Schuster. Achieving Scalability in Parallel Reachability Analysis of Very Large Circuits. Proc. *CAV*, pp.20–35, 2000

[15] J. Burch, E. Clarke, and D. Long. Symbolic model checking with partitioned transition relations. Proc. *VLSI*, pp.49–58, 1991.

[16] S. Kimura and E. Clarke. A parallel algorithm for constructing binary decision diagrams. Proc. *ICCD*, pp.220–223, 1990.

[17] K. Milvang-Jensen and A. Hu. BDDNOW : A parallel BDD package. Proc. *FMCAD*, pp.501–507, 1998.

[18] A. Miner and G. Ciardo. Efficient reachability set generation and storage using decision diagrams. Proc. *ICATPN*, pp.6–25, 1999.

[19] A. Narayan, A. Isles, J. Jain, R. Brayton, and A. Sangiovanni-Vincentelli. Reachability analysis using Partitioned-ROBDDs. Proc. *ICCAD*, pp.388–393, 1997.

[20] D. Nicol and G. Ciardo. Automated parallelization of discrete state-space generation. *J. Par. and Distr. Comp.*, 47:153–167, 1997.

[21] Y. Parasuram, E. Stabler, and S.-K. Chin. Parallel implementation of BDD algorithm using a distributed shared memory. Proc. *HICSS*, pp.16–25, 1994.

[22] E. Pastor, O. Roig, J. Cortadella, and R. Badia Petri net analysis using boolean manipulation. Proc. *ICATPN*, pp.416–435, 1994.

[23] R. Ranjan, J. Snaghavi, R. Brayton, and A. Sangiovanni-Vincentelli. Binary decision diagrams on network of workstations. Proc. *ICCD*, pp.356–364, 1996.

[24] R. Siminiceanu and G. Ciardo. Formal verification of the NASA Runway Safety Monitor. Proc. *AVoCS*, 2004.

[25] U. Stern and D. L. Dill. Parallelizing the Murϕ verifier. Proc. *CAV*, pp.256–267, 1997.

[26] T. Stornetta and F. Brewer. Implementation of an efficient parallel BDD package. Proc. *DAC*, pp.641–644, 1996.

[27] B. Yang and D. O'Hallaron. Parallel breadth-first BDD construction. Proc. *PPoPP*, pp.145–156, 1997.