Concurrent Systematic Testing at Scale

Jiri Simsa, Randy Bryant, Garth Gibson, Jason Hickey (Google)

CMU-PDL-12-101
May 2012

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

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 comes new, pragmatic challenges to the usefulness of the techniques. In this report 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, 4) has been demonstrated to scale to more than a thousand parallel workers, and 5) is guaranteed to avoid redundant exploration of overlapping portions of the state space.

Acknowledgements: This research was sponsored by the U.S. Army Research Office under grant number W911NF0910273. 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. Further, the authors would like to thank Brian Grant, John Wilkes, Robert Kennedy, Todd Wang, and Walfredo Cirne for their continuous technical feedback and support on the ongoing ETA effort. The authors are also thankful to Google for providing its hardware and software infrastructure for the experimental evaluation presented in this report. Last but not least, we also thank the members and companies of the PDL Consortium (including Actifio, American Power Conversion, EMC Corporation, Emulex, Facebook, Fusion-io, Google, Hewlett-Packard Labs, Hitachi, Huawei Technologies Co., Intel Corporation, Microsoft Research, NEC Laboratories, NetApp, Inc., Oracle Corporation, Panasas, Riverbed, Samsung Information Systems America, Seagate Technology, STEC, Inc., Symantec Corporation, VMware, Inc., and Western Digital) for their interest, insights, feedback, and support.

Keywords: Systematic Testing, Stateless Exploration, State Space Reduction, Parallel Processing

1 Introduction

Testing of concurrent systems is challenging because concurrency manifests as test non-determinism. A traditional approach to address this problem is stress testing. Stress testing repeatedly exercises concurrent operations of the system under test, hoping that sooner or later all concurrency scenarios (and errors) of interest will be covered.

Unfortunately, as the scale of concurrent systems and the heterogeneity of environments in which these systems 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 [7, 10, 12, 13, 17, 18]. Similar to stress testing, systematic testing also repeatedly exercises concurrent operations of the system 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.

To scale systematic testing to ever more complex programs, existing tools use a combination of stateless exploration [7] paired with state space reduction [5, 6] and parallel processing [20].

In this report, we present a new method for concurrent systematic testing at scale, which pushes the limits of systematic testing to an unprecedented scale. Unlike previous work on the topic [20], 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, 4) has been demonstrated to scale 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 report 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 concurrent systematic testing at scale. Section 4 presents the experimental evaluation of the implementation. Lastly, Section 5 discusses related work and Section 6 presents the conclusions drawn from the results presented in this report.

2 Background

In this section we give an overview of stateless exploration [7], dynamic partial order reduction [5], and distributed dynamic partial order reduction [20]. Together these techniques represent the state of the art in scalable systematic testing of concurrent systems.

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 the use of depth-first search results in space-efficient exploration. Further, tools for stateless exploration such as VeriSoft [7] use partial order reduction [6] to avoid exploration of equivalent sequences of program state transitions.

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: $frontier \leftarrow$ 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 $node \in$ TOP(*frontier*).
Ensure: The previously unexplored node *node* of the execution tree is explored and the exploration frontier *frontier* is updated according to the partial order reduction algorithm.

```
1: remove $node$ from TOP($frontier$)
2: if PERSISTENTSET($node$) $\neq \emptyset$ then
3:     PUSH(PERSISTENTSET($node$),$frontier$)
4:     for all $child \in$ TOP($frontier$) do
5:         navigate execution to $child$
6:         DFS-POR($child$,$frontier$)
7:     end for
8:     POP($frontier$)
9: end if
```

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 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 report and can be found in [6]. Note that unlike the presentation in [7], our presentation omits the use of sleep sets [6]. This omission is to achieve consistency with the techniques presented in the remainder of the report. The sleep sets can be added as described in [7].

2.2 Dynamic Partial Order Reduction

Dynamic partial order reduction is a technique that targets efficient stateless exploration. The goal of dynamic partial order reduction is to further mitigate the combinatorial explosion resulting from systematic enumeration of different total orders of concurrent events through the use of dynamic analysis.

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 which results in larger than necessary persistent sets. To address this problem, dynamic partial order reduction computes persistent sets using dynamic analysis.

When stateless exploration explores an edge of the execution tree, dynamic partial order reduction computes the happens-before [11] and the independence [6] relations over the set of program state transitions. These two relations are then used to augment the existing exploration frontier based on the newly discovered information.

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(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 previously unexplored node node of the execution tree is explored and the exploration frontier
    frontier is updated according to the dynamic partial order reduction algorithm.
 1: remove node from TOP(frontier)
 2: UPDATEPERSISTENTSETS(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
```

The pseudocode depicted in Algorithm 3 and 4 gives a high-level overview of dynamic partial order reduction. 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 UPDATEPERSISTENTSETS(*frontier*, *node*) function uses dynamic analysis to identify what subtrees of the execution tree need to be explored. In particular, the function inputs the current exploration frontier and the current node and infers which nodes need to be further added to the exploration frontier in order to explore all non-equivalent sequences of program state transitions of the execution tree. Importantly, the UPDATEPERSISTENTSETS(*frontier*, *node*) 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 UPDATEPERSISTENTSETS(*frontier*, *node*) are beyond the scope of this report and can be found in [5].

2.3 Distributed Dynamic Partial Order Reduction

Distributed dynamic partial order reduction is a technique that targets concurrent stateless exploration. The goal of distributed dynamic partial order reduction is to offset the combinatorial explosion resulting from systematic enumeration of different total orders of concurrent events through parallel processing.

At a first glance, parallelization of dynamic partial order reduction seems straightforward: assign different parts of the execution tree to different *workers* and let the workers explore the execution tree concurrently. However, as pointed out in [20], such a straightforward parallelization suffers from two problems. First, due to the non-local nature in which dynamic partial order reduction 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, effective load balancing is needed to enable linear speed up.

To address these two problems, the authors of [20] proposed two heuristics. Their first heuristic modifies dynamic partial order reduction so that it adds nodes to the exploration frontier eagerly instead of lazily, which was the case in [5]. As evidenced by the experiments of [20], introduction of this heuristic seems to avoid 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 it. 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 dynamic partial order reduction to a very large cluster inside of Google [14], we have identified several problems with the distributed dynamic partial order reduction of [20]:

1. At a very large scale, a failure of a worker is not a question of if but a question of when. Although [20] suggests how fault tolerance could be implemented, it does not implement it.

2. Although the mostly decentralized nature of [20] renders the communication overhead negligible, it complicates addition of features that are facilitated by centralized collection of information such as support for fault tolerance or state space size estimation.

3. The load balancing of the original design 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, [20] provides little insight into the problem of selecting a good threshold.

4. The dynamic partial order reduction modification used for avoiding redundant exploration in [20] is a heuristic and provides no guarantee that different workers will in fact explore disjoint portions of the execution tree.

In this section we present an alternative to [20]. In comparison, our design is more centralized and uses a single master and *n* workers to explore the execution tree. Despite the more centralized nature of the design, our experiments show that it scales to more than a thousand workers. Further, unlike [20], the new 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 dynamic partial order reduction is its favorable space complexity [7]. In fact, experience with systematic testing of concurrent programs based on stateless exploration [5, 13, 15, 18] suggests that for current computer architectures stateless exploration is bottle-necked by its CPU (and not memory) requirements. This is hardly surprising given that the worst-case time complexity of stateless exploration is linear in the size of the execution tree, while its space complexity is linear in the depth of the execution tree.

To enable parallel processing, the authors of [20] 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 of [20] mentioned above, we have designed a novel exploration algorithm, called *n-partitioned depth-first search*, which relaxes the strict depth-first search nature of dynamic partial order reduction in a controlled manner and, unlike traditional depth-first search, is amenable to parallelization.

For the sake of the presentation, we first present a sequential version of the dynamic partial order reduction based on *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.

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(n, frontier)
 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 ≥ SIZE(frontier).
Ensure: SIZE(frontier) = n or ∀fragment ∈ frontier : the sum of sizes of all sets 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 sum of sizes of all sets 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
```

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 previously unexplored node node of the execution tree is explored and the fragment fragment
    of the exploration frontier is updated according to the dynamic partial order reduction algorithm.
 1: remove node from TOP(fragment)
 2: UPDATEPERSISTENTSETS(frontier,fragment,node)
 3: if CHILDREN(node) $\neq \emptyset$ then
 4:     child $\leftarrow$ arbitrary element of CHILDREN(node)
 5:     PUSH($\{child\}$,fragment)
 6:     navigate execution to child
 7: end if
 8: while TOP(fragment) $= \emptyset$ do
 9:     POP(fragment)
10: end while
```

The pseudocode depicted in Algorithm 5, 6, and 7 gives a high-level overview of dynamic partial order reduction algorithm based on *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 constitute 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.

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 divided in order to increase the number of frontier fragments. 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 (a subtree of the execution tree that contains all ancestors and descendants of the nodes contained in the selected frontier fragment). The UPDATEPERSISTENTSETS(*frontier*,*fragment*,*node*) function operates in a similar fashion to the UPDATEPERSISTENTSETS(*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 the algorithm of [5], it adds these nodes to the current frontier fragment unless they are already present in some other frontier fragment. This way, the set of sets of nodes contained in each frontier fragments remains a partitioning of the set of nodes of the exploration frontier – an invariant that is maintained throughout the exploration.

3.2 Parallelization

In this subsection we describe how the above, seemingly sequential and inefficient, implementation of dynamic partial order reduction based on *n*-partitioned depth-first search can be efficiently parallelized.

First, observe that the presence or absence of the PARTITION step in the body of the main loop of the EXPLOREDPOR function has no effect on the correctness of the algorithm. This allows us to string 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 [20], such a copy can be concisely represented using the depth-first search stack state of the frontier fragment to be explored.

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 the time budget allocated for its exploration, it reports back with the results of the exploration. The results can be again concisely represented using the original and the final state of the depth-first search stack of the assigned frontier fragment.

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: $frontier \leftarrow$ NEWSET
 2: INSERT(PUSH($root$, NEWSTACK), $frontier$)
 3: while SIZE($frontier$) $> 0$ do
 4:     PARTITION($n$, $frontier$)
 5:     while exists an idle worker and an unassigned frontier fragment do
 6:         $fragment \leftarrow$ 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 8 presents the pseudocode of EXPLOREDISTRIBUTEDDPOR function, which approximates the actual implementation of our distributed dynamic partial order reduction. 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 and 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 PARTITION function behaves identically to the original one, except for the fact that it partitions unassigned fragments only.

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 or all branches are explored.
 1: $start\text{-}time \leftarrow$ GETTIME
 2: repeat
 3:     $node \leftarrow$ an arbitrary element of TOP($fragment$)
 4:     PDFS-DPOR($node$, $fragment$)
 5:     $current\text{-}time \leftarrow$ GETTIME
 6: until $current\text{-}time - start\text{-}time > budget$ or SIZE($fragment$) $= 0$
```

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 of EXPLOREDISTRIBUTEDDPOR.

Since our goal has been to scale the stateless exploration to thousands workers, our implementations actually 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

Our design and implementation assumes that the master process running EXPLOREDISTRIBUTEDDPOR is a reliable service and will not fail. The worker fleet on the other hand is not assumed to be a reliable and the exploration can recover from worker failures.

In particular, an RPC request issued by the master to a worker RPC server uses a deadline to decide whether the worker has failed or not. The value of the deadline is usually set to whatever the time budget assigned to the worker was plus several seconds to account for communication overhead and transient failures.

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 and other workers get a chance to further its exploration.

3.4 Load-balancing

The key to high utilization of the worker fleet is to create enough fragments so that each worker can contribute towards the exploration of the execution tree. There are two factors that impact the availability of unassigned fragments.

First factor 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 imposes reasonable space overhead while ensuring that workers that happen to complete exploration of a frontier fragment can be promptly assigned a new fragment.

Second factor is the size of the time budget used for worker exploration. Smaller time budgets lead to more frequent generation of new fragments but this elasticity comes at the cost of higher communication overhead. Although originally our implementation supported only fixed time budget values, the initial evaluation made us realize that a variable time budget can increase the worker fleet utilization at large scales. We discuss the benefits of using variable time budget in more detail in Section 4.

3.5 Avoiding Redundant Exploration

For clarity of presentation the description of EXPLOREDISTRIBUTEDDPOR omitted a provision that prevents concurrent workers from exploring overlapping portions of the execution tree. This could happen when two concurrent workers make calls to UPDATEPERSISTENTSETS 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 already owns. All other nodes are assumed to be shared with other workers and the node ownership is used to restrict 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.

Thus, if two concurrent workers make calls to UPDATEPERSISTENTSETS and add identical nodes to their frontier fragment copies, they won't be allowed to explore a potentially shared subtree. Instead, they will be forced to inform the master of their intention to do so and the master will assign the ownership of the subtree to one of the respective frontier fragments.

Although this provision could in theory lead to premature termination of worker exploration – causing increased communication overhead and decreased worker fleet utilization – our experiments indicate that in practice the provision does not affect the performance.

4 Evaluation

To evaluate our design of scalable dynamic partial order reduction, we implemented its prototype on top of ETA [14]. ETA is a tool developed at Google used for systematic testing of multi-threaded components that are part of the next generation of Google's 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. These tests exercise fundamental functionality of certain components of the cluster management system and are part of the unit test suite of the system.

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. Each resource is represented as an actor that expects `Acquire()` and `Release()` messages and maintains a boolean flag that denotes availability of the resource. When an `Acquire()` message is received and the resource is available, the resource is granted to the sender of the `Acquire()` message. If the resource is unavailable, the actor representing the resource queues the request and produces no immediate response. When a `Release()` message is received and there are no outstanding acquisition requests, the resource becomes available. Otherwise, the resource is used to satisfy one of the outstanding acquisition requests. Each user is also represented as an actor that acquires and releases resource according to its internal logic. A test passes when all users are able to acquire all resources they requested.

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. Each back-end node is represented as an actor that owns part of the key range and expects `Put(k,v)` and `Get(k)` messages. Each front-end node is represented as an actor that routes access requests to appropriate back-ends nodes. The front-end actor expects `PutRequest(k,v)`, `PutReply()`, `GetRequest(k)` and `GetReply(v)` messages.

Finally, each user is also represented as an actor that generates `PutRequest(k,v)` and `GetRequest(k)` messages and expects `PutReply()` and `GetReply(v)` messages according to its internal logic. A test passes when all get requests return the value set by the latest put request.

The SCHEDULING(X) test is representative of a class of actor program tests that evaluate interactions of *x* users issuing concurrent scheduling requests. Each user is represented as an actor that generates `Schedule(r)` messages that detail the nature of the scheduling request. Further, each actor program contains a single scheduling coordinator actor that expects `Schedule(r)` messages and determines whether the request can be serviced given a local snapshot of the global state of the cluster resources. A test passes when the set of scheduled requests is satisfiable from the available cluster resources.

Unless stated otherwise, each measurement presented in the remainder of this section was repeated three times and the results report the mean and the standard deviation of these measurements. Lastly, all experiments were carried out inside of a Google data center [8] using stock hardware and running each process on a separate machine.

4.2 Time Budget Selection

First, we focused on determining the effects of using different values of a fixed worker time budget. For the purpose of this evaluation, we measured the runtime of several test instances while ranging the value of a fixed time budget for the worker exploration.

Figures 1, 2, and 3 detail the results of these experiments. These experiments used a configuration with 32 workers and the upper bound on the number of frontier fragments was set to 64.

The results of the experiments validate the intuition mentioned in the previous section. Namely, a small time budget (1 second) is not sufficient to amortize the communication cost over useful work done by the workers, while a large time budget (50 seconds) leads to infrequent creation of new fragments that can cause unnecessary idling of workers. For the tests used in this evaluation, a 10-second time budget seemed to be a good compromise between the two conflicting trends on the size of the budget.

4.3 Faults

Next, we evaluated the ability of the implementation to handle worker failures. For the purpose of this evaluation, we extended the ETA tool with an option to simulate an RPC fault with a certain probability. When an RPC fault is simulated, the master ignores the RPC response from a worker and waits for the RPC deadline to trigger instead. These experiments used a time budget of 10 seconds, a configuration with 32 workers, and the upper bound on the number of frontier fragments was set to 64.

Figures 4, 5, and 6 detail the results of these experiments. For each test instance we present two graphs: one that visualizes how runtime changes with changing fault probability and one that visualizes the number of simulated RPC faults.

The results demonstrate that even if 10% RPC requests were to fail due to temporary worker or network failures, the impact on the exploration runtime would be negligible. In actual deployments of ETA, RPC requests fail with less than 1% probability, which implies that our support for fault tolerance is practical.

4.4 Scalability

Next, to measure the scalability of the implementation, we compared the time needed to complete an exploration by a sequential implementation of dynamic partial order reduction against the time needed to complete the same exploration by our distributed implementation.

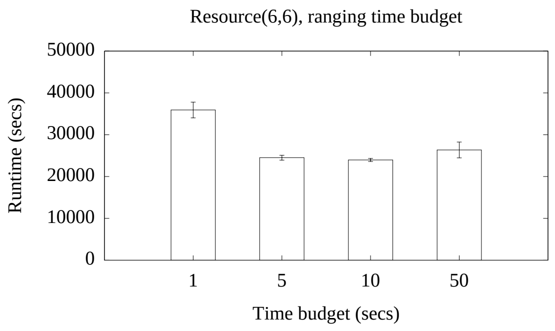


Figure 1: RESOURCE(6,6)

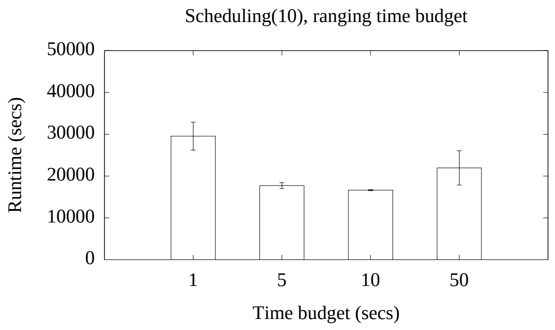


Figure 2: SCHEDULING(10)

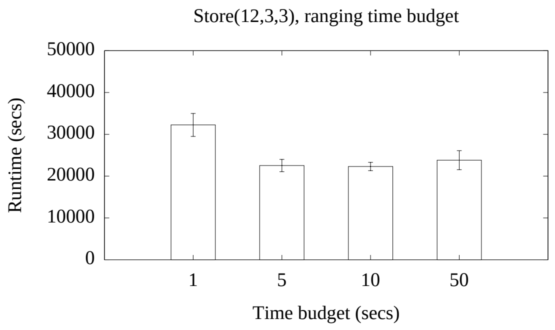


Figure 3: STORE(12,3,3)

4.4.1 Small Scale

First, we considered configurations with 1, 2, 4, 8, 16, 32, and 64 workers and examined the RESOURCE(5,5), STORE(11,3,3), and SCHEDULING(9) actor program tests. These experiments were run inside of a homogeneous cluster, with each worker having exclusive access to its machine. 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(5,5), STORE(11,3,3), and SCHEDULING(9) experiments are presented in Figure 7, Figure 8, and Figure 9 respectively. The figures visualize the speedup over the sequential algorithm and compare it to the ideal speedup. Note that both axes of the graphs are in logarithmic scale.

These results illustrate the scalability of our concurrent implementation of dynamic partial order reduc-

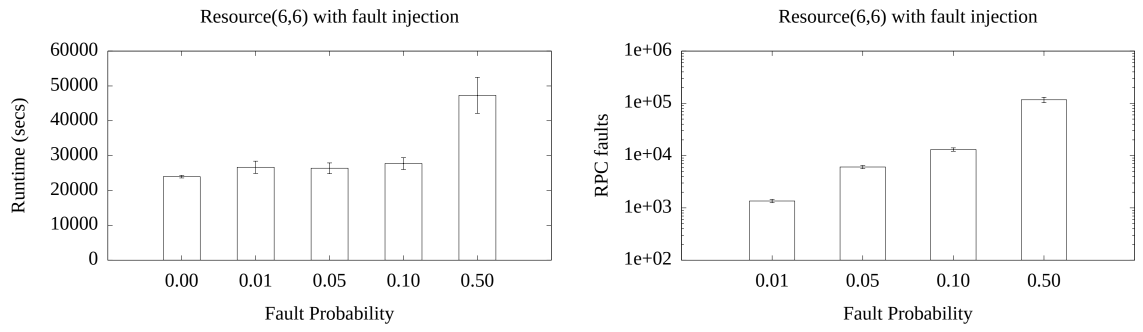


Figure 4: RESOURCE(6,6) with fault injection

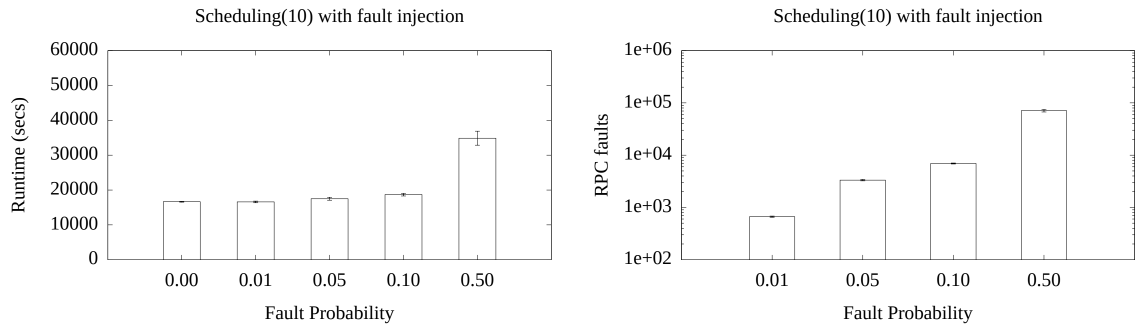


Figure 5: SCHEDULING(10) with fault injection

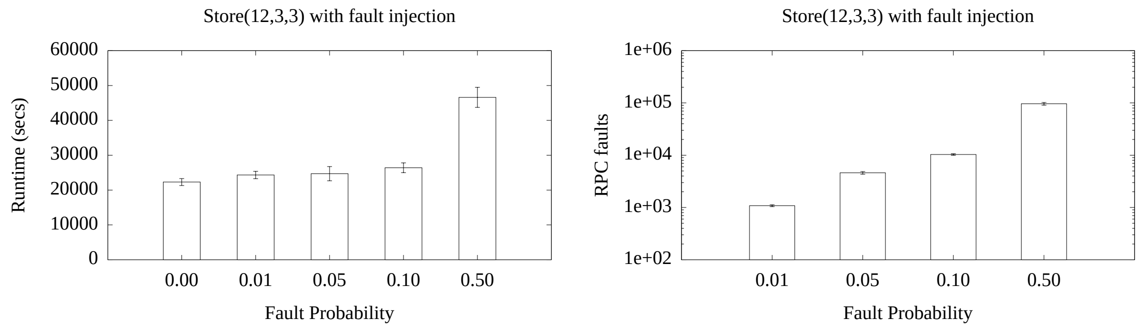


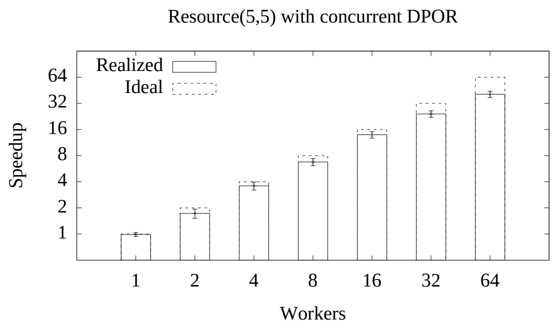
Figure 6: STORE(12,3,3) with fault injection

tion at a small scale. The largest configuration uses 64 workers and our implementation achieves a modest speedup that ranges between $40.81\times$ and $52.78\times$.

4.4.2 Large Scale

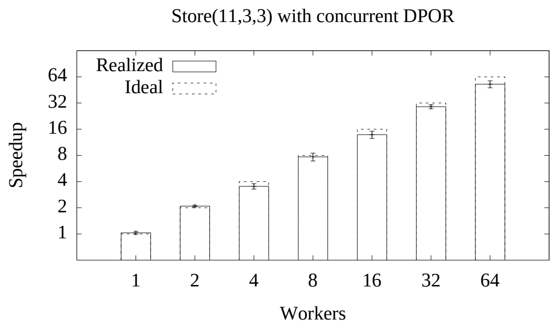
Second, we considered configurations with 32, 64, 128, 256, 512, and $1,024$ workers and applied the algorithm to the RESOURCE(6,6), STORE(12,3,3), and SCHEDULING(10) actor program tests. These experiments were run inside of a heterogeneous cluster, sharing the worker machines with other workloads. The 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



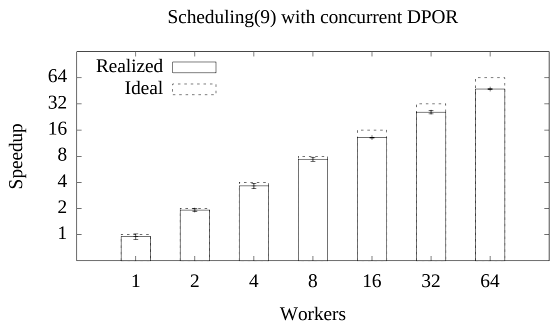
CONF	RUNTIME(S)	SPEEDUP
1 Worker	14,681.67	0.99
2 Workers	8,408.00	1.73
4 Workers	4,050.00	3.59
8 Workers	2,148.00	6.77
16 Workers	1,043.00	13.92
32 Workers	600.00	24.18
64 Workers	355.67	40.81

Figure 7: For this example, the sequential implementation of dynamic partial order reduction explores 461,504 branches and requires 4 hours to finish.



CONF	RUNTIME(S)	SPEEDUP
1 Worker	81,284.33	1.03
2 Workers	39,859.67	2.09
4 Workers	23,686.00	3.53
8 Workers	10,865.00	7.70
16 Workers	6,030.67	13.87
32 Workers	2,879.00	29.06
64 Workers	1,584.67	52.78

Figure 8: For this example, the sequential implementation of dynamic partial order reduction explores 2,766,228 branches and requires 23.2 hours to finish.

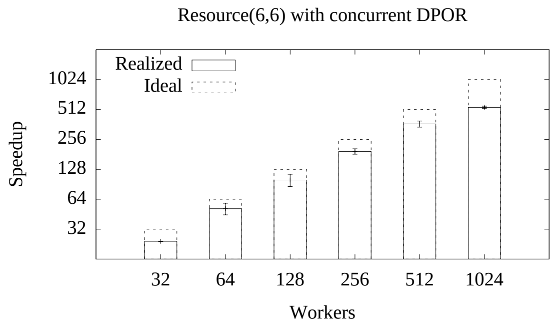


CONF	RUNTIME(S)	SPEEDUP
1 Worker	32,417.00	0.95
2 Workers	16,035.00	1.92
4 Workers	8,480.00	3.64
8 Workers	4,161.33	7.39
16 Workers	2,343.33	13.11
32 Workers	1,194.33	25.73
64 Workers	646.67	47.50

Figure 9: For this example, the sequential implementation of dynamic partial order reduction explores 362,880 branches and requires 8.5 hours to finish.

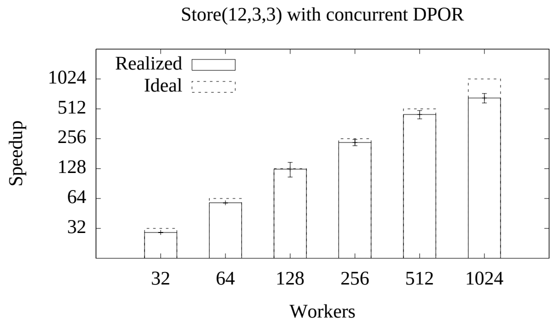
Figure 10, Figure 11, and Figure 12 respectively. Due to the magnitude of the state spaces being explored, the runtime required by the sequential algorithm to explore these state spaces was extrapolated using the speedup measurements from the previous subsection. The figures visualize the speedup over 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 illustrate the scalability of our concurrent implementation of dynamic partial order reduction at a large scale. The largest configuration uses $1,024$ workers and our implementation achieves a modest speedup that ranges between $538.49\times$ and $696.21\times$.



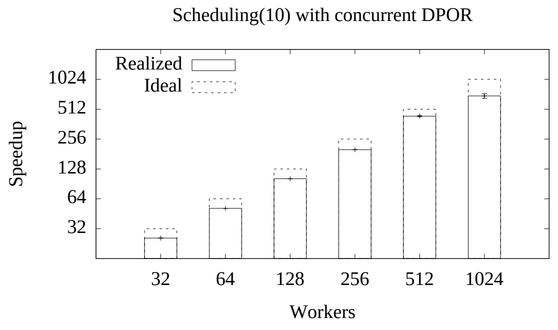
CONF	RUNTIME(S)	SPEEDUP
32 Workers	31,105.00	24.18
64 Workers	14,681.67	51.42
128 Workers	7,557.33	99.98
256 Workers	3,874.67	193.88
512 Workers	2,051.67	366.30
1024 Workers	1,396.33	538.49

Figure 10: For this example, dynamic partial order reduction explores on the order of 18.5 million branches and the sequential implementation is expected to require 209 hours to finish.



CONF	RUNTIME(S)	SPEEDUP
32 Workers	26,646.00	29.06
64 Workers	13,396.33	57.80
128 Workers	6,252.33	126.09
256 Workers	3,332.00	233.28
512 Workers	1,784.00	434.54
1024 Workers	1,269.33	610.61

Figure 11: For this example, dynamic partial order reduction explores on the order of 21 million branches and the sequential implementation is expected to require 215 hours to finish.



CONF	RUNTIME(S)	SPEEDUP
32 Workers	17,707.33	25.73
64 Workers	8,870.67	51.36
128 Workers	4,468.33	101.96
256 Workers	2,278.33	199.98
512 Workers	1,046.00	435.67
1024 Workers	655.67	696.21

Figure 12: For this example, dynamic partial order reduction explores on the order of 3.6 million branches and the sequential implementation is expected to require 126 hours to finish.

4.5 Improving Utilization

Although our initial scalability experiments achieved decent speedup, we observed that as the number of workers increases a gap between the realized and the ideal speed up opens up. Our next step was to understand where does this gap come from.

To this end we identified two factors that impact the realized speedup: 1) the heterogeneous nature of the large scale cluster, 2) time periods with insufficient number of frontier fragments to keep all workers busy.

The study of the former factor is beyond the scope of the report. We simply acknowledge that different workers might have different relative speeds and thus the speed up measurements presented here should be taken with a grain of salt.

To study the impact of the latter factor, we repeated some of our measurements, recording the number of active workers over time. Figure 13 plots this information for the SCHEDULING(10) test on a configuration with $1,024$ workers and an upper bound of $2,048$ frontier fragments. The figure is representative of all other measurements at such a scale.

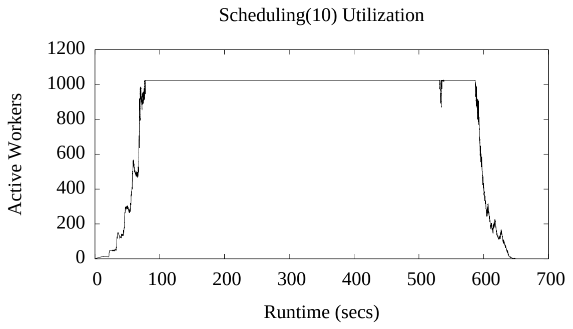


Figure 13: SCHEDULING(10) Without Optimizations

Note that one can identify three phases of the exploration. In the first phase, as the exploration starts unfolding the execution tree, the number of active workers gradually increases until there is enough frontier fragments to keep everyone busy. In the second phase, all workers are kept busy. Finally, in the third phase, as the exploration is nearing its end, the number of active workers gradually decreases all the way to zero.

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. In part, the nature of the first and the third phase depends on factors that we assume are fixed such as the program under test and the number of workers used for the exploration. Thus, a complete elimination of these two phases is impossible. However, we have developed two techniques that are expected to reduce the length of these two phases.

First, we employ a variable time budget. In particular, if the exploration is configured to use a time budget b, the master actually uses fractions of b instead in proportion 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 and 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 accelerate creation of new frontier fragments when possible and thus reduce the duration of the first and the third phase.

Second, we employ a technique that is intended to minimize the risk that the last few frontier fragments need to be re-explored because of worker failures. In particular, as soon as the implementation believes it has reached the third phase, it starts assigning each fragment to multiple workers. Note that this optimization will cause multiple workers exploring overlapping portions of the state space. Since the workers would be otherwise idle, we do not consider this redundancy as a source of inefficiency.

We implemented these two techniques and re-ran the our scalability measurements for the configuration with $1,024$ workers. For comparison with Figure 13, Figure 14 plots the number of active workers over time for the optimized implementation. For this example, the two techniques reduced the runtime from 655 seconds to 527 seconds, increasing the realized speed up from $696\times$ to $865\times$. For RESOURCE(6,6) and STORE(12,3,3), the two techniques improved realized speed up from $538\times$ to $916\times$ and from $610\times$ to $759\times$ respectively.

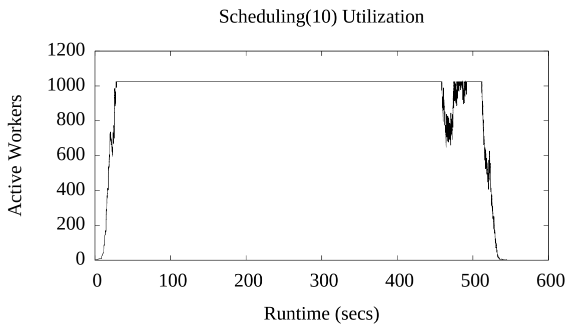


Figure 14: SCHEDULING(10) With Optimizations

4.6 Theoretical Limits

Finally, we carried out measurements that helped us evaluate the theoretical scalability limits of our implementation. To this aim we focused on measuring the memory and CPU requirements of the master.

Memory Requirements The memory overhead of our implementation is dominated by the cost to store the exploration frontier of the master copy of the execution. To approximate the memory needed to record the exploration frontier, we measured the number of explicitly stored nodes of the execution tree over time. Figure 15 plots this information for the SCHEDULING(10) test on a configuration with $1,024$ workers and an upper bound of $2,048$ frontier fragments. The graph of Figure 15 is representative of other measurements at such a scale. Further, the space complexity of our implementation is linear with respect to the number of nodes, requiring less than 100 bytes per node. Thus, for the experiment plotted in Figure 15 the memory overhead of our implementation is is less than 4MB. Consequently, for the current computer architectures, the memory requirements of our implementation allow it to scale to hundreds of thousands of workers and frontier fragments.

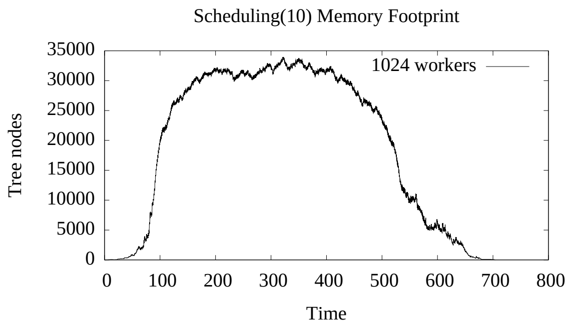


Figure 15: SCHEDULING(10) memory requirements

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 was experiencing CPU utilization under 20%. Consequently, for a 10-second time budget, our implementation is expected to scale to around $5,000$ workers on the current hardware and software infrastructure. To scale our implementation beyond that, one could proportionally scale the time budget, hardware performance, or optimize the software stack.

5 Related Work

Concurrent state space exploration have been previously studied in the context of several projects:

- Inspect [19] is a tool for systematic testing of `pthreads` C programs that implements the distributed dynamic partial order reduction [20] 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 [9] provides a framework for extending existing sequential model checkers, such as [10, 18], 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 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 in the context of [3] is the space of all possible programs inputs. Systematic enumeration of different program inputs is an orthogonal problem to the one addressed by this report.

- DiVinE [2] is a parallel and distributed explicit state LTL model checker. Unlike the technique presented here, DiVinE uses a stateful approach to state space exploration, storing different program states explicitly. Stateful exploration is less common for implementation-level model checkers, such as [7, 14, 18], where the cost of storing a program state explicitly becomes prohibitively expensive.

- In [4], the authors presented a parallel algorithm that explores the state space using a number of independent randomized depth-first searches to decrease the time needed to locate an error. In comparison, our parallelization of systematic testing aims to cover the full state space faster.

- Lastly, the work of [16] targets parallelization of symbolic execution using randomness and static partitioning. The parallelization presented in this report is systematic and uses dynamic partitioning.

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 dynamic partial order reduction 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 proved to be an essential building stone for scaling our design to thousands of workers.

Unlike previous work, our design provides support for fault tolerance and is guaranteed to avoid redundant exploration of identical parts of the state space by different workers. Further, a thorough evaluation of a prototype implementation of the design has demonstrated that the design achieves almost linear speed up for up to $1,024$ workers. Finally, a theoretical analysis of the potential bottlenecks of the design suggest that the design could be scaled up to hundreds of thousands of workers.

References

[1] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.

[2] J. Barnat, L. Brim, I. Černá, P. Moravec, P. Ročkai, and P. Šimeček. DiVinE – A Tool for Distributed Verification (Tool Paper). In *Computer Aided Verification*, volume 4144/2006 of *LNCS*, pages 278–281. Springer Berlin / Heidelberg, 2006.

[3] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. Parallel symbolic execution for automated real-world software testing. In *EuroSys*, pages 183–198, 2011.

[4] Matthew B. Dwyer, Sebastian Elbaum, Suzette Person, and Rahul Purandare. Parallel randomized state-space search. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 3–12, Washington, DC, USA, 2007. IEEE Computer Society.

[5] Cormac Flanagan and Patrice Godefroid. Dynamic Partial Order Reduction for Model Checking Software. *SIGPLAN Not.*, 40(1):110–121, 2005.

[6] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *LNCS*. Springer, 1996.

[7] Patrice Godefroid. Model Checking for Programming Languages using VeriSoft. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 174–186. ACM, 1997.

[8] Google Data Centers. http://www.google.com/about/datacenters/, 2011.

[9] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. Practical software model checking via dynamic interface reduction. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 265–278, New York, NY, USA, 2011. ACM.

[10] Charles Edwin Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *NSDI '07: Proceedings of the 5th Conference on USENIX Symposium on Networked Systems Design and Implementation*, 2007.

[11] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, 1978.

[12] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A pragmatic approach to model checking real code. In *OSDI '02: Proceedings of the 5th Conference on USENIX Symposium on Operating Systems Design and Implementation*, 2002.

[13] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. In *OSDI '08: Proceedings of the 8th Conference on USENIX Symposium on Operating Systems Design and Implementation*, pages 267–280, 2008.

[14] Jiri Simsa, Randy Bryant, Garth Gibson, and Jason Hickey. Efficient Exploratory Testing of Concurrent Systems. *PDL-CMU Technical Report*, 113, November 2011.

[15] Jiri Simsa, Garth Gibson, and Randy Bryant. dBug: Systematic Evaluation of Distributed Systems. In *SSV '10: Proceedings of 5th International Workshop on System Software Verification*, 2010.

[16] Matt Staats and Corina Păsăreanu. Parallel symbolic execution for structural test generation. In *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, pages 183–194, New York, NY, USA, 2010. ACM.

[17] Sarvani S. Vakkalanka, Subodh Sharma, Ganesh Gopalakrishnan, and Robert M. Kirby. ISP: A tool for model checking MPI programs. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 285–286, 2008.

[18] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MoDist: Transparent Model Checking of Unmodified Distributed Systems. In *NSDI '09: Proceedings of the Sixth Symposium on Networked Systems Design and Implementation*, pages 213–228, April 2009.

[19] Yu Yang, Xiaofang Chen, and Ganesh Gopalakrishnan. Inspect: A Runtime Model Checker for Multithreaded C Programs. *University of Utah Technical Report*, UUCS-08-004, 2008.

[20] Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M. Kirby. Distributed dynamic partial order reduction based verification of threaded software. In *Proceedings of the 14th international SPIN conference on Model checking software*, pages 58–75, Berlin, Heidelberg, 2007. Springer-Verlag.