Response property checking via distributed state space exploration

Brad Bingham and Mark Greenstreet
Department of Computer Science, University of British Columbia
201-2366 Main Mall, Vancouver, B.C., Canada, V6T 1Z4
{binghamb, mrg}@cs.ubc.ca

***Abstract*—A response property is a simple liveness property that, given state predicates p and q, asserts "whenever a p-state is visited, a q-state will be visited in the future". This paper presents an efficient and scalable implementation for explicit-state model of checking response properties on systems with strongly- and weakly-fair actions, using a network of machines. Our approach is a novel twist on the One-Way-Catch-Them-Young (OWCTY) algorithm. Although OWCTY has a worst-case time complexity of $O(n^2m)$ where n is the number of states of the model, and m is the number of fair actions, we show that in practice, the run-time is a very small multiple of n. This allows our approach to handle large models with a large number of fairness constraints. Our implementation builds upon PREACH, a distributed, explicit-state model checking tool. We demonstrate the effectiveness of our approach by applying it to several standard benchmarks on some real-world, proprietary, architectural models. *Index Terms*—distributed model checking, explicit-state model checking, murphi, liveness, fairness**

I. INTRODUCTION

Response properties are liveness properties of the form "From any state in which proposition p is satisfied, execution will eventually reach a state in which proposition q is satisfied." In LTL such properties are expressed as $\Box(p \rightarrow \Diamond q)$; the corresponding CTL specification is $\mathsf{AG}\,(p \rightarrow \mathsf{AF}\,q)$. Specifications of cache protocols and high-level architectural models often include response properties – e.g. if a processor attempts to write to a memory location, the processor will eventually have an exclusive copy of that location in its cache; or, if an instruction is fetched, eventually either it will be executed and committed or that (speculative) path will be aborted.

The standard approach to explicit state model checking of LTL properties involves constructing a product automaton that is the synchronous cross product of the Büchi automaton that accepts the negation of the property in question, and the Büchi automaton for the system itself [1], [2]. If the language accepted by the product automaton is empty, then the LTL property holds; otherwise, a counterexample trace is found. All model checking approaches are vulnerable to state-explosion problems, and the product-automaton construction for LTL model checking exacerbates this problem. If the original system has n reachble states, and the LTL specification, ϕ, consists of $|\phi|$ symbols and operators, then constructing the product automaton takes $\mathcal{O}(n2^{|\phi|})$ time and space.

This research was funded in part by generous support from NSERC Canada and Intel through their CRD and URO grants.

Response properties can be expressed with a Büchi automaton with only 2 states, and thus the blowup from the formula size is curbed. Unfortunately, only contrived systems that contain no cycles along any path from a p-state to a q-state will satisfy response. In practice, response is verified subject to *fairness assumptions* that attempt to characterize realistic traces. Response may be verified under those fairness assumptions that can be written as the LTL formula $Fair$, by using LTL model checking to verify the formula $Fair \rightarrow \Box(p \rightarrow \Diamond q)$. The Büchi automaton for this formula will grow exponentially in $|Fair|$, which in turn causes the number of states of the product automaton to explode.

Instead of expressing fairness as an antecedent to the LTL property of interest, fairness can be expressed in terms of how the original system is defined or as a specially handled input to the model checking algorithm. Kesten *et al.* [3] compare expressing fairness as a property antecedent with a "fair-aware" approach and show that latter achieves better performance. Manna and Pnueli [4], [5] present a model-checking algorithm property checking for response properties that takes advantage of two notions of action-based fairness. The Divine distributed explicit-state model checking tool has a specific mode where all transitions are assumed to be weakly fair [6]. In this paper, we follow suit and employ an algorithm that directly utilizes fairness assumptions for Manna and Pneuli's notions of strong and weak fairness. In the worst-case, the algorithm could perform $O(n^2|Fair|)$ state expansions, where n is the number of reachable system states. In the typical scenario where $|Fair|$ is much smaller than $\log(n)$, this far exceeds the number of worst-case expansions of the Büchi automaton approach which is $O(n2^{|Fair|})$. However, our results show on benchmark models that the algorithm vastly outperforms the worst case, which is indeed achievable (see online Appendix [7]). In contrast, Section VII reports results for a tool that implements the Büchi automaton approach and uses time and memory as one would expect from the worst-case analysis.

Our contributions are as follows:

1) present a novel, efficient, parallel approach for model checking response properties;
2) an implementation of the algorithm built as an extension of the PREACH [8], [9] model checker. PREACH is a distributed, explicit-state model checker based on Stern and Dill's [10] algorithm;

ISBN: 978-0-9835678-4-4.

3) demonstrate that verifying liveness in large, realistic systems augmented with both strong and weak fairness is tractable using a modest network of machines;
4) show that the time requirements for One-Way-Catch-Them-Young style algorithms are far better in practice than would be expected from the worst case analysis. In practice, we observe that each state is visited a small number of times (typically less than 30).

II. Overview

Stern and Dill's distributed model checking algorithm [10] partitions the state space among processes with a uniform random hash function. Processes are said to *own* states that hash to their process IDs. Once a state has been visited, its owner process is responsible for storing it locally. In PReach this is done with the Murφ model checker's hash table [11] which uses a predetermined number of bits[1] to represent each state. The use of hash compaction and bloom filters in explicit-state model checking is a thoroughly studied area [12], [13] and lends itself to practical approaches. Hash table compression admits a small probability that some state will erroneously be viewed as visited when it actually hasn't been. In our experience this probability is tiny; for example, a very large model checking experiment with about 100 billion states had only a 0.03% chance of a missed state [8]. The experiments in this paper admit a much smaller probability than this; the `German6` model with over 316 million states had a probability of a missed state of less than 7.36×10^{-5}. If this probability were of practical concern, the user could simply re-run the tool using a different seed for randomization and reduce the probability of a missed state in *both* runs to less than $(7.36 \times 10^{-5})^2 < 5.42 \times 10^{-9}$.

Once a state has been checked in the hash table, **HT**, it is queued for expansion in the work queue, **WQ**, the other key data structure of the Stern-Dill algorithm. Unlike the **HT** which has static size and resides in memory, the **WQ** has dynamic size and stores full state descriptors. Typically only a small percentage of the **WQ** is in memory; the rest is delegated to disk. Because states can be read and written in large batches, using disk storage for the **WQ** does not create a bottleneck. A key feature to PReach performance, particularly in a heterogeneous computing environment, is load balancing. Once a state enters **WQ**, it is irrelevant which process actually checks the invariants, computes the successor states and sends them off to their respective owners. Thus, processes that amass a longer **WQ** will offload a chunk of their states to another process with a shorter **WQ**.

Erlang's message passing system relies on nonblocking sends. When a message arrives for some process, it resides in a message inbox in memory until a matching Receive is called. The dynamic nature of distributed state space exploration and the performance asymmetry introduced by heterogeneous machines, or any other performance irregularities, can lead to very long messages queues. This is especially problematic as we have observed that the time it takes the Erlang runtime to consume an inbox message increases with the number of messages in the process's inbox. To combat this issue, PReach employs a crediting mechanism that bounds the size of each process' inbox. If process A has states to send to their owner, process B, but it does not have sufficient credits to do so, the states are simply queued in A's "outbox" for B. Outboxes that grow large are also written to disk.

To check response properties, we have implemented an algorithm inspired by the set-based *One-Way-Catch-Them-Young* algorithm described in [14], [15]. We focus on systems with both *strongly fair actions* (a.k.a. compassion), denoted $\mathcal{C}$ and *weakly fair actions* (a.k.a. justice), denoted $\mathcal{J}$.

A. Preliminaries

A fair transition system, FTS, is a tuple $(\mathcal{S}, I, T, \mathcal{J}, \mathcal{C})$ where

- $\mathcal{S}$ is a finite set of states;
- $I \subseteq \mathcal{S}$ is the set of initial states;
- transition relation $T \subseteq \mathcal{S} \times \mathcal{S}$;
- weakly fair actions $\mathcal{J} \subseteq 2^T$;
- strongly fair actions $\mathcal{C} \subseteq 2^T$.

An *action* is a subset of T. Function $En : \mathcal{S} \to 2^{\mathcal{C} \cup \mathcal{J}}$ gives the set of actions enabled at state s, i.e. $En(s) = \{a \in \mathcal{C} \cup \mathcal{J} : \exists s'.\ (s, s') \in a\}$. State s enables action a if $a \in En(s)$. Given state s we use the shorthand notations $\mathcal{C}_s$ and $\mathcal{J}_s$ to refer to the sets of enabled actions that are strongly and weakly fair, respectively. Formally, $\mathcal{J}_s = \mathcal{J} \cap En(s)$ and $\mathcal{C}_s = \mathcal{C} \cap En(s)$. For convenience we assume transitions that are not members of any element of $\mathcal{J} \cup \mathcal{C}$ are members of the *non-fair* set, i.e. $NF = T \setminus \left(\bigcup_{a \in \mathcal{J} \cup \mathcal{C}} a\right)$. For $A \subseteq \mathcal{S}$, $\langle A \rangle$ denotes the subgraph of the digraph $(\mathcal{S}, T)$ induced by A.

A *trace* is a finite sequence of states $s_0 \circ s_1 \circ \ldots \circ s_k$ where $s_o \in I$, and $(s_i, s_{i+1}) \in T$ for $0 \leq i < k$. A *predecessor trace* for state s is any trace where $s_k = s$.

An *execution* is an infinite sequence of states, $s_0 \circ s_1 \circ \ldots$, where $s_0 \in I$, and $\forall i \geq 0.\ (s_i, s_{i+1}) \in T$. For a given trace, action a satisfies

- $\mathit{InfOftenTaken}(a)$, if $\forall i \geq 0.\ \exists j \geq i.\ (s_j, s_{j+1}) \in a$,
- $\mathit{InfOftenEn}(a)$, if $\forall i \geq 0.\ \exists j \geq i.\ a \in En(s_j)$, and
- $\mathit{InfOftenDisabled}(a)$, if $\forall i \geq 0.\ \exists j \geq i.\ a \notin En(s_j)$.

An execution is called *fair* if

$$\begin{aligned} & \forall a \in \mathcal{C}.\ \mathit{InfOftenEn}(a) \Rightarrow \mathit{InfOftenTaken}(a) \\ \wedge\ & \forall a \in \mathcal{J}.\ \mathit{InfOftenTaken}(a) \vee \mathit{InfOftenDisabled}(a). \end{aligned}$$

In other words, an execution is fair if all actions of $\mathcal{C}$ are taken infinitely often or are never enabled beyond some finite prefix of the execution, and all actions of $\mathcal{J}$ are taken infinitely often *or* are disabled infinitely often. A strongly connected component (SCC) is called *fair* (a FSCC) if all enabled strongly fair actions in the SCC's states are taken within the SCC, and all enabled weakly fair actions in the SCCs states are either taken within the SCC or disabled at some state. Section III presents an algorithm that detects FSCCs within

[1]This number is a configuration parameter. The results in this paper use the default value of 40 bits.

the subgraph of reachable states that can be reached on a path from some p-state without visiting a q-state along the way (this subset is referred to as *pending*; see Figure 1). Such SCCs are counterexamples to the response property $\Box(p \rightarrow \Diamond q)$; furthermore, every counterexample execution has an infinite suffix that only visits states in a FSCC. Note that p is a subset of $pending$, and q is disjoint with $pending$. The initial states are usually disjoint from both p and $pending$, but this need not be the case.

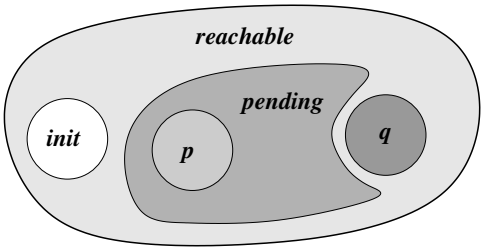

Fig. 1: Sets of interest when checking a system adheres to $\Box(p \rightarrow \Diamond q)$.

B. A note about stuttering

We note that fair systems may be defined with or without inherent stuttering, the former assuming that every state has a transition to itself and the latter does not. For simplicity in the following presentation, we assume that stuttering is allowed, thereby requiring a fair "reason" why indefinite stuttering cannot occur. This assumption requires that T is reflexive. Including stuttering simplifies the presentation; for example, it ensures that all traces can be extended to infinite executions.

III. Algorithm

Our distributed response checking algorithm is based on the One-Way-Catch-Them-Young (OWCTY) [14] approach. The key idea of the algorithm is to begin by initializing a set, $MaybeFair$, with the $pending$ states, and then iteratively remove states from $MaybeFair$ that cannot belong to a FSCC. A state, s, is removed when it is discovered that there is no predecessor trace of s in $\langle MaybeFair \rangle$ along which action $a \in \mathcal{C}$ is taken, where $a \in \mathcal{C}_s$. Similarly, s is removed if it is found that there is no predecessor trace of s in $\langle MaybeFair \rangle$ along which action $a \in \mathcal{J}_s$ is either taken *or* disabled at some state s' of the trace, where $a \in \mathcal{J}_s$. The response property holds iff $MaybeFair$ is empty when the algorithm terminates. To see this, note that any state that is removed from $MaybeFair$ cannot belong to a FSCC; thus, $\langle MaybeFair \rangle$ contains all of the FSCCs of $\langle pending \rangle$. The FSCCs of $\langle MaybeFair \rangle$ form a DAG. Let F be any FSCC of $\langle MaybeFair \rangle$ that has no predecessor FSCCs. It is straightforward to construct a cycle in F that satisfies all fairness constraints. By construction, this cycle is reachable from some initial state.

The description of OWCTY from [15] for model checking LTL formulas with strong and weak state-based fairness operates on sets of states performing union and disjunction operations, as well as deleting all members from a set which have

Algorithm 1 High level algorithm

```
 1 procedure FindFairCycle(S, I, T, C, J, p, q)
 2     ▷ Compute the pending states
 3     pending ← Reachability(S, I, T, p, q)
 4     ptfa ← new bit[pending][J ∪ C]                ▷ array of bit-strings
 5     Clear(ptfa)                                   ▷ initialize to all 0s
 6     MaybeFair ← pending
 7     Prev ← ∅
 8     while MaybeFair ≠ Prev do
 9         Prev ← MaybeFair
10         ToExpand ← MaybeFair
11         while ToExpand ≠ ∅ do
12             s ← RemoveSomeElement(ToExpand)
13             ▷ Weakly fair actions not enabled at s
14             for all a ∈ J \ J_s do
15                 ptfa[s][a] ← 1
16             end for
17             Next ← Successors(s) \ q
18             for all s' ∈ Next do
19                 OldActions ← ptfa[s']
20                 a ← WhatActionTaken(s, s')
21                 if a ∈ J ∪ C then
22                     ptfa[s'][a] ← 1               ▷ Record action taken
23                 end if
24                 ▷ Actions preceeding s also preceed s'
25                 ptfa[s'] ← BitwiseOr(ptfa[s], ptfa[s'])
26                 if (ptfa[s'] ≠ OldActions) then
27                     ToExpand ← ToExpand ∪ {s'}
28                 end if
29             end for
30         end while
31         for all s ∈ MaybeFair do
32             if ∃a ∈ J_s ∪ C_s : ptfa[s][a] = 0 then
33                 MaybeFair ← MaybeFair − {s}
34             end if
35         end for
36         Clear(ptfa)
37     end while
38     return MaybeFair ≠ ∅
39 end procedure
```

no predecessor within the set until a fixed point is reached[2]. As described in Section II, PReach uses lossy compression when hashing states; thus, we cannot reconstruct states from hashtable entries. To retain the efficiency advantages of the Murφ hashtables, we avoid the explicit representation of large sets of states, and replace the union and intersection operations of OWCTY with tag bit manipulations, where each hash table entry includes one such tag bit per fair action. In Algorithm 1, these bits are stored in *ptfa (predecessor trace fair actions)*, which is a two-dimensional array of bits initialized to all 0s. Bit $ptfa[s][a]$ is set for action $a \in \mathcal{J} \cup \mathcal{C}$ and state $s \in MaybeFair$ is set if a is taken in a predecessor trace of s in $\langle MaybeFair \rangle$, or if $b \in \mathcal{J}$ is disabled at some state of a predecessor trace of s in $\langle MaybeFair \rangle$. The set $pending$ stores the states of interest for response, those that can be reached on a path from a p-state without visiting a q-state.

Each iteration of the outer while-loop is called a *round*, and involves two *phases*.

Action Propagation Phase (AP):
This step is the while-loop from lines 11 to 30. Some state s is removed from $ToExpand$ and the tag bits are set for each

[2]To the best of our knowledge, the algorithm from [15] not been implemented.

weakly fair action that is disabled at s; this is because any eventual successor of s within $\langle pending \rangle$ *may* be part of an SCC with s. If so, this SCC is fair with respect to these weakly fair actions. Then, the successors of s within $\langle pending \rangle$ are computed. For each of these the current tag bits are saved in $OldActions$. If the transition that is taken from s to reach a successor s' is a member of some $a \in \mathcal{J} \cup \mathcal{C}$, the $ptfa[s'][a]$ is set (line 22). Then, the bit-string $ptfa[s]$ is ORed with the $ptfa[s']$, as any predecessor trace ρ for s implies a predecessor trace for s', namely $\rho \circ s'$. If any of these operations have set new bits for s', it must be added to $ToExpand$ so the bits are propagated along. Otherwise, the s' is discarded. This loop continues until a fixed point is reached for the contents of $ptfa$.

Figure 2 illustrates some operations of AP with an example. For this example, $\mathcal{J} = \{a_0, a_1, a_2, a_3\}$ and $\mathcal{C} = \{a_4, a_5, a_6, a_7\}$, and PTFAs are represented as $a_7 \ldots a_0$, as seen below each state. Assume that $En(b) = \{a_0, a_2, a_3, a_4\}$, $En(c) = \{a_0, a_1, a_7\}$, and $En(d) = \{a_0, a_1, a_3, a_5\}$. When b is expanded, the PTFA on the arc is passed to state e which changes the PTFA for e and requires e to be expanded. Subsequently, c is expanded and the PTFA for e is again updated and another e expansion is needed to communication the new PTFA to successors. Finally, when d is expanded the PTFA sent to e contains no new actions, so e does not need another expansion.

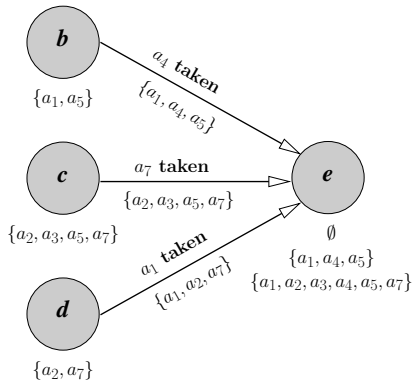

Fig. 2: Example of PTFA updates as states are expanded.

State Deletion Phase (SD):
This phase appears from lines 31 to 36. Any states that enabled a fair action a but with the corresponding tag bit cleared cannot be part of a FSCC and are removed from $MaybeFair$.

Soundness for the algorithm was described at the beginning of this section. To see that the algorithm terminates, we first note that the while loop at lines 10–28 must terminate because the flag bits in $ptfa$ are strictly increasing with successive iterations of the loop. The while-loop at lines 7–35 terminates because the loop adds no new elements to $MaybeFair$.

IV. Distributed Implementation

The distributed version of this algorithm starts with a Stern-Dill style reachability computation that identifies all p and $pending$ states. Each worker process stores its p states on disk, and $pending$ states are marked with tag bits in the hash-table. Initially, the PTFA for $pending$ states are set to all fair actions, $\mathcal{J} \cup \mathcal{C}$. The distributed algorithm then performs rounds that correspond to those of the sequential version, Algorithm 1. As described in more detail below, each round propagates PTFA tags according to the next state relation until a fix point is reached. At the boundary between rounds, states are identified whose PTFAs do not satisfy the fairness constraints for the state. Such states cannot be part of an FSCC and are marked as "dead" (i.e, removed from $MaybeFair$). The number of live, $MaybeFair$ states is non-increasing. The algorithm terminates when this number no-longer decreases. If at this point, all $MaybeFair$ states have been eliminated, then the response property is satisfied. Otherwise, counter-example is generated. The remainder of this section describes this algorithm in more detail.

Algorithm 2 shows pseudo-code for the root process. It initiates the initial reachability computation to identify p and $pending$ states. It then initiates rounds of propagating PTFA tags and eliminating $pending$ states until no further states can be eliminated. The termination detection algorithm from the original Stern and Dill approach is used to identify the end of each round and compute the total number of $pending$ states. This provides a barrier separating the computations of successive rounds. After the final round, the root process notifies the workers that the computation is complete and reports either that the response property has been verified or provides a counter-example.

Algorithm 3 shows pseudo-code for the worker processes. Like the reachability computation, each worker has two main activities: receiving incoming states and checking if they have been "seen" previously, and expanding states to send their successors to their owners. Algorithm 3 augments each of these activities to maintain the tags for PTFAs. At the beginning of each round, each process checks its subset of the p states to determine which ones satisfied their associated fairness constraints in the previous round. Those that don't are marked as dead. All p-states are added to the work-queue, $ToExpand$, even if they are dead to ensure that their successors are examined in this round. When a state is received, the algorithm first checks to see if this is the first time the state has been seen for the current round. If so, the state's PTFA is checked to see if the state should be marked as dead, and all states are entered into $ToExpand$ the first time they are visited in each round. If the state has been seen before, then if the new PTFA indicates incoming paths for fairness constraints that haven't already been satisfied, these constraints are added to the state's PTFA, and the state is enqueued in $ToExpand$ to propagate this information to its successors.

When a worker removes a state from its work queue, $ToExpand$, it computes all successor states as in the original reachability algorithm. Because the incoming paths to this state are prefixes of incoming paths for its successors, the PTFA of the successor must contain the PTFA for this state. Furthermore, if the transition to the new state corresponds to a fair action, then this action is added to the PTFA. These

updates are made to the PTFA for the successor, and the successor with this PTFA set is sent to the successor's owner.

Every operation either marks a state a dead or adds a fair action to some state's PTFA. Thus, the activities for updating fairness information eventually reach a fixpoint and the round terminates. Many optimizations are possible to improve the performance of this algorithm. These are described in the next section.

Algorithm 2 Root Process

```
1  function RootStart(I, p, q)
2      ▷ Tags for initial states
3      for all s ∈ I do
4          SendState((s, ∅))
5      end for
6      CurMaybeFairCount ← Tally(nstates)
7      PrevMaybeFairCount ← CurMaybeFairCount + 1
8      while CurMaybeFairCount ≠ PrevMaybeFairCount do
9          Broadcast(doRound)
10         PrevMaybeFairCount ← CurMaybeFairCount
11         CurMaybeFairCount ← Tally(nstates)
12     end while
13     Broadcast(stop)
14     if CurStates > 0 then
15         return GenerateCounterexampleTrace(...)
16     else
17         return verified
18     end if
19 end function
```

V. Optimizations

Early experiments with a prototype implementation revealed several opportunities to improve performance. We aim to address the average number of state expansions during a phase, the number of states visited during a phase, *and* the number of rounds. A key observation is that for many examples, the number of states in the *pending* set decreases rapidly with successive rounds. Thus, it is important to avoid touching "dead" states so that the work done in later rounds decreases with the smaller size of *pending*. This also means that most of the time is spent in the initial reachability computation and the first two or three rounds of the liveness computation. Thus, optimization should focus on these early rounds. Furthermore, the same state can be updated several times during a single round. Consolidating these updates was simple and led to significant performance gains. The remainder of this section presents three methods of reducing each of these metrics in turn. In addition, various optimizations are inherited from PReach's state space exploration technique. Namely, load balancing of states offers modest speedups even in a homogeneous network of machines. Batching of states into messages containing hundreds or thousands is also of benefit. The reader may consult [8] for details.

A. Saved Expansions

The description in the algorithms and implementations presented so far have states paired with their tags, including PTFAs, when enqueued to the **WQ**. When the **WQ** grows large, state s may arrive tagged with PTFA b_2 while the same

Algorithm 3 Worker Process

```
1  function Worker(S, I, T, J, C, p, q, rootPid)
2      PS ← ComputePStates(S, I, T, J, C, p, q)
                          ▷ Global variable queue that stores p-states
3      RoundCount ← 0
4      while true do
5          case Receive() of                          ▷ Blocking receive
6              doRound → ok
7              stop → break while loop
8          end case
9          RoundCount ← RoundCount + 1
10         for all s ∈ PS do
11             WQ ← InitStateForRound(s, ∅, RoundCount)
12         end for
13         ▷ Stern and Dill's termination alg
14         while round not terminated do
15             while (s, thisPTFA) ← Receive() do      ▷ Nonblk. recv
16                 T ← HT.GetTags(s)
17                 if T.round ≠ RoundCount then
18                     InitStateForRound(s, thisPTFA, RoundCount)
19                 else if ¬T.dead ∧ (thisPTFA ⊈ T.PTFA) then
20                     T.PTFA ← T.PTFA ∪ thisPTFA
21                     WQ.Insert((s, T))
22                     HT.UpdateTags(s, T)
23                 end if
24             end while
25             ExpandAndSend(J, C)                     ▷ See Alg. 4
26         end while
27         send (nstates, MyMaybeFairCount) to rootPid
28     end while
29 end function
30
31 function InitStateForRound(s, thisPTFA, RoundCount)
32     T ← HT.GetTags(s)
33     if Enabled(s) ⊈ T.PTFA then
34         T.dead ← true
35         thisPTFA ← ∅
36     end if
37     T.round ← RoundCount
38     T.PTFA ← thisPTFA
39     WQ.Insert((s, T))
40     HT.UpdateTags(s, T)
41 end function
```

state is waiting for expansion in the **WQ** while paired with PTFA b_1, which matches the PTFA at the **HT** entry for s. When b_2 has at least one bit set that b_1 does not, s is enqueued for expansion in **WQ** paired with PTFA $b_1 \cup b_2$. This renders the earlier **WQ** entry of (s, b_1) redundant and unnecessary.

To avoid this scenario, the **HT** is used to maintain PTFA information, and **WQ** entries do not contain a PTFA. When a state s is enqueued, a new **HT** tag bit $InWQ$ is set; when s is dequeued, $InWQ$ is cleared and the current **HT** value for PTFA is used when computing the PTFA for s's successors. If state s with PTFA b_2 arrives when the **HT** entry has $InWQ$ set, then **HT** PTFA $b_{\mathbf{HT}}$ is set to $b_{\mathbf{HT}} \cup b_2$ and the just-arrived state s is discarded. This approach reduces the number of state expansions at the cost of an additional bit in **HT** per state, and one additional **HT** lookup.

B. Dynamic Kernel

The algorithm implementation above uses the reachable p-states as the *kernel*, defined as follows.

Definition 1: Given a FTS, $K \subseteq \mathcal{S}$ is a *kernel* for $A \subseteq \mathcal{S}$ if A is a subset of the reachable states from K in the digraph

Algorithm 4 Dequeues a **WQ** state and sends next states with tags to their owners.

```
1  function ExpandAndSend(J, C)
2      if IsEmpty(WQ) then
3          return done
4      end if
5      (X, Tags) ← Dequeue(WQ)
6      NextStates ← ComputeSuccessors(X)
7      if Tags.dead then
8          for all s' ∈ NextStates do
9              SendState((s', ∅))
10         end for
11         return
12     end if
13     PTFA ← Tags.PTFA
14     PTFA ← PTFA ∪ (J − Enabled(X))
15     for all s' ∈ NextStates do
16         ActionTaken ← whatActionTaken(X, s')
       ▷ Successor PTFA is current state PTFA with the fair action taken
17         if ActionTaken ∈ NF then
18             NextPTFA ← PTFA
19         else
20             NextPTFA ← PTFA ∪ ActionTaken
21         end if
22         SendState((s', NextPTFA))
23     end for
24     return
25 end function
```

$(\mathcal{S}, T)$.

Note that the initial states I is a kernel for any subset of the reachable states. In the code presented in Section IV, we used the reachable p-states K_p as a kernel for $MaybeFair$ to initiate each phase because K_p is a kernel for every subset of $pending$. Our experiments showed that for typical examples, the number of states in $MaybeFair$ drops rapidly with each SD phase. The expansion of such deleted states can be avoided by modifying K after each SD phase, using an extra **HT** tag bit InK and additional disk space.

During the initial phase, only the p-states have InK set to true, and these states are saved to disk in the kernel-queue. When a state s is removed from $MaybeFair$ during SD that has InK set, this flag is cleared. When a process receives state s' tagged with mode delete_pred (signaling that a predecessor of s' has just been removed from $MaybeFair$), then if s' has its InK flag cleared, it is set to true and s' is added to the kernel-queue. Finally, at the start of an AP phase the kernel-queue is copied to the **WQ** to serve as the set of initial states, but any state encountered that has its InK flag cleared is ignored and removed from the kernel-queue.

While this approach does not necessarily maintain the smallest possible kernel for $MaybeFair$, its simple implementation and low overhead lead to large performance gains.

C. Deletion by Predecessor Counting

There are performance advantages when storing the number of predecessors each state has in $\langle MaybeFair \rangle$. Under the assumption of stuttering and ensuring the safety property that every state $s \in pending$ has $|\mathcal{J}_s \cup \mathcal{C}_s| \geq 1$, any state with 0 predecessors in $\langle MaybeFair \rangle$ will be deleted from $MaybeFair$ in the next SD phase. However, storing the number of predecessors in **HT** allows detection of this case in order to preemptively remove such states. We choose to add 8 bits to the **HT** tags to store the predecessor count. This additional bookkeeping complicates Algorithms 3 and 4 somewhat (details omitted). In particular, a state may be expanded more than once during an SD. This occurs when the first time a state is visited the condition on line 33 of Algorithm 3 holds, but subsequently all of its predecessors are deleted. However, this turns out to be a rare occurrence in the benchmarks, and this strategy can reduce the number of phases. Note that the impact of this optimization is omitted from the Results section as it was inherent to our early implementation versions.

VI. Results

We ran PReach on a variety of combinations of Murφ models with all optimizations of section V enabled, summarized in Table I. For each, we chose a suitable response property such as "requests for exclusive access to a cache line are eventually granted", or "processes waiting to enter the critical section will eventually do so". The Murφ models used are the German cache coherence protocol, the Peterson mutual exclusion algorithm, the MCS lock mutual exclusion algorithm, a snoopy protocol used as a benchmark in previous verification work [16] and an Intel proprietary protocol. Let `GermanX` denote the German model with `X` caches; `petersonY` is Peterson's algorithm with `Y` processes and `mcslock5` is the MCS Lock algorithm with 5 processes; `snoop2` is the snoopy protocol with 2 L1 caches and 2 clusters. Models `saw`, `gbn` and `swp` are various sliding window communication protocols, with the response property that the sender can always eventually accept new data to transmit. All models and the PReach code is provided online [7]. Each Murφ "rule" (a.k.a. guarded command) is considered a separate action; we attached suitable fairness assumptions specific to the model. The network of machines used for experiments are as follows:

- UBC cluster: 40 PReach processes on a homogeneous cluster of 20 Intel Core i7-2600K at 3.40 GHz with 8 GB of memory (non-`intel_*` models).
- Intel cluster: 16 PReach processes on a heterogeneous network of contemporary Intel® Xeon® machines, each with at least 8 GB of memory (`intel_*` models).

Not included in the table, but worth noting, is an Intel proprietary sliding window protocol model. With over 450 million states and tens of fairness (both strong and weak), we were able to verify response in about 5 and a half hours using 32 cores.

A few modifications were required when checking the `snoop` protocol. This model was created to represent a cache-coherence protocol in a realistic processor. The protocol appears to have been designed with an emphasis on safety, and liveness does not appear to have been primary concern. For example, requests for cache lines are clearly not responsive as they may be *negatively acknowledged* (Nackd) an arbitrary number of times. To avoid this, we changed the protocol so that Nacks of this type are simply ignored, and the request

model	runtime	states	p-states	$pending$-states	q-states	rounds	exp/state	no `-ko`	no `-se`	no opt.
`German5_sf`	189	15,836,445	3,699,486	4,858,596	5,103	1	3.48	0.98	2.42	2.86
`German6_sf`	4,253	316,542,087	74,465,244	95,266,520	18,225	1	3.33	1.01	3.30	3.52
`peterson6_wf`	820	13,817,679	2,947,800	12,111,713	45,209	14	12.91	1.65	1.30	1.95
`peterson6_sf`	423	13,817,679	2,947,800	12,111,713	45,209	5	9.03	1.36	1.73	2.12
`peterson7_wf`	26,957	380,268,668	79,029,594	340,549,743	775,138	17	14.19	1.65	1.66	2.16
`peterson7_sf`	14,613	380,268,668	79,029,594	340,549,743	775,138	6	10.11	1.27	2.26	-
`mcslock5_wf`	1415	59,318,541	27,785,789	51,474,427	2,780,517	3	5.09	1.17	1.10	1.25
`snoop2_sf`	160	2,648,763	670,689	1,313,100	1,335,663	3	12.71	1.07	4.57	5.00
`saw20_sf`	323	314,183	309,140	309,140	5,043	23	44.06	1.04	1.09	1.15
`gbn3_2_sf`	369	12,753,395	7,859,200	7,859,200	4894195	6	6.44	1.60	1.95	2.56
`swp4_2_sf`	503	18,595,425	11,715,440	11,715,440	6,879,985	6	6.58	1.59	1.63	2.22
`intel_small_sf`	285	476,778	268,078	268,078	164,057	4	6.36	-	-	-
`intel_med_sf`	1,015	2,696,059	1,944,360	1,944,360	635,672	4	8.59	-	-	-
`intel_big_sf`	13,872	51,791,350	29,899,694	29,899,694	19,855,989	8	11.92	-	-	-

TABLE I: Column "runtime" is given in seconds; "exp/state" is the average number of times each $pending$-state was expanded. Model `peterson6_sf` is `peterson6` with all actions strongly fair, as opposed to `peterson6_wf` where some rules were weakly fair and the rest as not fair (for example, the rule that initiates the move from the noncritical section to requesting to enter the critical section needs no fairness assumption). These two models have the same number of states of each type but perform a different number of expansions, and illustrate the benefit of only using more fairness than required for the response property to hold. All other models require strong fairness.

persists. This turned out to also not be responsive, although less obviously so – the counterexample trace included 72 transitions. Therefore, not all of the pending states were deleted. Online Appendix [7] Figure 5 shows that about half of the the $pending$-states remained in the $MaybeFair$ set when the algorithm terminated. Additional plots for the experiments appear in the Appendix.

The rightmost three columns of Table I show the slowdown when benchmarks are run without the kernel optimization, without the saved expansions optimization and without either, respectively. The kernel optimization is of most benefit when the number of rounds is large[3]. In particular, it is of no benefit for those benchmarks that only require a single round, as the kernel states are only used during subsequent rounds. The saved expansion optimization offers large performance gains in many cases. Typically, only 5 to 10% of the total state expansions are explicitly avoided by the when a just-received state state is present in the **WQ**. However, avoiding these redundant expansions can in turn save many expansions of successor states which in turn saves expansions of states that are two transitions away. This cascading effect decreases the total number of expansions by a significant factor.

VII. Related Work

Divine is a parallel and distributed LTL model checker that is the closest tool to ours [17]. Divine constructs a product Büchi automaton to check liveness properties; thus, Divine's space requirement grows as the product of the number of states in the system model and the number in the system automaton. Applying Divine to the examples from Section VI, we observed that it ran out of memory for all examples except for those with no or a small number of strong fairness constraints. Divine provides a mode for models where all transitions are weakly fair. Using this feature, Divine performed well for the Peterson example for which weak fairness constraints are sufficient to ensure responsiveness. However, many problems require strong fairness; for example cache coherence protocols often include states where taking one action disables another. We found that for an encoding of the German protocol with 4 caches, the reachable state space of Divine's product automaton doubled with *each* additional fair action included. For only 6 fair rules, Divine on a multicore machine took 17 minutes to construct the system automata, 13 minutes to perform the model checking task and used over 16 GB of main memory. In our experiments, adding one more fair rule exhausted the main memory of our 32 GB machine and rendered the computation time infeasible.

Our algorithm has a worst-case time complexity that is at least $O(n^3)$ where n is the number of reachable states – it is straightforward to construct an example where the transition relation has $O(n^2)$ edges, and for which Algorithm 1 removes one state per iteration of the outer while-loop. In practice, we observe that the transition relation is sparse and Algorithm 1 converges in far fewer then n rounds – the most extreme case in Table I has 23 rounds. The *worst-case* time complexity of Divine is better, $O(n2^{|\phi|})$ – Divine replaces a factor of the system model size with the number of states for the checking Büchi automaton. However, our experiments show that the actual time and memory requirements for Divine's algorithm are fairly close to what one would expect from the worst-case bounds, while our approach, in practice, scales much more efficiently. We see this gap between worst-case and actual performance as a promising area for further investigation.

Using a sequential algorithm for accepting cycle detection such as Tarjan's [18], SCCs may be found in $\mathcal{O}(|V| + |E|)$ time. However, such DFS-based algorithms are unsuited to parallelization unless $P = NC$ [19]. Manna and Pnueli presented sequential algorithm for model checking response properties of fair transitions systems [5], but this is not easily parallelizable and so scalability is limited. Recently, Holzmann implemented some interesting liveness checking algorithms in a multicore version of SPIN [20]; however this approach will only find counterexamples of bounded length. Other work related to ours includes that of the authors of the LTSMin

[3]One exception is `saw_20_sf` where a large proportion of the runtime is spent coordinating threads between rounds.

model checking tool, most notably their algorithms for parallel SCC decomposition on multicore machines [21], [22].

VIII. Conclusions and Future Work

We have extended the PReach explicit-state, distributed model-checking tool to support verification of response properties under both strong and weak fairness of actions. Our approach uses multiple rounds of reachability computation to implement a variation of the OWCTY algorithm. For a model with n states, m fairness constraints, OWCTY could expand states $O(nm)$ times on average. This would be prohibitively expensive. Our implementation shows that for practical examples, the number of rounds is small – typically less than 30, with a maximum of about 44. Thus, OWCTY appears to provide a practical approach to checking response properties for real-world problems. For these examples, liveness checking is slower than safety checking, but not prohibitively so.

Implementing our algorithm on top of the PReach distributed model checker allows it to exploit the aggregate memory of large compute clusters. This enabled verification of response properties for a sliding-window protocol with over 450 million states in about $5\frac{1}{2}$ hours.

We compared our approach with a tool that uses the standard product-automaton formulation, with one automaton for the system model, and the other for the LTL liveness property. As predicted by the worst-case analysis, we observed that the size of the property automaton grew exponentially with the number of fairness constraints. The product-automaton approach was significantly faster than PReach for the problems that it could complete. However, it ran out of memory for all but the smallest examples.

This approach can be generalized in a number of directions. One is to handle other simple liveness properties such as *reactivity*, expressed in LTL as $\Box\Diamond p \vee \Diamond\Box q$, where p and q are past formulas. We hope to combine these model checking methods with the decompositional inference rules of Manna and Pnueli [4], [5]. Such decompositions establish that a response property is implied by a handful of safety properties and "smaller" response properties, i.e. depending on a smaller fraction of the state space. Adapting our algorithm to verify multiple such response properties in the same model checking run would leverage human insight to increase performance.

Acknowledgments

The authors extend their gratitude to Jiří Barnat for his help in understanding and running Divine. They also appreciate assistance from colleagues Jesse Bingham and Jim Grundy at Intel for providing examples of architectural models during the first author's internship, and Flemming Andersen for his vision and support for developing and demonstrating scalable verification methods and tools.

References

[1] M. Y. Vardi and P. Wolper, "An automata-theoretic approach to automatic program verification," in *Proceedings of the 1st Annual Symposium on Logic in Computer Science (LICS'86)*. IEEE Comp. Soc. Press, Jun. 1986, pp. 332–344.

[2] R. Gerth, D. Peled, M. Y. Vardi, R. Gerth, D. D. Eindhoven, D. Peled, M. Y. Vardi, and P. Wolper, "Simple on-the-fly automatic verification of linear temporal logic," in *In Protocol Specification Testing and Verification*. Chapman & Hall, 1995, pp. 3–18.

[3] Y. Kesten, A. Pnueli, L.-O. Raviv, and E. Shahar, "Model checking with strong fairness," *Form. Methods Syst. Des.*, vol. 28, pp. 57–84, January 2006. [Online]. Available: http://portal.acm.org/citation.cfm?id=1116046.1116050

[4] Z. Manna and A. Pnueli, "Completing the temporal picture," *Theor. Comput. Sci.*, vol. 83, pp. 97–130, June 1991. [Online]. Available: http://portal.acm.org/citation.cfm?id=111775.111780

[5] ——, "Temporal verification of reactive systems: Progress (draft)," http://theory.stanford.edu/~zm/tvors3.html, 1996.

[6] J. Barnat, J. Havlíček, and P. Ročkai, "Distributed LTL Model Checking with Hash Compaction," *Electr. Notes Theor. Comput. Sci.*, vol. 296, pp. 79–93, 2013. [Online]. Available: http://dx.doi.org/10.1016/j.entcs.2013.07.006

[7] B. Bingham, "Preach-response," https://bitbucket.org/binghamb/preach-response, 2013.

[8] B. Bingham, J. Bingham, F. M. de Paula, J. Erickson, G. Singh, and M. Reitblatt, "Industrial strength distributed explicit state model checking," in *Parallel and Distributed Model Checking*, 2010.

[9] J. Bingham, J. Erickson, B. Bingham, and F. M. de Paula, "Open-source PReach," http://bitbucket.org/jderick/preach, 2013.

[10] U. Stern and D. L. Dill, "Parallelizing the murphi verifier," *Formal Methods in System Design*, vol. 18, no. 2, pp. 117–129, 2001.

[11] ——, "Improved probabilistic verification by hash compaction," in *Correct Hardware Design and Verification Methods, IFIP WG 10.5 Advanced Research Working Conference, CHARME '95*, 1995, pp. 206–224.

[12] P. Wolper and D. Leroy, "Reliable hashing without collision detection," in *IN COMPUTER AIDED VERIFICATION. 5TH INTERNATIONAL CONFERENCE*. Springer-Verlag, 1993, pp. 59–70.

[13] P. C. Dillinger and P. Manolios, "Bloom filters in probabilistic verification," in *Formal Methods in Computer-Aided Design*. Springer, 2004, pp. 367–381.

[14] Y. Kesten, A. Pnueli, and L. on Raviv, "Algorithmic verification of linear temporal logic specifications," in *Proc. 25th Int. Colloq. Aut. Lang. Prog., volume 1443 of Lect. Notes in Comp. Sci*. Springer-Verlag, 1998, pp. 1–16.

[15] I. Černá and R. Pelánek, "Distributed explicit fair cycle detection," in *Proc. SPIN workshop*, ser. LNCS, vol. 2648. Springer, 2003, pp. 49–74.

[16] X. Chen, Y. Yang, M. Delisi, G. Gopalakrishnan, and C.-T. Chou, "Hierarchical cache coherence protocol verification one level at a time through assume guarantee," in *High Level Design Validation and Test Workshop, 2007. HLVDT 2007. IEEE International*, 2007, pp. 107–114.

[17] J. Barnat, L. Brim, V. Havel, J. Havlíček, J. Kriho, M. Lenčo, P. Ročkai, V. Štill, and J. Weiser, "DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs," in *Computer Aided Verification (CAV 2013)*, ser. LNCS, vol. 8044. Springer, 2013, pp. 863–868.

[18] R. E. Tarjan, "Depth-first search and linear graph algorithms," *Siam Journal on Computing*, vol. 1, pp. 146–160, 1972.

[19] J. Barnat, L. Brim, and P. Ročkai, "A Time-Optimal On-the-Fly Parallel Algorithm for Model Checking of Weak LTL Properties," in *Formal Methods and Software Engineering (ICFEM 2009)*, ser. LNCS, vol. 5885. Springer, 2009, pp. 407–425.

[20] G. J. Holzmann, "Parallelizing the spin model checker," in *Proceedings of the 19th international conference on Model Checking Software*, ser. SPIN'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 155–171. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-31759-0_12

[21] A. W. Laarman, R. Langerak, J. C. van de Pol, M. Weber, and A. Wijs, "Multi-core nested depth-first search," in *Proceedings of the 9th International Symposium on Automated Technology for Verification and Analysis, ATVA 2011, Tapei, Taiwan*, ser. Lecture Notes in Computer Science, vol. 6996. London: Springer Verlag, July 2011, pp. 321–335.

[22] S. Evangelista, A. W. Laarman, L. Petrucci, and J. C. van de Pol, "Improved multi-core nested depth-first search," in *Proceedings of the 10th International Symposium on Automated Technology for Verification and Analysis, ATVA 2012, Thiruvananthapuram (Trivandrum), Kerala*, ser. Lecture Notes in Computer Science, vol. 7561. London: Springer Verlag, October 2012, pp. 269–283.