Scalable Dynamic Partial Order Reduction*

Jiri Simsa[1], Randy Bryant[1], Garth Gibson[1], and Jason Hickey[2]

[1] Carnegie Mellon University, Pittsburgh PA 15213, USA
[2] Google, Inc., Mountain View CA 94043, USA

Abstract. Systematic testing, first demonstrated in small, specialized cases 15 years ago, has matured sufficiently for large-scale systems developers to begin to put it into practice. With actual deployment come new, pragmatic challenges to the usefulness of the techniques. In this paper we are concerned with scaling dynamic partial order reduction, a key technique for mitigating the state space explosion problem, to very large clusters. In particular, we present a new approach for distributed dynamic partial order reduction. Unlike previous work, our approach is based on a novel exploration algorithm that 1) enables trading space complexity for parallelism, 2) achieves efficient load-balancing through time-slicing, 3) provides for fault tolerance, which we consider a mandatory aspect of scalability, 4) scales to more than a thousand parallel workers, and 5) is guaranteed to avoid redundant exploration of overlapping portions of the state space.

1 Introduction

Testing of concurrent programs is challenging because concurrency manifests as test non-determinism. A traditional approach to address this problem is stress testing, which repeatedly exercises concurrent operations of the program under test, hoping that eventually all concurrency scenarios of interest will be covered.

Unfortunately, as the scale of concurrent programs and the heterogeneity of environments in which these programs are deployed increases, the state space of possible scenarios explodes and stress testing stops being an effective mechanism for exercising all scenarios of interest.

To address the increasing complexity of software testing, researchers have turned their attention to systematic testing [8, 12, 14, 18, 19]. Similar to stress testing, systematic testing also repeatedly exercises concurrent operations of the program under test. However, unlike stress testing, systematic testing avoids test non-determinism by controlling the order in which concurrent operations happen, exercising different concurrency scenarios across different test executions.

* This research was sponsored by the U.S. Army Research Office under grant number W911NF0910273. The authors are also thankful to Google for providing its hardware and software infrastructure for the evaluation presented in this paper. Further, we also thank the members and companies of the PDL Consortium. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

To push the limits of systematic testing, existing tools combine of stateless exploration [8] with state space reduction [5, 7, 9] and parallel processing [21].

In this paper, we present a new method for distributed systematic testing of concurrent programs, which pushes the limits of systematic testing to an unprecedented scale. Unlike previous work [21], our approach is based on a novel exploration algorithm that 1) enables trading space complexity for parallelism, 2) achieves load-balancing through time-slicing, 3) provides for fault tolerance, which we consider a mandatory aspect of scalability, 4) scales to more than a thousand parallel workers, and 5) is guaranteed to avoid redundant exploration of overlapping portions of the state space.

The rest of the paper is organized as follows. Section 2 reviews stateless exploration, state space reduction, and parallel processing. Section 3 presents a novel exploration algorithm and details its use for distributed systematic testing at scale. Section 4 presents experimental evaluation of the implementation. Section 5 discusses related work and Section 6 presents the conclusions drawn from the results presented in this paper.

2 Background

In this section we give an overview of stateless exploration [8], dynamic partial order reduction (DPOR) [5], and distributed DPOR [21], which represent the state of the art in scalable systematic testing of concurrent programs.

2.1 Stateless Exploration

Stateless exploration is a technique that targets systematic testing of concurrent programs. The goal of stateless exploration is to explore the state space of different program states of a concurrent program by systematically enumerating different total orders in which concurrent events of the program can occur.

To keep track of the exploration progress, stateless exploration abstractly represents the state space of different program states using an *execution tree*. Nodes of the execution tree represent non-deterministic choice points and edges represent program state transitions. A path from the root of the tree to a leaf then uniquely encodes a program execution as a sequence of program state transitions.

Abstractly, enumeration of branches of the execution tree corresponds to enumeration of different sequences of program state transitions. Notably, the set of explored branches of a partially explored execution tree identifies which sequences of program state transitions have been explored. Further, assuming that concurrency is the only source of non-determinism in the program, the information collected by past executions can be used to generate schedules that describe in what order to sequence program state transitions of future executions in order to explore new parts of the execution tree.

Typically, stateless exploration uses depth-first search to explore the execution tree because of the space-efficient nature of its exploration, which is linear in the depth of the tree. Further, tools for stateless exploration such as VeriSoft [8] use partial order reduction (POR) [7] to avoid exploration of equivalent sequences of program state transitions.

The pseudocode depicted in Algorithm 1 and 2 gives a high-level overview of stateless exploration. The ExplorePor algorithm maintains an exploration *frontier*, represented as a stack of sets of nodes, and uses depth-first search to explore the execution tree. The PersistentSet(*node*) function uses static analysis to identify what subtrees of the execution tree need to be explored. In particular, it inputs a node of the execution tree and outputs a subset of the children of this node that need to be explored in order to explore all *non-equivalent* sequences of program state transitions of the execution tree. The details behind the computation of PersistentSet(*node*) are beyond the scope of this paper and can be found in Godefroid's seminal treatment [7]. Note that our presentation of [8] omits the use of sleep sets [7]. This simplification is made to achieve consistency with other techniques [5, 21] presented later in this section.

Algorithm 1 ExplorePor(*root*)

Require: A root node *root* of an execution tree.
Ensure: The execution tree rooted at the node *root* is explored.
1: *frontier* ← NewStack
2: Push({*root*}, *frontier*)
3: DFS-Por(*root*, *frontier*)

Algorithm 2 DFS-Por(*node*, *frontier*)

Require: A node *node* of an execution tree and a reference to a non-empty stack *frontier* of sets of nodes such that *node* ∈ Top(*frontier*).
Ensure: The node *node* of the execution tree is explored and the exploration frontier *frontier* is updated according to the POR algorithm.
1: remove *node* from Top(*frontier*)
2: **if** PersistentSet(*node*) ≠ ∅ **then**
3: Push(PersistentSet(*node*), *frontier*)
4: **for all** *child* ∈ Top(*frontier*) **do**
5: navigate execution to *child*
6: DFS-Por(*child*, *frontier*)
7: **end for**
8: Pop(*frontier*)
9: **end if**

2.2 Dynamic Partial Order Reduction

Dynamic partial order reduction (DPOR) is a technique that targets efficient state space exploration [5, 22]. The goal of DPOR is to further mitigate the combinatorial explosion of stateless exploration.

The stateless exploration discussed in the previous subsection uses static analysis to identify which subtrees of the execution tree need to be explored. However, precise static analysis of complex programs is often costly or infeasible and results in larger than necessary persistent sets. To address this problem, DPOR computes persistent sets using dynamic analysis.

When stateless exploration explores an edge of the execution tree, DPOR computes the happens-before [13] and the independence [7] relations over the

set of program state transitions. These two relations are then used to decide how to augment the existing exploration frontier.

The pseudocode depicted in Algorithm 3 and 4 gives a high-level overview of DPOR. The ExploreDpor algorithm maintains an exploration *frontier*, represented as a stack of sets of nodes, and uses depth-first search to explore the execution tree. The UpdateFrontier(*frontier*, *node*) function uses dynamic analysis to identify which subtrees of the execution tree need to be explored. In particular, the function inputs the current exploration frontier and the current node and computes the happens-before and independence relation between the transitions leading to the current node. This information is then used to infer which nodes need to be further added to the exploration frontier in order to explore all *non-equivalent* sequences of program state transitions. Importantly, the function modifies the exploration frontier in a *non-local* fashion as it can add nodes to an arbitrary set of the exploration frontier stack. The details behind the computation of UpdateFrontier(*frontier*, *node*) are beyond the scope of this paper and can be found in the original paper [5].

Algorithm 3 ExploreDpor(*root*)

Require: A root node *root* of an execution tree.
Ensure: The execution tree rooted at the node *root* is explored.

```
1: frontier ← NewStack
2: Push({root}, frontier)
3: Dfs-Dpor(root, frontier)
```

Algorithm 4 Dfs-Dpor(*node*, *frontier*)

Require: A node *node* of an execution tree and a reference to a non-empty stack *frontier* of sets of nodes such that *node* ∈ Top(*frontier*).
Ensure: The node *node* of the execution tree is explored and the exploration frontier *frontier* is updated according to the DPOR algorithm.

```
 1: remove node from Top(frontier)
 2: UpdateFrontier(frontier, node)
 3: if Children(node) ≠ ∅ then
 4:     child ← arbitrary element of Children(node)
 5:     Push({child}, frontier)
 6:     for all child ∈ Top(frontier) do
 7:         navigate execution to child
 8:         Dfs-Dpor(child, frontier)
 9:     end for
10:     Pop(frontier)
11: end if
```

2.3 Distributed Dynamic Partial Order Reduction

Distributed DPOR is a technique that targets concurrent stateless exploration. The goal of distributed DPOR is to offset the combinatorial explosion of possible permutations of concurrent events through parallel processing.

At a first glance, parallelization of DPOR seems straightforward: assign different parts of the execution tree to different *workers* and explore the execution

tree concurrently. However, as pointed out by Yang et al. [21], such a parallelization suffers from two problems. First, due to the non-local nature in which DPOR updates the exploration frontier, different workers may end up exploring identical parts of the state space. Second, since the sizes of the different parts of the execution tree are not known in advance, the load-balancing needed to enable linear speedup is non-trivial.

To address these two problems, Yang et al. [21] proposed two heuristics. Their first heuristic modifies Flanagan and Godefroid's lazy addition of nodes to the exploration frontier [5] so that they add nodes to the exploration frontier eagerly. As evidenced by their experiments, replacing lazy addition with eager addition mitigates the problem of redundant exploration of identical parts of the execution tree by different workers. Their second heuristic assumes the existence of a centralized load-balancer that workers can contact in case they believe they have too much work on their hands and would like to offload some of the work. The centralized load-balancer keeps track of which workers are idle and which workers are active and facilitates offloading of work from active to idle workers.

3 Scalable Dynamic Partial Order Reduction

While scaling distributed DPOR to a large cluster at Google [15], we have identified several problems with previous work of Yang et al. [21].

First, at large scale, the algorithm must explicitly cope with the failure of worker processes or machines. Although Yang et al. suggest how fault tolerance could be implemented, they do not quantify how their design affects scalability. Second, although the out-of-band centralized load-balancer of Yang et al. renders the communication overhead negligible, it precludes features that are enabled by centralized collection of information such as support for fault tolerance or state space size estimation. Third, the load-balancing of Yang et al. uses a heuristic based on a threshold to offload work from active to idle workers. It is likely that for different programs and different number of workers, different threshold values should be used. However, Yang et al. provide no insight into the problem of selecting a good threshold. Fourth, their DPOR modification for avoiding redundant exploration is a heuristic does not guarantee zero redundancy.

In this section we present an alternative design for distributed DPOR. Our design is centralized and uses a single master and n workers to explore the execution tree. Despite its centralized nature, our experiments show that our design scales to more than a thousand workers. Unlike previous work, our design can tolerate worker faults, is guaranteed to avoid redundant exploration, and is based on a novel exploration algorithm that allows 1) trading off space complexity for parallelism and 2) efficient load-balancing through time-slicing.

3.1 Novel Exploration Algorithm

The key advantage of using depth-first search for the purpose of DPOR is its favorable space complexity [8]. In fact, experience with systematic testing of concurrent programs based on stateless exploration [5, 14, 16, 19] suggests that the bottleneck for stateless exploration is CPU power, and not memory size.

To enable parallel processing, Yang et al. [21] depart from the strict depth-first search nature of stateless exploration. Instead, the execution tree is explored using a collection of (possibly overlapping) depth-first searches and the exploration order is determined by a load-balancing heuristic.

To overcome the limitations mentioned above, we have designed a novel exploration algorithm, called *n-partitioned depth-first search*, which relaxes the strict depth-first search nature of DPOR in a controlled manner and, unlike traditional depth-first search, is amenable to parallelization.

Algorithm 5 ExploreDpor(n, *root*)

Require: A positive integer n and a root node *root* of an execution tree.
Ensure: The execution tree rooted at the node *root* is explored.

```
 1: frontier ← NewSet
 2: Insert(Push({root}, NewStack), frontier)
 3: while Size(frontier) > 0 do
 4:     Partition(frontier, n)
 5:     fragment ← an arbitrary element of frontier
 6:     node ← an arbitrary element of Top(fragment)
 7:     Pdfs-Dpor(node, fragment, frontier)
 8:     if Size(fragment) = 0 then
 9:         Remove(fragment, frontier)
10:     end if
11: end while
```

Algorithm 6 Partition($frontier, n$)

Require: A non-empty set *frontier* of non-empty stacks of sets of nodes and a positive integer n such that $n \geq$ Size(*frontier*).
Ensure: Size(*frontier*) $= n$ or $\forall$*fragment* $\in$ *frontier* : the number of nodes contained in *fragment* is 1.

```
 1: for all fragment ∈ frontier do
 2:     if Size(frontier) = n then
 3:         return
 4:     end if
 5:     while the number of nodes contained in fragment is greater than 1 and
    Size(frontier) < n do
 6:         node ← an arbitrary element of a set contained in fragment
 7:         remove node from fragment
 8:         new-fragment ← a new frontier fragment for node
 9:         Insert(new-fragment, frontier)
10:     end while
11: end for
```

For the sake of the presentation, we first present a sequential version of DPOR based on the n-partitioned depth-first search. The main difference between depth-first search and n-partitioned depth-first search is that the exploration frontier of the new algorithm is partitioned into up to n frontier *fragments* and the new algorithm explores each fragment using a depth-first search interleaving exploration of different fragments.

The pseudocode depicted in Algorithm 5, 6, and 7 gives a high-level overview of DPOR algorithm based on the n-partitioned depth-first search. The algorithm maintains an exploration *frontier*, represented as a set of up to n stacks of sets of nodes. The elements of the exploration frontier are referred to as *fragments* and together they form a partitioning of the exploration frontier. The execution tree is explored by interleaving depth-first search exploration of frontier fragments. Algorithm 5 implements this idea by repeating two steps – PARTITION and PDFS-DPOR – until the execution tree is fully explored.

Algorithm 7 PDFS-DPOR(*node*, *fragment*, *frontier*)

Require: A node *node* of an execution tree, a reference to a non-empty stack *fragment* of sets of nodes such that *node* $\in$ TOP(*fragment*), and a reference to a set *frontier* of non-empty stacks of sets of nodes.

Ensure: The node *node* of the execution tree is explored and the fragment *fragment* of the exploration frontier is updated according to DPOR.

```
1: remove node from TOP(fragment)
2: UPDATEFRONTIER(frontier, fragment, node)
3: if CHILDREN(node) ≠ ∅ then
4:     child ← arbitrary element of CHILDREN(node)
5:     PUSH({child}, fragment)
6:     navigate execution to child
7: end if
8: pop empty sets from the fragment stack
```

The PARTITION step is detailed in Algorithm 6. During the PARTITION step, the current frontier is inspected to see whether existing frontier fragments should be and can be further partitioned. A new frontier fragment *should* be created in case there is less than n frontier fragments. A new frontier fragment *can* be created if there exists a frontier fragment with at least two nodes.

The PDFS-DPOR step is detailed in Algorithm 7. The PDFS-DPOR step is given one of the frontier fragments and uses depth-first search to explore the next edge of the subtree induced by the selected frontier fragment (the subtree that contains all ancestors and descendants of the nodes contained in the selected frontier fragment). The UPDATEFRONTIER(*frontier*, *fragment*, *node*) function operates in a similar fashion to the UPDATEFRONTIER(*frontier*, *node*) function described in the previous section. The main distinction is that after the function identifies which nodes are to be added to the exploration frontier using Flanagan and Godefroid's algorithm [5], these nodes are added to the current frontier fragment only if they are not already present in some other fragment. This way, the set of sets of nodes contained in each fragment remains a partitioning of the exploration frontier – an invariant maintained throughout our exploration that which helps our design to avoid redundant exploration.

3.2 Parallelization

In this subsection we describe how to efficiently parallelize the above sequential DPOR design based on n-partitioned depth-first search.

First, observe that the presence or absence of the Partition step in the body of the main loop of the ExploreDpor function of Algorithm 5 has no effect on the correctness of the algorithm. This allows us to sequence several Pdfs-Dpor steps together, which hints at possible distribution of the exploration.

Namely, one could spawn concurrent workers and use them to carry out sequences of Pdfs-Dpor steps over different frontier fragments. However, a straightforward implementation of this idea would require synchronization when concurrent workers access and update the exploration frontier, which is shared by all workers. The trick to overcome this obstacle to efficient parallelization is to give each worker a private copy of the execution tree. As pointed out by Yang et al. [21], such a copy can be concisely represented using the state of the depth-first search stack of the frontier fragment to be explored.

Algorithm 8 ExploreDistributedDpor(n, *budget*, *root*)

Require: A positive integer n, a time budget *budget* for worker exploration, and a root node *root* of an execution tree.
Ensure: The execution tree rooted at the node *root* is explored.

```
 1: frontier ← NewSet
 2: Insert(Push(root, NewStack), frontier)
 3: while Size(frontier) > 0 do
 4:     Partition(frontier, n)
 5:     while exists an idle worker and an unassigned frontier fragment do
 6:         fragment ← an arbitrary unassigned element of frontier
 7:         Spawn(ExploreLoop, fragment, budget, ExploreCallback)
 8:     end while
 9:     wait until signaled by ExploreCallback
10: end while
```

Algorithm 9 ExploreLoop(*fragment*, *budget*)

Require: A non-empty stack *fragment* of sets of nodes.
Ensure: Explores previously unexplored branches of the subtree induced by the nodes of *fragment* until all branches are explored or the timeout expires.

```
1: start-time ← GetTime
2: repeat
3:     node ← an arbitrary element of Top(fragment)
4:     Pdfs-Dpor(node, fragment)
5:     current-time ← GetTime
6: until current-time − start-time > budget or Size(fragment) = 0
```

A worker can then repeatedly invoke the Pdfs-Dpor function over (a copy of) the assigned frontier fragment. Once the worker either completes the exploration of the assigned frontier fragment or it exceeds a time budget allocated for its exploration, it reports back with the results of the exploration. The exploration progress can be concisely represented using the original and the final state of the depth-first search stack of the assigned frontier fragment.

The pseudocode depicted in Algorithm 8 and 9 presents a high-level approximation of the actual implementation of our distributed DPOR. The implementation operates with the concept of "fragment assignment". When a frontier

fragment is created, it is unassigned. Later, a fragment becomes assigned to a particular worker through the invocation of the Spawn function. When the worker finishes its exploration, or exhausts the time budget assigned for exploration, it reports back the results, the fragment assigned to this worker becomes unassigned again. The results of worker exploration are mapped back to the "master" copy of the execution tree using the ExploreCallback callback function. The time budget for worker exploration is used to achieve load-balancing through time-slicing. The Partition function behaves identically to the original one, except for the fact that it partitions unassigned fragments only.

Algorithm 9 presents the pseudocode of the ExploreLoop function, which is executed by a worker. The Pdfs-Dpor function is identical to the sequential version of the algorithm. The workers are started through the Spawn function which creates a private copy of a part of the execution tree. Notably, the copy contains only the nodes that the worker needs to further the exploration of the assigned frontier fragment. Structuring the concurrent exploration in this fashion enables both multi-threaded and multi-process implementations.

Since our goal has been to scale the stateless exploration to thousands of workers, the scale of clusters available today, our implementation implements each worker as an RPC server running as a separate process. In such a setting, the Spawn function issues an asynchronous RPC request that triggers invocation of the ExploreLoop function with the appropriate arguments at the RPC server of the worker. The response to the RPC request is then handled asynchronously by the ExploreCallback function, which maps the results of the worker exploration into the master copy of the execution tree and resumes execution of the main loop of Algorithm 8.

3.3 Fault Tolerance

As is commonly done in large distributed applications [4, 6], failure of one out of thousands of nodes must be anticipated and handled gracefully, but failure of just one particular node is infrequent enough to be dealt with using re-execution. In accordance with this practice, our design assumes that the master, which is running the ExploreDistributedDpor function, will not fail. The workers on the other hand are allowed to fail and the exploration can tolerate such events.

In particular, an RPC request issued by the master to a worker RPC server uses a deadline to decide whether the worker has failed. The value of the deadline is set commensurately to the value of the worker time budget.

When the deadline expires without an RPC response arriving, the master simply assumes that the worker has failed and makes no changes to the frontier fragment originally assigned to the failed worker. The fragment becomes unassigned again and other workers get a chance to further its exploration.

3.4 Load-balancing

The key to high utilization of the worker fleet is effective load-balancing. To achieve load-balancing, our design time-slices frontier fragments among available workers. The availability of frontier fragments is impacted by two factors.

The first factor is the upper bound n on the number of frontier fragments that the ExploreDistributedDpor creates. This parameter determines the size of the pool of available work units. The higher this number, the higher the memory requirements of the master but the higher the opportunity for parallelism. In our experience, setting n to twice the number of workers worked fairly well. Future work on dynamic selection of the number of workers might be beneficial if the impact on the parallelism and memory use can be managed.

The second factor is the size of the time slice used for worker exploration. Smaller time slices lead to more frequent generation of new fragments but this elasticity comes at the cost of higher communication overhead. In our initial design we used a fixed time budget, choosing the value of 10 seconds as a good compromise for the elasticity vs. communication overhead trade-off. However, the initial evaluation of our prototype made us realized that a variable time budget improves worker fleet utilization at large scales.

In particular, we observed that as the number of workers increases, a gap between the realized and the ideal speed up opens up. Our investigation identified time periods during the exploration with insufficient number of frontier fragments to keep all workers busy.

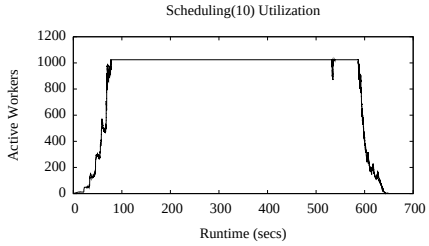


Fig. 1. Without Optimizations

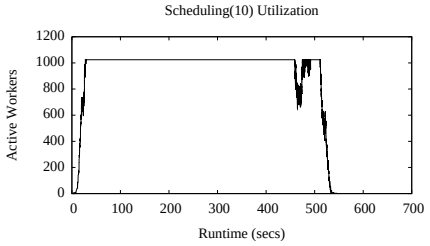


Fig. 2. With Optimizations

To study this problem, we recorded the number of active workers during the lifetime of a test. Figure 1 plots this information for one of our test programs on a configuration with $1,024$ workers and an upper bound of $2,048$ frontier fragments. The figure is representative of other measurements at such a scale.

One can identify three phases of the exploration. In the first phase, the number of active workers gradually increases over 100 seconds until there is enough frontier fragments to keep everyone busy. In the second phase, all workers are kept busy. In the third phase, the number of active workers gradually decreases to zero over 100 seconds. Ideally, the first and the third phase should be as short as possible in order to minimize the inefficiency resulting from not fully utilizing the available worker fleet.

To this aim, we have developed a technique based on a variable time budget. In particular, if the exploration is configured to use a time budget b, the master actually uses fractions of b proportional to the number of active workers. For example, the first worker will receive a budget of $\frac{b}{n}$, where n is the number of workers. When half of the workers are active, the next worker to be assigned work will receive a budget of $\frac{b}{2}$. The scaling of the time budget is intended

to reduce the time before the master gets the opportunity to re-partition and load-balance and thus to reduce the duration of the first and the third phase.

We implemented this technique and re-ran the our scalability measurements. For comparison with Figure 1, Figure 2 plots the number of active workers over time for the optimized implementation. For this test program, the two techniques reduced the runtime from 655 seconds to 527 seconds. Similar runtime improvements have been achieved for other test programs.

3.5 Avoiding Redundant Exploration

For clarity of presentation, Algorithm 8 omits a provision that prevents concurrent workers from exploring overlapping portions of the execution tree. This could happen when two workers make concurrent UpdateFrontier calls and add identical nodes to their frontier fragment copies.

To avoid this problem, our implementation introduces the concept of "node ownership". A worker exclusively owns a node if it is contained in the original frontier fragment currently assigned to the worker, or is a descendant of a node that the worker owns. All other nodes are assumed to be shared with other workers and the node ownership restricts which nodes a worker may explore.

In particular, the depth-first search exploration of a worker is allowed to operate only over nodes that the worker owns. When it encounters a shared node during its exploration, the worker terminates its exploration and sends an RPC response to the master indicating which nodes of the final frontier fragment are shared. The ExploreCallback function checks which newly discovered shared nodes are already part of some other frontier fragment. If a newly discovered node is not part of some other fragment, the node is added to the master copy of the currently processed frontier fragment (ownership is claimed). Otherwise, the ownership of the node has been already claimed and the node is not added to the master copy of the currently processed frontier fragment.

Although this provision could in theory lead to increased communication overhead and decreased worker fleet utilization, our experiments indicate that in practice the provision does not affect performance.

4 Evaluation

To evaluate our design, we implemented its prototype on top of ETA [15], a tool developed at Google used for systematic testing of multi-threaded components of a cluster management system. These components are written using a library based on the actors paradigm [1] and the ETA tool is used to systematically enumerate different total orders in which messages between actors can be delivered in order to exercise different concurrency scenarios.

4.1 Experimental Setup

For the purpose of evaluation of our implementation we have used instances of the three following tests. The Resource(x,y) test is representative of a class of actor program tests that evaluate interactions of x different users that acquire and release resources from a pool of y resources. The Store(x,y,z) test is representative of a class of actor program tests that evaluate interactions

of x users of a distributed key-value store with y front-end nodes and z back-end nodes. The SCHEDULING(X) test is representative of a class of actor program tests that evaluate interactions of x users issuing concurrent scheduling requests. These tests exercise fundamental functionality of core components of the cluster management system and are part of the unit test suite of the system.

Unless stated otherwise, each measurement presented in the remainder of this section presents a run of a complete exploration of the given test and the results report the mean and the standard deviation of three repetitions of a run. Lastly, all experiments were carried out on a Google data center using stock hardware and running each process on a separate virtual machine.

4.2 Faults

First, we evaluated the ability of the implementation to handle worker failures. Notably, we extended the ETA tool with an option to inject an RPC fault with a certain probability. When an RPC fault is injected, the master fails to receive the RPC response from a worker and waits for the RPC to timeout instead.

Our experiments with injected RPC faults have demonstrated that the runtime increases proportionally to the underlying geometric progression (of repeated RPC failures). For example, if each RPC had a 50% chance of failing, the runtime doubled. Since in actual deployments of ETA, RPC requests fail with probability well under 1%, our support for fault tolerance is practical.

4.3 Scalability

Next, to measure the scalability of the implementation, we compared the time needed to complete an exploration by a sequential implementation of DPOR against the time needed to complete the same exploration by our distributed implementation. We considered configurations with 32, 64, 128, 256, 512, and $1,024$ workers and applied the algorithm to the RESOURCE(6,6), STORE(12,3,3), and SCHEDULING(10) actor program tests, parameters of which were chosen to stimulate interesting state space sizes.

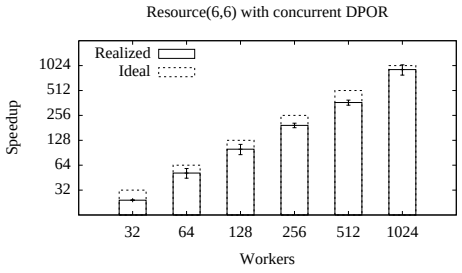
These experiments were run inside of a dynamically shared cluster; that is, machines running worker processes are shared with other workloads. The time budget of each worker exploration was set to 10 seconds and the target number of frontier fragments was set to twice the number of workers.

The results of RESOURCE(6,6), STORE(12,3,3), and SCHEDULING(10) experiments are presented in Figure 3, Figure 4, and Figure 5 respectively. Due to the magnitude of the state spaces being explored, the runtime of the sequential algorithm was extrapolated using a partial run. The figures visualize the speedup over the extrapolated runtime of the sequential algorithm and compare it to the ideal speedup. Note that both axes of the graphs are in logarithmic scale.

These results evidence the scalability of our implementation of DPOR at a large scale. The largest configuration uses $1,024$ workers and our implementation achieves speedup that ranges between $760\times$ and $920\times$.

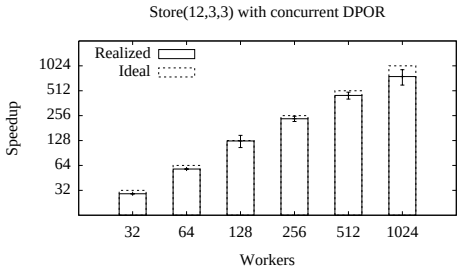
4.4 Theoretical Limits

Finally, we carried out measurements that helped us to evaluate the theoretical scalability limits of our implementation. The purpose of the section is to project



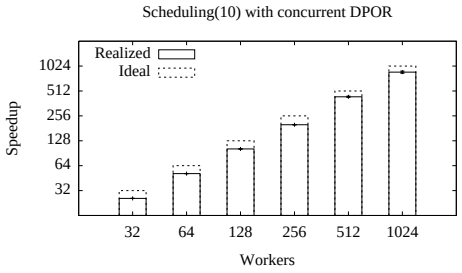
Conf	Runtime	Speedup
32 Workers	31,105s	24×
64 Workers	14,681s	51×
128 Workers	7,557s	100×
256 Workers	3,874s	194×
512 Workers	2,051s	366×
1024 Workers	820s	916×

Fig. 3. For this example, DPOR explores on the order of 18.5 million branches and the sequential implementation is expected to require 209 hours to finish.



Conf	Runtime	Speedup
32 Workers	26,646s	29×
64 Workers	13,396s	58×
128 Workers	6,252s	126×
256 Workers	3,332s	233×
512 Workers	1,784s	434×
1024 Workers	1,022s	758×

Fig. 4. For this example, DPOR explores on the order of 21 million branches and the sequential implementation is expected to require 215 hours to finish.



Conf	Runtime	Speedup
32 Workers	17,707s	26×
64 Workers	8,870s	51×
128 Workers	4,468s	102×
256 Workers	2,278s	200×
512 Workers	1,046s	436×
1024 Workers	527s	865×

Fig. 5. For this example, DPOR explores on the order of 3.6 million branches and the sequential implementation is expected to require 126 hours to finish.

future bottlenecks. To this aim we focused on measuring memory and CPU requirements of the master.

Memory Requirements: The memory overhead of our implementation is dominated by the cost to store the master copy of the exploration frontier. To estimate the overhead, we measured the amount of memory allocated for the explicitly stored nodes of the execution tree and the exploration frontier data structures over time. For the Scheduling(10) test on a configuration with $1,024$ workers and an upper bound of $2,048$ frontier fragments, the peak number of the allocated memory was less than 4MB. This number is representative of results for other tests at such a scale. Consequently, for the current computer architectures, the memory requirements scale to millions of workers.

CPU requirements: With 1024 workers and a 10-second time budget, the master is expected to issue around 100 RPC requests and to process around 100 RPC responses every second. For such a load, the stock hardware running exclusively the master process experiences peak CPU utilization under 20%. Consequently, for the current computer architectures, the CPU requirements scale to around $5,000$ workers. To scale our implementation beyond that, one can scale the time budget, hardware performance, or optimize the software stack. For instance, one could replace the single master with a hierarchy of masters. The performance of our algorithm shows that hierarchical organization is not needed to scale to the size of state of the art cluster.

5 Related Work

Concurrent state space exploration have been previously studied in the context of several projects: **Inspect** [20] is a tool for systematic testing of `pthreads` C programs that implements the distributed DPOR [21] discussed in Section 2. Unlike our work, the Inspect tool does not support fault tolerance, is not guaranteed to avoid redundant exploration, and has not been demonstrated to scale beyond 64 workers. **DeMeter** [10] provides a framework for extending existing sequential model checkers [12, 19] with a parallel and distributed exploration engine. Similar to our work, the framework focuses on efficient state space exploration of concurrent programs. Unlike our work, the design has not been thoroughly described or analyzed and has been demonstrated to scale only up to 32 workers. **Cloud9** [3] is a parallel engine for symbolic execution of sequential programs. In comparison to our work, the state space being explored is the space of all possible programs inputs. Systematic enumeration of different program inputs is an orthogonal problem to the one addressed by this paper. Parallelization of software verification was also investigated in the context of explicit state space model checkers in tools such as **MurPhi** [17], **DiVinE** [2], or **SWARM** [11]. Stateful exploration is less common for implementation-level model checkers [8, 15, 19] where storing a program state explicitly becomes prohibitively expensive.

6 Conclusions

This paper presented a technique that improves the state of the art of scalable techniques for systematic testing of concurrent programs. Our design for distributed DPOR enables the exploitation of a large-scale cluster for the purpose of systematic testing. At the core of the design lies a novel exploration algorithm, n-partitioned depth-first search, which has proven to be essential for scaling our design to thousands of workers.

Unlike previous work [21], our design provides support for fault tolerance, a mandatory aspect of scalability, and is guaranteed to avoid redundant exploration of identical parts of the state space by different workers. Further, our implementation and deployment of a real-world system at scale has demonstrated that the design achieves almost linear speed up for up to $1,024$ workers. Lastly, we carried out a theoretical analysis of the design to identify scalability bottlenecks of the design.

References

1. G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA, 1986.
2. J. Barnat, L. Brim, M. Češka, and P. Ročkai. DiVinE: Parallel Distributed Model Checker (Tool paper). In *HiBi/PDMC 2010*, pages 4–7. IEEE, 2010.
3. S. Bucur, V. Ureche, C. Zamfir, and G. Candea. Parallel symbolic execution for automated real-world software testing. In *EuroSys 2011*, pages 183–198, 2011.
4. F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. BigTable: A distributed storage system for structured data. In *OSDI 2006*, pages 205–218, 2006.
5. C. Flanagan and P. Godefroid. Dynamic Partial Order Reduction for Model Checking Software. *SIGPLAN Not.*, 40(1):110–121, 2005.
6. S. Ghemawat, H. Gobioff, and S. Leung. The Google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.
7. P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem.* Springer-Verlag, 1996.
8. P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *POPL 1997*, pages 174–186. ACM, 1997.
9. G. Gueta, C. Flanagan, E. Yahav, and M. Sagiv. Cartesian Partial-Order Reduction. In *SPIN 2007*, pages 95–112. Springer-Verlag, 2007.
10. H. Guo, M. Wu, L. Zhou, G. Hu, J. Yang, and L. Zhang. Practical software model checking via dynamic interface reduction. In *SOSP 2011*, pages 265–278, New York, NY, USA, 2011. ACM.
11. Gerard J. Holzmann, Rajeev Joshi, and Alex Groce. Swarm Verification Techniques. *IEEE Transactions on Software Engineering*, 37:845–857, 2011.
12. C. E. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *NSDI 2007*, 2007.
13. L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, 1978.
14. M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. In *OSDI 2008*, pages 267–280, 2008.
15. J. Simsa, R. Bryant, G. Gibson, and J. Hickey. Efficient Exploratory Testing of Concurrent Systems. *CMU-PDL Technical Report*, 113, November 2011.
16. J. Simsa, G. Gibson, and R. Bryant. dBug: Systematic Evaluation of Distributed Systems. In *SSV 2010*, 2010.
17. Ulrich Stern and David L. Dill. Parallelizing the MurPhi Verifier. *Formal Methods in System Design*, 18(2):117–129, 2001.
18. S. S. Vakkalanka, S. Sharma, G. Gopalakrishnan, and R. M. Kirby. ISP: A tool for model checking MPI programs. In *PPoPP 2008*, pages 285–286, 2008.
19. J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. MoDist: Transparent Model Checking of Unmodified Distributed Systems. In *NSDI 2009*, pages 213–228, April 2009.
20. Y. Yang, X. Chen, and G. Gopalakrishnan. Inspect: A Runtime Model Checker for Multithreaded C Programs. *University of Utah Tech. Report*, UUCS-08-004, 2008.
21. Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby. Distributed dynamic partial order reduction based verification of threaded software. In *SPIN 2007*, pages 58–75. Springer-Verlag, 2007.
22. Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby. Efficient Stateful Dynamic Partial Order Reduction. In *SPIN 2008*, pages 288–305. Springer-Verlag, 2008.