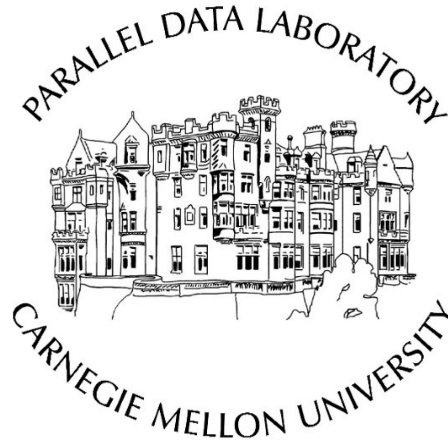


Efficient Exploratory Testing of Concurrent Systems

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

CMU-PDL-11-113
November 2011

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

Abstract

In our experience, exploratory testing has reached a level of maturity that makes it a practical and often the most cost-effective approach to testing. Notably, previous work has demonstrated that exploratory testing is capable of finding bugs even in well-tested systems [4, 17, 24, 23]. However, the number of bugs found gives little indication of the efficiency of a testing approach. To drive testing efficiency, this paper focuses on techniques for measuring and maximizing the coverage achieved by exploratory testing.
In particular, this paper describes the design, implementation, and evaluation of Eta, a framework for exploratory testing of multi-threaded components of a large-scale cluster management system at Google. For simple tests (with millions to billions of possible executions), Eta achieves complete coverage one to two orders of magnitude faster than random testing. For complex tests, Eta adopts a state space reduction technique to avoid the need to explore over 85% *of executions and harnesses parallel processing to explore multiple test executions concurrently, achieving a throughput increase of up to* $17.5\times$.

Acknowledgements: The authors would like to thank Brian Grant, John Wilkes, Robert Kennedy, Swapnil Patil, Vijay Vasudevan, and Walfredo Cirne for their technical feedback. Further, the authors are 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 APC, EMC, Facebook, Google, Hewlett-Packard, Hitachi, IBM, Intel, LSI, Microsoft, NEC, NetApp, Oracle, Panasas, Riverbed, Samsung, Seagate, STEC, Symantec, and VMware) for their interest, insights, feedback, and support.

Keywords: Exploratory Testing, State Space Exploration, State Space Size Estimation, State Space Reduction, Parallel Processing

1 Introduction

The performance of high-end computing environments has been experiencing an exponential increase, doubling roughly every 15 months [15]. This has been due to a shift to symmetric multiprocessing as well as massive parallelization through distributed computing. At the same time, the software infrastructure that provides access to the raw hardware has undergone a similar transition; from sequential code to multi-threaded and distributed code.

The increased performance has come at the cost of increased concurrency and complexity of the hardware layer. With hardware failures unavoidable at large scale [19], the software layer must tolerate them, which further increases its complexity. As the software layer becomes more concurrent and complex, it gives software engineers more opportunities to err and the errors they make are harder to deal with.

In particular, testing of concurrent systems is challenging because concurrency exacerbates test non-determinism and leads to combinatorial explosion of the number of possible executions. *Exploratory testing*, pioneered by Godefroid [8], is a promising approach that tackles test non-determinism stemming from concurrency by controlling the order in which concurrent events in the system happen and uses this mechanism to enumerate different behaviors of the system. At the same time, exploratory testing lends itself amenable to methods for mitigation of the combinatorial explosion.

In our experience, exploratory testing has reached a level of maturity that makes it a practical and often the most cost-effective approach to testing. Notably, previous work has demonstrated that exploratory testing is capable of finding bugs even in well-tested systems [4, 17, 24, 23]. However, the number of bugs found gives little indication of the efficiency of a testing approach. To drive testing efficiency, this paper focuses on techniques for measuring and maximizing the coverage achieved by exploratory testing.

The goal of this paper is to answer the following questions: 1) is exploratory testing more efficient than stress testing, 2) what else besides finding bugs can exploratory testing be used for, 3) can the progress of an exploratory test be approximated, and 4) what techniques can help to scale exploratory testing.

To this end, the paper presents the design, implementation, and evaluation of Eta, a framework developed for testing of multi-threaded components of a large-scale cluster management system at Google. For simple tests (with millions to billions of possible executions), Eta achieves complete coverage one to two orders of magnitude faster than random testing. For complex tests, Eta adopts a state space reduction technique to avoid the need to explore over 85% of executions and harnesses parallel processing to explore multiple test executions concurrently, achieving a throughput increase of up to 17.5×.

Contributions The paper presents a number of novel contributions: 1) a space-efficient algorithm for systematic enumeration of different executions of a concurrent program, 2) a time-efficient online algorithm for estimation of the number of different executions of a concurrent program, 3) a distributed algorithm for systematic enumeration of different executions of a concurrent program, and 4) an in-depth evaluation of the exploratory approach to testing of concurrent programs.

Organization The rest of the paper is organized as follows. Section 2 introduces a model of the concurrent programs tested by Eta. Section 3 compares different approaches to testing of concurrent programs. Section 4 presents a novel technique for estimation of the number of different executions of a concurrent program. Section 5 describes an adaptation of a state space reduction technique and evaluates its performance. Section 6 describes a novel concurrent state space exploration algorithm and evaluates its performance. Section 7 describes the related work and Section 8 draws conclusions.

2 Actors

We start with an informal description of *actors* [1] and then proceed to define a model that provides an abstract representation of the Google actors library. Programs written in this library are the target of Eta.

2.1 Informal Description

Conceptually, an *actor program* consists of a set of actors, each of which has its own private state, a public queue for receiving messages, and a set of handlers for processing of queued messages. The only way two actors are allowed to communicate is by sending messages to one another. Further, each individual actor is sequential, repeatedly invoking handlers to process queued messages. Processing of a message is assumed never to block indefinitely and can both modify the private state of the actor processing the message and push new messages on queues of other actors. An actor program realizes concurrency by concurrently executing multiple actors.

The actors paradigm can be used to implement both one-shot computations and reactive systems. A one-shot computation implemented as an actor program typically starts by inserting a set of initial messages, the inputs, and then letting the actors repeatedly invoke their handlers until there are no messages left to be processed (i.e., the program has reached a quiescent state). When the actors paradigm is used to represent a reactive system, an actor program typically contains one or more *adapters*. Adapters are similar to actors except that they also interface the actor program with the environment in which it runs. For example, an adapter can expose a remote procedure call interface to the environment by translating the actor program messages to remote procedure call requests / responses and vice versa.

2.2 Formal Definitions

Definition 2.1. An *n-actor program* $\mathbb{M}$ is a tuple of actors $(A_1,\ldots,A_n)$, where for $i \in 1,\ldots,n$ an *actor* A_i is in turn a tuple (Q_i,L_i,Δ_i) consisting of a set Q_i of queues, a set L_i of local states, and a transition function $\Delta_i : \mathit{Message} \times L_i \to (Q_1 \times \ldots \times Q_n) \times L_i$. A *queue* $q_i \in Q_i$ is a finite, and possibly empty, sequence $(m_1,\ldots,m_k) \in \mathit{Message}^k$ of messages and for $j \in \{1,\ldots,k\}$ we say the message m_j is *contained* in the queue q_i.

Note that the above definition leaves several terms undefined. The set *Message* is assumed to be an abstract representation of the set of messages actors use to communicate. In particular, the inputs and outputs of the actor program are encoded as messages. The sets $L_1,\ldots,L_n$ are assumed to be an abstract representation of the private state of each actor. Lastly, the functions $\Delta_1,\ldots,\Delta_n$ are assumed to be an abstract representation of the handlers invoked to process queued messages. Next, we define a state of an actor program as a vector of local states and queues of actors of an actor program.

Definition 2.2. Let $\mathbb{M}$ be an n-actor program. We define a *state of* $\mathbb{M}$ to be a tuple $((q_1,l_1),\ldots,(q_n,l_n)) \in (Q_1 \times L_1) \times \ldots \times (Q_n \times L_n)$. Further, a state of $\mathbb{M}$ can be inspected using the following projections:

- $\mathit{actor} : ((Q_1 \times L_1) \times \ldots \times (Q_n \times L_n)) \times \{1,\ldots,n\} \to \bigcup_{i=1}^{n}(Q_i \times L_i)$, which is a function such that: $\mathit{actor}(((q_1,l_1),\ldots,(q_n,l_n)),i) = (q_i,l_i)$
- $\mathit{queue} : \bigcup_{i=1}^{n}(Q_i \times L_i) \to \bigcup_{i=1}^{n} Q_i$, which is a function such that $\mathit{queue}(q_i,l_i) = q_i$
- $\mathit{local} : \bigcup_{i=1}^{n}(Q_i \times L_i) \to \bigcup_{i=1}^{n} L_i$, which is a function such that $\mathit{local}(q_i,l_i) = l_i$

Next, we define the transition function of an actor program as a combination of the transition functions of actors of an actor program.

Definition 2.3. Let $\mathbb{M}$ be an n-actor program. We define the *transition function of* $\mathbb{M}$ to be a function $\Delta_{\mathbb{M}} : (Q_1 \times L_1) \times \ldots \times (Q_n \times L_n) \times \{1,\ldots,n\} \rightarrow (Q_1 \times L_1) \times \ldots \times (Q_n \times L_n)$ such that $\Delta_{\mathbb{M}}((q_1,l_1),\ldots,(q_n,l_n),i)$ is undefined if $q_i = \emptyset$ and equals to $((q'_1,l'_1),\ldots,(q'_n,l'_n))$ otherwise. The values $(q'_1,l'_1),\ldots,(q'_n,l'_n)$ are related to the values $(q_1,l_1),\ldots,(q_n,l_n)$ and i as follows. If $\Delta_i(head(q_i),l_i) = ((q''_1,\ldots,q''_n),l''_i)$, then the following holds:

- $l'_i = l''_i$ and $l'_j = l_j$ for all $j \neq i$
- $q'_i = tail(q_i) \circ q''_i$ and $q'_j = q_j \circ q''_j$ for all $j \neq i$

Remark 2.4. If Δ_i is well-defined for $i \in \{1,\ldots,n\}$, then $\Delta_{\mathbb{M}}$ is well-defined.

In other words, an actor program transitions from one state to another by choosing an actor with a non-empty queue, popping a message from that queue, and invoking a handler on that message to process it. By the nature of the definition, the invoked handler is only allowed to make changes to the private state of the actor processing the message and push messages to queues of other actors. Notably, the invoked handler cannot change the private state of any actor other than the actor processing the message.

Next, we define an execution of an actor program as a sequence of states and identify this sequence with a sequence of indices that determine in what order the execution processes messages.

Definition 2.5. Let $\mathbb{M}$ be an n-actor program, $\Delta_{\mathbb{M}}$ its transition function, and s its state. We define an *execution of* $\mathbb{M}$ from s to be a finite sequence $\alpha = (s_0,\ldots,s_k)$ of states of $\mathbb{M}$ such that $s = s_0$ and there exists an *index sequence* $I(\alpha) = (i_0,\ldots,i_{k-1}) \in \{1,\ldots,n\}^k$ such that $s_{j+1} = \Delta_{\mathbb{M}}(s_j,i_j)$ for all $j \in \{0,\ldots,k-1\}$.

Remark 2.6. If α is an execution, then $I(\alpha)$ is unique.

Lastly, we formally express what it means for one state to be reachable from another state.

Definition 2.7. Let $\mathbb{M}$ be an n-actor program and s its state. We say that a state s' is *reachable* from the state s if there exists an execution $\alpha = (s_0,\ldots,s_k)$ such that $s = s_0$ and $s' = s_k$ and we use $\mathbb{R}(\mathbb{M},s)$ to denote the set of all states of $\mathbb{M}$ reachable from the state s.

The following section presents algorithms that input an actor program and a test exercising some behavior of the program, and enumerate the actor program states reachable through different executions of the test.

3 Testing of Actors

In this section we describe and evaluate several algorithms for testing of actor programs. For the purpose of the presentation, we will assume that the order in which messages are handled in an actor program is controlled by an *actor manager*. To optimize for performance in deployment, an actor manager can be implemented as a collection of threads, one for each actor, that concurrently remove messages from message queues and invoke respective message handlers. However, in testing, we replace this *concurrent* actor manager with an actor manager that enables a fine-grained control of message sequencing instead. Such an approach to testing relies on the assumption that the different actor managers used in deployment and in testing are functionally identical.

We describe and compare three actor managers: a *deterministic* actor manager, a *random* actor manager, and an *exploratory* actor manager. All three actor managers serialize and record the order in which messages are handled. Under the assumption that this order is the only source of non-determinism, this provides for deterministic replay and, in the case of the exploratory actor manager, a mechanism for systematic enumeration of possible message sequences. In practice, this assumption can be met by making all

other, potentially non-deterministic, interactions with the environment deterministic; for instance, by using deterministic seeds for pseudo-random number generators or by mocking remote procedure call servers. Alternatively, the assumption can be relaxed / lifted by using a more general *event manager* that controls several / all sources of non-determinism.

The three actor managers differ in the policy they use for deciding which message is handled next. The deterministic actor manager simply round-robins between the non-empty message queues of the program. The random actor manager selects a non-empty message queue uniformly at random. The exploratory actor manager keeps a history of decisions made in past executions and steers future executions towards yet unexplored orders in which messages are handled.

The motivation for the deterministic actor manager is its simplicity and reproducibility. The obvious disadvantage of such an actor manager is that a repeated execution of a test only ever explores one order in which messages are handled. The random actor manager tries to alleviate this problem by using chance to determine in what order messages will be handled. With minimum modification to the deterministic actor manager, repeated execution is expected to explore different behaviors. However, using a random actor manager has its limits. For instance, a random actor manager gives us no indication as to how many times we need to run a test to achieve satisfactory coverage, or does it indicate if the random actor manager is exploring executions that are (substantially) different.

The purpose of re-executing a non-deterministic test of an actor program is to enumerate different states of the program. The exploratory actor manager makes this its primary objective and sets out to provide a guarantee that repeated execution of a test will not explore identical orders in which messages are handled. The next subsection describes the particular mechanism used to achieve this guarantee.

3.1 Exploratory Actor Manager

The exploratory actor manager is based on the stateless search of VeriSoft [8]. This means that, except for the current state, the algorithm does not store the states of the program explicitly. Instead, states of the program are represented implicitly using index sequences of executions that lead to them. To explore a part of the state space, the test is run to completion (or until the execution of the test times out). To explore a different part of the state space, the test is restarted and a different index sequence is explored. The different states revealed by past executions are recorded and stored in a *decision tree*, with nodes implicitly representing states and edges representing index sequence elements. Initially, the algorithm holds no knowledge about the structure of the decision tree. As new index sequences are explored, the decision tree is gradually unfolded.

```
Node *dtree;
queue<Node *> seeds;
ExploratoryTest() {
  dtree = new Node();
  seeds->insert(dtree);
  while (!dtree->explored()
         && !timeout()) {
    stack<int> schedule =
      GenerateNewSchedule(seeds);
    Execution *execution =
      ExecuteTest(dtree,schedule);
    ProcessExecution(dtree,execution);
  }
}
```

Listing 1: Exploratory Testing

Unlike the deterministic actor manager and the random actor manager, the exploratory actor manager needs to store the decision tree across different executions of a test. To this end, we use an exploratory test wrapper depicted in Listing 1. The `dtree` variable stores a pointer to the node representing the initial state of the actor program. The `seeds` variable stores a collection of pointers identifying states to be explored in the future. The main loop is repeated until it is concluded that the whole state space has been explored (or until the exploration times out).

Each iteration of the loop first generates an index sequence using the `GenerateNewSchedule()` method. Notably, this method selects a seed representing a state to be explored and generates an index sequence that steers the execution towards this state. Next, an execution of the test is explored. Finally, the `ProcessExecution()` traverses the explored branch from the leaf node, which represents the final state of the test execution, to the root collecting seeds to newly discovered states and updating the exploration status of the visited states.

Note that in order for this wrapper to correctly explore all possible sequences in which a test execution can order messages, the message queue identifiers must be identical across different test executions (property of the actors library) and identical index sequences must produce identical message queues contents across different test executions (property of the actor program and its test).

```
Node *current;
stack<int> schedule;
SelectMessage() {
  int next;
  Queue<int> qids =
    GetNonEmptyQueues();
  if (!schedule.empty()) {
    next = schedule.top();
    schedule.pop();
  } else {
    next = PickNext(qids);
  }
  HandleMessageFromQueue(next);
  if (current->children().empty()) {
    Set<int>::iterator iter;
    for (iter = qids.start();
         iter != qids.end(); ++iter) {
      handle->add_child(*iter);
    }
  }
  current = current[next];
}
```

Listing 2: Message Selection

The purpose of exploring different executions of an actor program test is to systematically enumerate states of the program that a test execution can reach. Typically, an actor program test sets up the initial state of the program an uses the actors library API to both trigger message handling (e.g. handle one message, handle all messages, or continue handling messages until a condition becomes true) and to inspect the state of the program for errors. Every time a test asks for a message to be handled, the exploratory actor manager invokes the `SelectMessage()` method depicted in Listing 2. The method guarantees that the execution is steered towards the state selected by the exploratory test wrapper. Once that state has been reached, all

TEST	# EXECUTIONS	TIME
Store(1,1,1,3)	1,946 (0.00%)	12.6s (0.60%)
Store(1,1,2,3)	10,933 (0.00%)	75.2s (0.43%)
Store(1,2,2,3)	18,275 (0.00%)	133.8s (0.33%)
Store(2,2,2,3)	82,719 (0.00%)	623.2s (0.53%)
Store(2,2,3,3)	239,466 (0.00%)	1885.6s (0.64%)
Phils(2)	150 (0.00%)	0.8s (0.40%)
Phils(3)*	379,077 (0.34%)	3600.0s (0.00%)
PingPong(10)	2,048 (0.00%)	18.2s (0.40%)
PingPong(15)	65,536 (0.00%)	912.8s (0.15%)
Scheduler*	7,659 (0.77%)	3600.0s (0.00%)

Table 1: Experimental Results of Exploratory Testing. The TEST column identifies the test used in the experiment along with the parameters used for instantiating the test. The # EXECUTIONS column lists the mean and the standard deviation of the number of executions explored. The TIME column lists the mean and the standard deviation of the time spent exploring the executions.

future messages are handled in an arbitrary order, essentially unfolding a random branch of an unexplored part of the decision tree.

3.2 Evaluation

We have implemented the three actor managers as part of the Google actors library and the testing wrapper of Listing 1 as an extension to the Google Test framework [9]. Notably, the Google Test framework provides a number of macros, such as `TEST()` or `TEST_F()`, through which a software engineer can describe a unit test. When extending Google Test with support for exploratory testing, our goal was to make the transition from non-exploratory tests of actor programs to exploratory tests of actor programs as easy as possible. Consequently, we created a number of macros, such as `ETEST()` or `ETEST_F()`, that can be used as drop-in replacements for their Google Test counterparts.

The exploratory actor manager was used for testing of actor programs in a project that uses actors to build multi-threaded components of a large-scale cluster scheduling system. Table 1 shows the results of our initial experimental evaluation of the exploratory actor manager. Each experiment was repeated five times and a timeout of one hour was used to limit the execution time of each experiment. The asterisk symbol denotes experiments that failed to cover all possible branches of the decision tree. The measurements were taken using a Dell Precision T3500 workstation with 6-core Intel Xeon processor and 12 GB of memory.

The STORE test implements a multi-threaded key-value store and the parameters describe the number of front-ends, back-ends, clients, and requests used in the test. The PHILS test implements a coordination protocol similar to that of dining philosophers and the parameter determines the number of coordinating entities. The PINGPONG protocol implements a communication with acknowledgements between two actors and the parameter determines the number of messages used in the test. Lastly, the SCHEDULER test implements a scheduling prototype.

The results demonstrate that the exploratory actor manager is capable of systematic exploration. However, it hints at the limits of exploratory testing, which stem from the scarcity of time. If a single execution of an actor program test takes on the order of 10 milliseconds using the exploratory actor manager, then one should expect to explore on the order of 10^7 different executions in a day.

However, this is true only if the execution time is constant with respect to the size of the explored state space. A naive approach that stores all explored parts of the decision tree in memory does not have this property as it might run out of memory. A key insight that helps to overcome this problem is to realize that

for the sake of systematic exploration there is no need to keep fully explored subtrees of the decision tree in memory. When combined with a depth-first based enumeration of tree branches, this observation yields an exploration algorithm whose space complexity is independent of the number of explored executions, providing for a space-efficient exploration.

Further, the coverage improvement achieved by the exploratory actor manager dominates the performance overhead over the deterministic actor manager implied by reduced CPU cache hit rate and execution post-processing. However, it is not as clear whether the same holds true for a comparison between the exploratory actor manager and the random actor manager. The random actor manager is as fast as the deterministic actor manager (or even faster if messages are not serialized) and, unlike the deterministic actor manager, is expected to explore different executions.

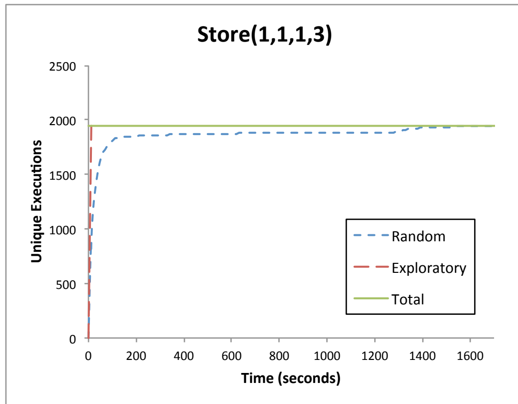
To draw a comparison between the exploratory actor manager and the random actor manager, we designed an experiment which measures the number of unique executions explored by each of the managers over time. The results of the experiment are depicted in Figure 1. The dotted lines show the rate at which both the exploratory actor manager and the random actor manager explore new executions. For the tests the exploratory actor manager was able to fully explore in under one hour, the total number of executions is shown in the graphs as a solid horizontal line. A timeout of one hour was used to limit the duration of each experiment.

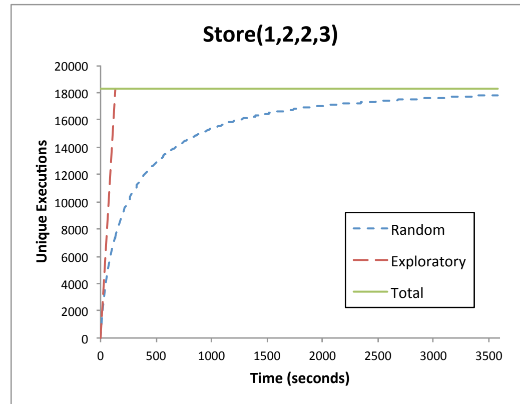
The graphs suggest that the rate at which the random actor manager explores new executions is sublinear, while the rate at which the exploratory actor manager explores new executions is linear. A coverage gap opens up between the two managers over time and the random actor manager often needs substantially more time to achieve full coverage. Out of the measurements carried out, the random actor manager covered all executions only for STORE(1,1,1,3) in 1,931 seconds, for PINGPONG(10) in 125.2 seconds, and for PHILS(2) (not depicted) in 13.76 seconds. In comparison, the exploratory actor manager needed 12.52 seconds, 17.77 seconds, and 0.73 seconds respectively to cover all executions. We conjecture that the more uneven the probability with which the random actor manager samples the execution, the bigger the coverage gap between the exploratory actor manager and the random actor manager.

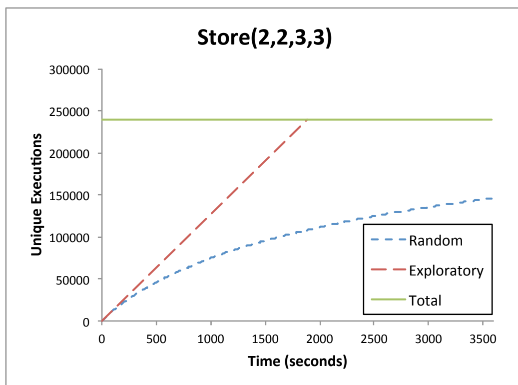
In essence, the random actor manager is trying to solve the coupon collector problem. If the random actor manager is sampling executions uniformly at random, the expected value of the number of trials needed to encounter each execution at least once is known to be $O(n \log n)$, where n is the total number of executions. Further, it can be shown that such a sampling strategy is optimal. However, in practice it is difficult to implement the optimal sampling strategy and, similarly to our approach, it is common to resort to a coarse approximation of the optimal strategy. The downside of sampling executions with a non-uniform probability is that the expected value of the number of trials needed to encounter each execution is potentially unbounded.

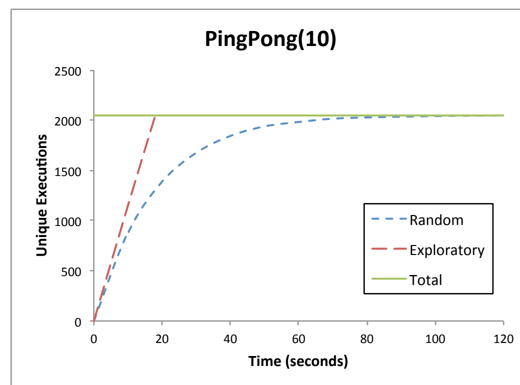
Based on our experimentation we conclude that the exploratory actor manager outperforms a random actor manager when it comes to fully covering a state space. However, given a time budget in which only a small fraction of a state space can be explored (c.f. Figure 1), a random actor is almost as effective as an exploratory actor manager.

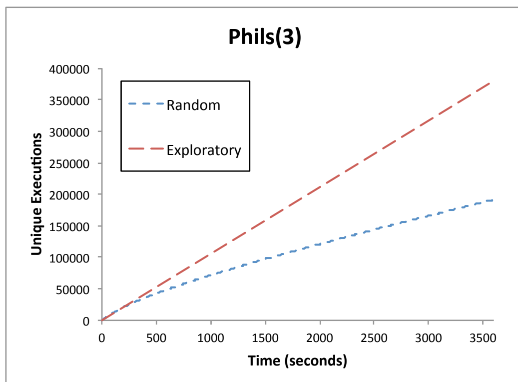
Unfortunately, the conclusion drawn from the above experiments is hardly satisfying for at least two reasons. Firstly, one does not know how big a state space is until it is actually fully explored. Secondly, due to combinatorial explosion, even for small tests, the size of the state space might be too big to be within the time limits discussed earlier. We address the first objection in the following section, presenting a novel method for estimation of the total number of executions. The second objection is addressed in the two subsequent sections, evaluating a state space reduction method and a concurrent state space exploration method.











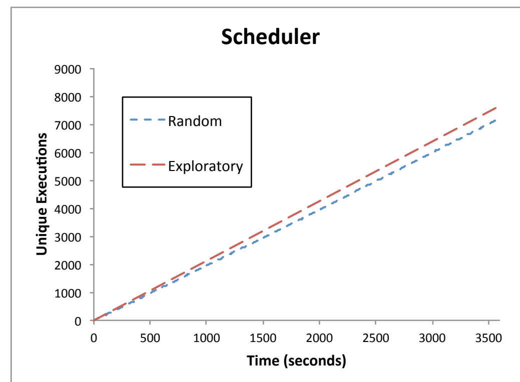


Figure 1: Exploratory Actor Manager vs. Random Actor Manager

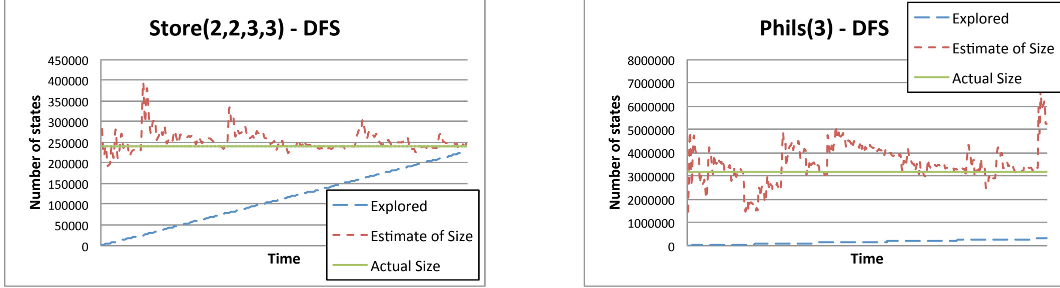


Figure 2: State Space Estimation with DFS

4 State Space Estimation

In this section we present a novel algorithm for online estimation of the total number of branches of a partially explored decision tree.

The algorithm is based on two ideas. First, the algorithm computes a size estimate for each subtree of the decision tree. Trading space for time, this idea allows the algorithm to use a newly explored execution to recompute the size estimates in time that is linear with respect to the length of the explored branch. Second, the algorithm assumes that the decision tree has balanced subtrees, an assumption used for estimating the size of an unexplored subtree based on the sizes of its explored siblings.

When a state of a newly explored execution is processed in the `ProcessExecution()` method of the wrapper from Listing 1, the function additionally computes the average estimated size of all of the children subtrees visited by at least one execution. Assuming the states of an execution are processed in a bottom-up manner, these estimates are readily available; when the size of a tree rooted at some state is estimated, all visited children subtrees already had their size estimated. The average estimated size of the visited children is then multiplied by the number of children (both visited and unvisited) and the result is the new estimate for the size of the tree rooted at the state being currently processed.

We implemented this algorithm and used to it periodically estimate the size of the state space to be explored by the exploratory actor manager. The results of our experiments are depicted in Figure 2. The dotted lines show the number of explored executions and the value of the state space size estimate. The total number of executions is shown in the graphs as a solid horizontal line. A time out of one hour was used to limit the duration of each experiment.

On one hand, when a decision tree is perfectly balanced, which is true for example for the PING-PONG(15) test (not depicted), the estimate is accurate and stable. One the other hand, in the STORE(2,2,3,3) and PHILS(3) tests, there are points in time when the estimate is inaccurate by a factor of up to 2. This inaccuracy stems from the fact that the assumption of the algorithm that the decision tree has balanced subtrees is not always met. Since our main application of the size estimate is to approximate the progress of an exploratory test, the above inaccuracy is acceptable. In particular, it allows us to determine whether an exploratory test is expected to finish in hours, days, or years.

Further, the depth-first search nature of the decision tree exploration causes the estimation to be based largely on the information computed by the last execution, which leads to the instability of the estimate. To validate the connection between the depth-first search nature of the exploration and the instability of the estimate, we replaced the exploration algorithm with an algorithm that picks the next branch to be explored uniformly at random out of the candidate branches. Note that such an exploration algorithm in general no longer runs with space complexity independent of the number of explored executions. The results of the experiments are depicted in Figure 3 and validate our hypothesis that the previous estimate instability was an artifact of the depth-first search based exploration algorithm. Further, the random exploration algorithm

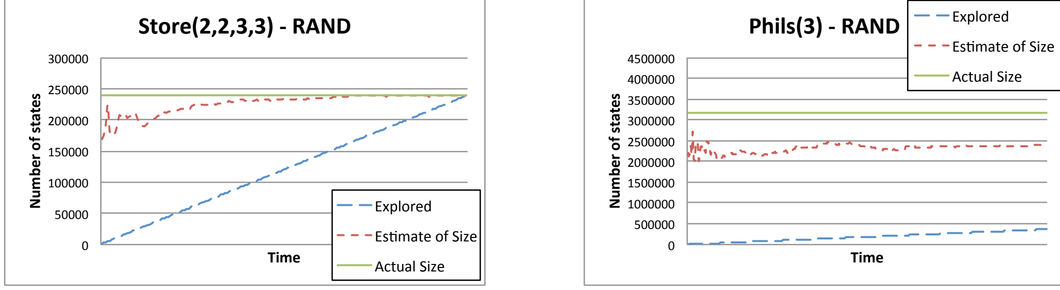


Figure 3: State Space Estimation with RAND

achieves higher accuracy (less than 50% error in all of our experiments). However, the increased accuracy and stability of the estimate comes at a price of space complexity that makes the this algorithm practical only for state spaces that can fit in the main memory.

Finally, the SCHEDULER experiment (not depicted) is by far the most complex of our tests. The estimation for this test suggests that there is on the order of 10^{17} possible executions. Under the assumption that a single execution takes 0.5 seconds, enumerating all executions of such a state space would require 10^9 years. The next two sections evaluate two orthogonal approaches – state space reduction and concurrent state space exploration – that address this problem.

5 State Space Reduction

In this section we first lay out a formal foundation for reasoning about independence between messages and equivalence between executions. Then, we recast the dynamic partial reduction (DPOR) technique [6] in the context of actor programs. Lastly, after describing the DPOR algorithm for actor programs, we evaluate its performance and interaction with the state space estimation algorithm.

5.1 Formal Definitions

First, we define notation that allows us to refer to the sequences of messages handled in an execution.

Definition 5.1. Let $\alpha = (s_0, \ldots, s_k)$ be an execution and $I(\alpha) = (i_0, \ldots, i_{k-1})$ its index sequence. We define the *message sequence* $M(\alpha) = (m_0, \ldots, m_{k-1}) \in Message^k$ to be a sequence of messages such that $m_j = head(queue(actor(s_j, i_j)))$ for $j \in \{0, \ldots, k-1\}$. Further, we use M_α to denote the set $\{m_0, \ldots, m_{k-1}\}$ of messages.

Remark 5.2. If α is an execution, then $M(\alpha)$ is unique.

Next, we define a happens-before relation [12] over messages of an execution. The happens-before relation is induced by processing order, specifically the delivery of a message happens before any messages sent as the result of processing it.

Definition 5.3. Let $\alpha = (s_0, \ldots, s_k)$ be an execution, $I(\alpha) = (i_0, \ldots, i_{k-1})$ its index sequence, and $M(\alpha) = (m_0, \ldots, m_{k-1})$ its message sequence. We define a binary *happens-before* relation $\leadsto_\alpha \subseteq M_\alpha \times M_\alpha$ as the smallest relation that satifies the following conditions:

- for all $j, l \in \{0, \ldots, k-1\}$:

  if $j < l$ and $i_j = i_l$, then $m_j \leadsto_\alpha m_l$

- for all $j \in \{0, \dots, k-1\}$:
  if $\Delta_{i_j}(m_j, local(actor(s_j, i_j))) = ((q'_1, \dots, q'_n), l')$, then $m_j \rightsquigarrow_\alpha m$ for every message m contained in any of the queues $q'_1, \dots, q'_n$

- if $m \rightsquigarrow_\alpha m'$ and $m' \rightsquigarrow_\alpha m''$, then $m \rightsquigarrow_\alpha m''$

Lastly, we define an independence relation over messages of an execution. The motivation behind the definition of independence is to identify pairs of messages with commutative effects. Note that our definition is also an independence relation in the sense of [7].

Definition 5.4. Let $\alpha = (s_0, \dots, s_k)$ be an execution, $I(\alpha) = (i_0, \dots, i_{k-1})$ its index sequence, and $M(\alpha) = (m_0, \dots, m_{k-1})$ its message sequence. We define a binary *independence* relation $I_\alpha \subseteq M_\alpha \times M_\alpha$ as a relation that satisfies the following condition:

- for all $j, l \in \{0, \dots, k-1\}$:
  if $\Delta_{i_j}(m_j, local(actor(s_j, i_j))) = ((q'_1, \dots, q'_n), l')$ and $\Delta_{i_l}(m_l, local(actor(s_l, i_l))) = ((q''_1, \dots, q''_n), l'')$ and for all $i \in \{1, \dots, n\}$ either q'_i or q''_i is an empty sequence, then $I_\alpha(m_{i_j}, m_{i_l})$

The above definition is not a maximal relation that meets the restrictions imposed on an independence relation in [7]. In general, one aims to identify as many pairs of independent messages with as little overhead. The more independent messages there are, the better the realized reduction is. We crafted the above relation so that it can be easily computed at runtime, avoiding the need for static analysis and program annotations. A potential direction for future work is to refine the above relation and, at the same time, to provide a runtime for computing the refined relation.

5.2 Algorithm

Our DPOR algorithm for actors has two parts. First, the actor manager is extended with a component that computes the happens-before relation and the independence relation between messages. Second, the `ProcessExecution()` method of the wrapper from Listing 1 is modified. In particular, it uses the happens-before relation and the independence relation to identify which partial branches exposed by the currently processed execution need to be explored in future as they might correspond to executions that are not equivalent to the currently processed one. In this context, equivalent executions are executions that differ only in the order of independent messages. In other words, DPOR infers that two executions are equivalent by exploring only one of them and extrapolating the other one using the happens-before relation and the independence relation.

During the bottom-up processing of states of an execution, each state is checked for *evil ancestors*, defined as follows. Given an execution α, an evil ancestor of a state s_i is a state s_j such that the following holds:

- s_j precedes the state s_i in the execution: $i > j$

- the message processed during the transition to s_j did not happen before the message processed during the transition to s_i: $m_j \not\rightsquigarrow_\alpha m_i$

- the message processed during the transition to s_j is not independent of the message processed during the transition to s_i: $(m_j, m_i) \notin I_\alpha$

Intuitively, an evil ancestor captures the fact that two messages are concurrent and handling them in a different order might produce different states.

If a state has an evil ancestor, a *good sibling* of the nearest evil ancestor needs to be identified. A good sibling g of an evil ancestor e of a state s is a sibling of e such that the message processed during the transition to g either is the message processed during the transition to s or happened before the message processed during the transition to s. Intuitively, a good sibling identifies an execution prefix that allows the two concurrent messages to happen in the opposite order. An evil ancestor can have multiple good siblings and it suffices to only identify one of them. As an optimization, our implementation favors siblings that have been visited or selected as a good sibling before.

Finally, the results from [6] imply that exploring only the branches corresponding to good siblings is a sufficient condition for visiting all equivalence classes of the above mentioned equivalence relation. Further, the reduced state space preserves validity of any state predicates that do not relate private states of different actors. For predicates that relate private states of different actors, DPOR might produce false negatives, but can still be useful as a heuristic for sampling executions from a very large state space.

5.3 Evaluation

We have implemented the algorithm described in the previous subsection as part of Google actors library and used it for exploratory testing of the programs presented in the previous section. The Table 2 shows the results of our experimental evaluation of DPOR for actors. The experimental setup was identical to the one used for the experiments presented in Table 1.

TEST	# EXECUTIONS	TIME
Store(1,1,1,3)	289 (0.00%)	2.28s (0.01%)
Store(1,1,2,3)	765 (0.00%)	6.70s (0.89%)
Store(1,2,2,3)	1,235 (0.00%)	10.50s (0.19%)
Store(2,2,2,3)	12,216 (0.00%)	97.46s (0.62%)
Store(2,2,3,3)	25,887 (0.00%)	216.85s (1.00%)
Phils(2)	44 (0.00%)	0.22s (0.00%)
Phils(3)	67,192 (0.00%)	814.34s (8.59%)
PingPong(10)	1,024 (0.00%)	11.68s (0.05%)
PingPong(15)	32,768 (0.00%)	622.86s (4.31%)
Scheduler*	5,227 (0.32%)	3,600.00s (0.00%)

Table 2: Experimental Results for DPOR

The results demonstrate that for the tests that can be fully explored by the *exhaustive* approach, which uses no reduction, the *selective* approach based on DPOR speeds up the exploration by a factor between 1.4× and 12.5× and 6.3× on average. In comparison, the state space reduction realized for these tests is between 2.0× and 14.8× and 7.4× on average. The difference between the realized speedup and reduction is caused by the overhead of computing the happens-before and independence relations, evil ancestors and good siblings.

Unlike the exhaustive approach, the selective approach managed to fully cover the state space of PHILS(3) in under one hour (c.f. Table 2). For the sake of comparison, the exhaustive approach was also used to fully exhaust the state space of the PHILS(3) test, resulting in $3.2M$ states being explored in just under 9 hours. In other words, the selective approach achieved a speedup of almost 40× and a reduction of over 47×.

Next, given that the selective approach explores only a subset of all possible executions, it is of interest to investigate its impact on the accuracy of the state space size estimation mechanism. To evaluate the interaction between DPOR and the state space size estimation we carried out the following experiments.

TEST	EXHAUSTIVE	SELECTIVE	ACC
Store(1,1,1,3)	1,946	1,090	1.79
Store(1,1,2,3)	10,933	3,283	3.33
Store(1,2,2,3)	18,275	5,887	3.10
Store(2,2,2,3)	82,719	39,666	2.09
Store(2,2,3,3)	239,466	85,814	2.79
Phils(2)	150	118	1.27
Phils(3)	3,159,870	1,104,386	2.86
PingPong(10)	2,048	2,048	1.00
PingPong(15)	65,536	65,536	1.00

Table 3: Accuracy of Size Estimation with DPOR

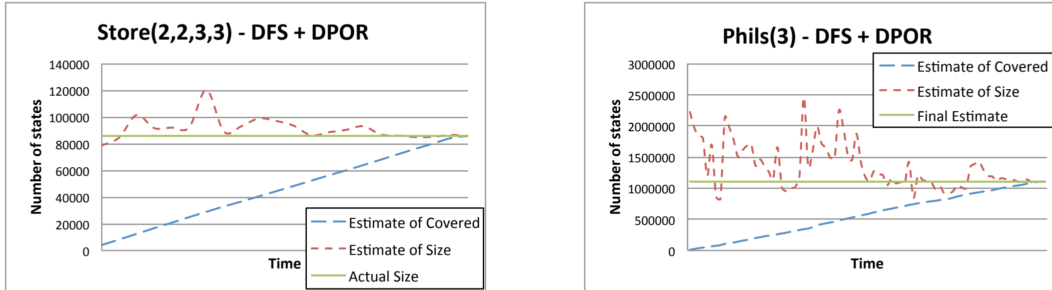


Figure 4: State Space Estimation with DFS and DPOR

Table 3 compares the actual size of the state space computed by the exhaustive approach (column EXHAUSTIVE) to the final estimate of the state space size computed by the selective approach (column SELECTIVE). The ACC column lists the accuracy of this estimate computed as $2^{|\log(y/x)|}$, where x is the actual size and y is the estimated size. Interestingly, the selective approach underestimates the total size of the space for all the tests and we do not have a good explanation for this phenomenon. The result of this experiment shows that the accuracy of the state space size estimate of the selective approach ranges between $1.00\times$ and $3.33\times$ and is $2.14\times$ on average, which is worse than for the exhaustive approach.

Fortunately, the decreased accuracy of the final state space size estimate computed by the selective approach does not impact the accuracy with which one can approximate the time to completion of the selective approach. For an accurate estimate of the time to completion, computed as the estimated number of covered executions over the estimated number of all executions, the estimated number of all executions should converge to the final estimate quickly and the estimated number of covered executions should increase linearly over time.

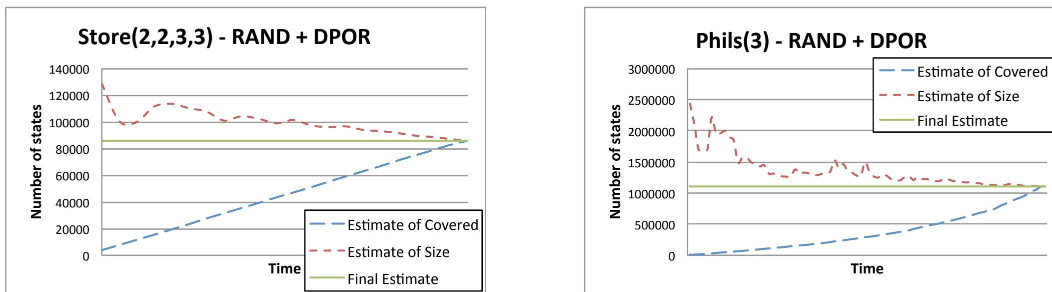


Figure 5: State Space Estimation with RAND and DPOR

To examine the reality, the experiments presented in Figures 2 and 3 were repeated using the selective approach. The results of these experiments are presented in Figure 4 and 5. Instead of plotting the rate at which executions are explored, the graphs plot the rate at which executions are estimated to be covered. Further, the solid horizontal line identifies the final value of the size estimate (as opposed to the actual size of the state space). The result of these experiments look similar to the ones presented in Figures 2 and 3 and indicate that DPOR and our estimation technique are compatible.

Overall, DPOR is a practical technique and in the experiments we carried out it helped us to push the limits of exploratory testing by an order of magnitude. Our experiments with very large state spaces suggest that one should think of DPOR as a search heuristic rather than a method that can cover all states of an arbitrarily large state space.

6 Concurrent State Space Exploration

Unlike the previous section, which tries to mitigate the combinatorial explosion with the help of formal methods, this section attacks the same problem with the help of parallel processing. In particular, this section presents the design, implementation, and evaluation of an algorithm that explores the branches of a decision tree using an army of workers.

6.1 Design

The challenge in concurrent exploration of different executions of a concurrent system is how to achieve efficiency. In particular, a concurrent exploration should guarantee that different explorations indeed explore different executions and at the same time the per execution overhead of concurrent exploration is as little as possible.

Our algorithm for concurrent state space exploration uses a single *master* process and multiple *worker* processes. The master process maintains the state of the exploration represented as a decision tree. However, instead of exploring yet unexplored branches of the decision tree, the master delegates this work to workers. When finished, the workers report the results of their exploration back to the master and the master then maps the results into the decision tree.

To bootstrap the exploration, the master first asks a single worker to explore some fixed number k_1 of branches, using a breadth-first policy to determine what branches to explore next. These branches are used to identify disjoint subtrees. When a worker becomes idle, the master selects a subtree that has not been fully explored and asks the worker to explore some fixed number k_2 of branches of that subtree, using a depth-first policy to determine what branches to explore next.

The number k_1 determines how many independent chunks of work will be created. Given that different chunks of work will generally represent subtrees of different size, to achieve load-balancing it is advisable to pick k_1 that is an order of magnitude larger then the number of workers. Further, the depth-first based nature of the subsequent exploration implies that the master can represent the decision tree using k_1 branches. Thus, similarly to the sequential exploration algorithm, the space complexity of the concurrent exploration is independent of the number of explored executions.

The number k_2 determines how many branches should a worker explore before reporting back to the master. On one hand, by selecting a number that is too low, one risks that the overhead of 1) sending a request from the master to a worker, 2) sending a response from a worker to the master, and 3) processing the response at the master imposes a prohibitive per execution overhead. On the other hand, by selecting a number that is too high, one risks that a large amount of work might be lost when a worker fails and exacerbates the stragler problem.

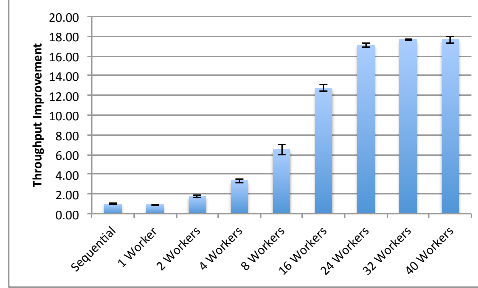


Figure 6: Scalability of Concurrent State Space Exploration

6.2 Implementation

We implemented a prototype of the above design using a Google RPC library and the Protocol Buffers [18] serialization format. The master is multi-threaded and asynchronous. The main thread keeps a queue of yet to be explored subtrees (generated by the initial breadth-first based exploration), repeatedly checks if there still is work to be done, and sends RPC work requests to idle workers. The responses are handled asynchronously using callback threads, which map the results of the exploration into the decision tree. To achieve mutual exclusion the main thread and the callback threads serialize their access to the decision tree.

Further, the implementation tolerates worker failures by using a timeout. If a worker fails to send a response within a fixed time window, the main thread re-assigns the subtree the failed worker operated on to someone else, or marks it as unassigned in case there are no idle workers.

6.3 Evaluation

To evaluate the performance of our concurrent state space exploration algorithm, we compared the number of executions explored in an hour by the sequential state space exploration to the number of executions explored in an hour by a concurrent state space exploration with 1, 2, 4, 8, 16, 24, 32, and 40 workers. Each measurement was repeated three times and the Figure 6 depicts the mean and the standard deviation of these measurements. The horizontal axis identifies the configuration and the vertical axis identifies the throughput improvement relative to the number of executions explored in an hour by the sequential algorithm.

The results demonstrate that our implementation of the concurrent exploration achieves up to $17.5\times$ throughput improvement over the sequential algorithm. At 24 workers, the serialization of the decision tree access becomes a bottleneck and adding extra workers has little additional benefit. This bottleneck is an aspect of our implementation and is not inherent to concurrent state space exploration. We believe that another order of throughput improvement can be achieved by optimization of the critical sections in the master and we are currently working towards this goal. Abstractly, a centralized master will always be a bottleneck and one can be remove it by adopting a de-centralized solution.

The concurrent state space exploration algorithm is compatible with both DPOR and the state space size estimation techniques described earlier. In case of DPOR, one could implement the execution processing logic either in the worker or in the master. Implementing the reduction in the worker, reduces both the amount of information a response to the master needs to contain and the amount of time the master needs to spend in its critical section. However, this might result in a suboptimal reduction because the worker does not always have up-to-date information about what states have been explored and might fail to choose good siblings optimally. We experimented with both approaches and concluded that off-loading the computation of reduction to the workers achieves similar reduction and scales better.

7 Related Work

Exploratory testing of concurrent programs have been pioneered by VeriSoft [8], which is a tool for systematic testing of multi-threaded C and C++ programs. Recently, the work on VeriSoft has been followed by a number of papers that extend the scope of exploratory testing to new domains. MaceMC [11] is an exploratory testing checker for distributed programs written in the Mace language [10] capable of exploring different orders in which concurrent Mace transactions execute. ISP [21] is an exploratory testing tool with support for DPOR and capable of exploring different orders of concurrent MPI function invocations. CHESS [17, 16] is an exploratory testing tool for multi-threaded Windows programs capable of exploring different orders in which threads are pre-empted. Modist [23] is an exploratory testing tool for multi-threaded and distributed Windows programs with support for DPOR and capable of exploring different orders of concurrent Windows API function invocations. Basset [13, 14] is an exploratory testing tool based on Java Pathfinder [22] for actor programs written in Java with support for DPOR and capable of exploring different orders of concurrent Actor-Foundry API function invocations. Finally, dBug [20] is an exploratory testing tool for multi-threaded and distributed Linux programs capable of exploring different orders of POSIX API invocations.

Similarly to some of the above tools, Eta also utilizes DPOR, but in a different setting, using a Google actors library and C++ codebases. Unlike previous work, we have developed a mechanism for estimating the total number of executions, which allows us to quantify the complexity of a test. Further, to help mitigate the effect of scale, we developed concurrent exploration and avoid keeping the entire explored state in memory.

An alternative approach to handling test non-determinism due to concurrency is to eliminate it through the use of a deterministic multi-threading (DMT) system. In general, DMT systems aim to provide the guarantee that given the same inputs, a multi-threaded program will execute concurrent events in the same order. The challenge addressed by a number recent papers has been to build an efficient and robust DMT system. CoreDet [2] is a compiler-based DMT system, that introduces non-negligible runtime overhead. Peregrine [5] is a symbolic-execution based DMT system that partition the input space into equivalence classes and use the same schedule for equivalent inputs. Finally, `dthreads` [3] is a DMT library that can be used as a drop-in replacement for the `pthreads` library, achieving performance comparable to `pthreads`.

8 Conclusion

This paper set out to answer a number of questions related to exploratory testing. To conclude this paper, we use the results presented in this paper to answer these questions.

Is exploratory testing more efficient than stress testing? The evaluation in Section 3 indicates that the rate at which random testing encounters new behaviors is sub-linear, while the rate at which exploratory testing encounters new behaviors is linear. One might argue that, not needing to serialize concurrent events, random testing can compensate for the aforementioned inefficiency by exploring executions faster than exploratory testing. However, we believe that this is an artifact of existing implementations and instead of serializing all concurrent events, exploratory testing could, similarly to [5], serialize only a subset of concurrent events.

What else besides finding bugs can exploratory testing be used for? Section 3 demonstrated that exploratory testing provides both a convenient mechanism for measuring the coverage of the space of possible executions and a platform for implementing state space exploration heuristics.

Can the progress of an exploratory test be approximated? Section 4 demonstrated that exploratory testing can be extended with a technique for estimation of the state space size, which can be used to approximate the time to completion of an exploratory test. Although the accuracy achieved by this technique is not perfect, it allow us to estimate the time to completion with a reasonable precision.

What techniques can help to scale exploratory testing? Section 5 adapted an existing state space reduction technique [6] for actor programs and demonstrated that it can avoid the need to explore over 85% of executions. Section 6 developed a new concurrent state space exploration algorithm and demonstrated that it can improve the throughput of state space exploration by a factor of over 17×.

Our experience suggests that unlocking the potential of both formal methods and parallel processing is the key to pushing the limits of exploratory testing.

References

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

[2] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution. In James C. Hoe and Vikram S. Adve, editors, *ASPLOS*, pages 53–64, 2010.

[3] Emery Berger, Tongping Liu, and Charlie Curtsinger. dthreads: Efficient and Deterministic Multithreading. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, 2011.

[4] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI '08: Proceedings of the 8th Conference on USENIX Symposium on Operating Systems Design and Implementation*, pages 209–224, 2008.

[5] Heming Cui, Jingyue Wu, John Gallagher, Huayang Guo, and Junfeng Yang. Efficient Deterministic Multithreading through Schedule Relaxation. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, 2011.

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

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

[8] 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.

[9] Google Test. http://code.google.com/p/googletest/, 2011.

[10] Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin M. Vahdat. Mace: Language Support for Building Distributed Systems. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 179–188, 2007.

[11] 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.

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

[13] Steven Lauterburg, Mirco Dotta, Darko Marinov, and Gul Agha. A Framework for State-Space Exploration of Java-Based Actor Programs. In *IEEE/ACM International Conference on Automated Software Engineering*, pages 468–479, 2009.

[14] Steven Lauterburg, Rajesh Karmani, Darko Marinov, and Gul Agha. Evaluating Ordering Heuristics for Dynamic Partial-Order Reduction Techniques. In *Fundamental Approaches to Software Engineering*, pages 308–322. 2010.

[15] Hans Meuer, Erich Strohmaier, Jack Dongarra, and Horst Simon. Top 500 Supercomputing Sites. http://www.top500.org, 2011.

[16] Madanlal Musuvathi and Shaz Qadeer. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 446–455, 2007.

[17] 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.

[18] Protocol Buffers. http://code.google.com/p/protobuf/, 2011.

[19] Bianca Schroeder and Garth Gibson. Understanding Failures in Petascale Computers. *Journal of Physics: Conference Series*, 78(1):12–22, 2007.

[20] Jiri Simsa, Garth Gibson, and Randy Bryant. dBug: Systematic Testing of Unmodified Distributed and Multi-threaded Systems. In *SPIN*, pages 188–193, 2011.

[21] 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.

[22] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model Checking Programs. *Automated Software Engineering*, pages 203–232, 2003.

[23] 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.

[24] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using Model Checking to Find Serious File System Errors. *ACM Trans. Comput. Syst.*, 24(4):393–423, 2006.