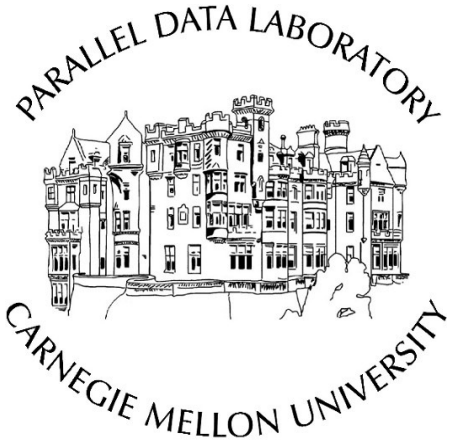

Runtime Estimation and Resource Allocation for Concurrency Testing

Jiri Simsa, Randy Bryant, Garth Gibson

CMU-PDL-12-113
December 2012

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

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 Alexey Tumanov, Samantha Gottlieb, and the members of Google's cluster management team for their technical feedback. The authors are also thankful to Google for providing the traces used for the experimental evaluation presented in this paper. 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: Concurrency, Stateless Exploration, State Space Reduction, Runtime Estimation, Resource Allocation

Abstract

In the past 15 years, stateless exploration, a collection of techniques for automated and systematic testing of concurrent programs, has experienced wide-spread adoption. As stateless exploration moves into practice, becoming part of testing infrastructures of large-scale system developers, new practical challenges are being identified.

In this paper we address the problem of efficient allocation of resources to stateless exploration runs. To this end, this paper presents techniques for estimating the total runtime of stateless exploration runs and policies for allocating resources among tests based on these runtime estimates.

Evaluating our techniques on a collection of traces from a real-world deployment at Google, we demonstrate the techniques' success at providing accurate runtime estimations, achieving estimation accuracy above 60% *after as little as* 1% *of the state space has been explored. We further show that these estimates can be used to implement intelligent resource allocation policies that meet testing objectives more than twice as efficiently as the round-robin policy.*

1 Introduction

In the past 15 years, stateless exploration [5], a collection of techniques for automated and systematic testing of concurrent programs, has witnessed wide-spread adoption [9, 20, 25] due to its ability to discover rare concurrency errors better than stress testing [14].

As stateless exploration moves into wider practice, becoming part of testing infrastructures of large-scale system developers [14, 15], new practical challenges are emerging. An example of such a practical challenge is the problem of efficient allocation of resources to a collection of tests.

In our experience [15], a test suite typically consists of tens to hundreds of tests of varied, and initially unknown, length. In the context of stateless exploration it is reasonable to assume that the resources available for running these tests are not always sufficient to complete all tests by a deadline. In such cases, high-level testing objectives, such as "maximize the number of completed tests" or "achieve even state space coverage among tests" are used to drive the allocation of testing resources. Mapping these high-level testing objectives into working allocation mechanisms is an important, practical, and yet unresolved problem.

In this paper, we propose a solution to this problem based on runtime estimation. In particular, we design and evaluate techniques that estimate the time needed to complete an ongoing stateless exploration run. Besides offering a measure of test complexity, our evaluation demonstrates that runtime estimation enables efficient allocation of resources among a collection of tests.

In most implementations [5, 9, 17, 20, 25], the state space explored by a stateless exploration run is represented as a tree that records the different executions of a test explored so far. Under this abstraction, the problem of estimating the runtime of a stateless exploration run can be framed as the problem of estimating the time needed to explore a tree.

The estimation techniques presented in this paper can be characterized as *online* – updating the estimate as stateless exploration progresses – and *passive* – not mandating a particular order in which the exploration proceeds.

The benefit of online estimation is that the estimate can be refined as new information about the state space is gathered. The benefit of passive estimation is that it can be combined with any state state exploration strategy. In addition, passive estimation techniques can be evaluated using traces of stateless exploration runs. We took advantage of this fact in our evaluation, using a collection of stateless exploration traces from a real-world deployment at Google [18].

The contributions of this paper are as follows. First, building on research on search tree size estimation [8], this paper presents techniques for runtime estimation of stateless exploration. Second, this paper evaluates the accuracy of the presented estimation techniques using a collection of stateless exploration traces from a real-world deployment. Third, this paper demonstrates the practicality of runtime estimation by using it to implement efficient resource allocation policies and evaluating the efficiency of these policies.

The rest of the paper is organized as follows. First, Section 2 provides an overview of stateless exploration and describes the syntax and semantics of exploration traces. Section 3 presents techniques for runtime estimation of stateless exploration and resource allocation policies based on runtime estimation. Section 4 describes a collection of exploration traces and uses the traces to evaluate the accuracy of the runtime estimation techniques and the efficiency of resource allocation policies based on these runtime estimates in meeting testing objectives. Finally, we discuss related work in Section 5 work and conclude in Section 6.

2 Background

In this section we first provide a brief overview of the state of the art stateless exploration and then define the syntax and semantics of traces of stateless exploration.

2.1 Stateless Exploration

Stateless exploration is a technique for 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 orders in which concurrent events of the program can be exercised by a program test.

To keep track of the exploration progress, stateless exploration abstractly represents the state space of the program test using an *execution tree*. Nodes of the execution tree represent non-deterministic choice points and edges record non-deterministic choices representing program state transitions. A branch leading from the root of the tree to a leaf thus encodes a unique test execution as a sequence of non-deterministic choices.

In this model, a branch of the execution tree corresponds to a particular sequence of program state transitions. Notably, the set of explored branches of a partially explored execution tree identifies the test executions that 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 depth-first search results in space-efficient exploration. To mitigate the state space explosion problem, modern tools for systematic testing of concurrent software [2, 17, 20, 25] combine stateless exploration with dynamic partial order reduction [3] that avoids exploration of redundant parts of the execution tree. In particular, when stateless exploration executes a program state transition, dynamic partial order reduction computes the happens-before [11] and the independence [4] relations over the set of program state transitions and uses this information to decide to also explore the program state transitions that could have been explored instead.

2.2 Exploration Traces

Given the passive nature of estimation techniques we study in this paper, we can describe the problem of online runtime estimation of stateless exploration using the abstraction of an *exploration trace*, which identifies events pertinent to runtime estimation.

An exploration trace is a sequence of exploration events, where an event is one of the following:

1. `AddNode x y` – A node `x` with parent `y` has been added to the execution tree (the root node is `0` and the parent of the root node is `-1`).
2. `Explore x` – The node `x` has been marked for exploration (note that due to the nature of dynamic partial order reduction, `x` does not need to be a child of the current node).
3. `Transition x` – The exploration transitioned from the current node to node `x`.
4. `Start` – New test execution, setting node `0` as the current node, has started.
5. `End t` – Current test execution finished after `t` time units.

Figures 1 and 2 give an example of a simple execution tree and an exploration trace describing its exploration. Initially, the root node 0 is added and marked for exploration. Next, an execution is started from the root node. The children 1, 2, and 3 of the root node are added and the child 1 is marked for exploration. The execution then transitions to node 1. The children 4 and 5 of node 1 are added and the child 4 is marked for exploration. The execution transitions to node 4. The child 6 of node 4 is added and marked for exploration. Finally, the execution transitions to node 6 and for the sake of this example we assume that the use of dynamic partial order reduction results in the node 3 being marked for exploration. Since node 6 has no children, the execution ends, requiring a total of 0.42 time units. Next, a new execution

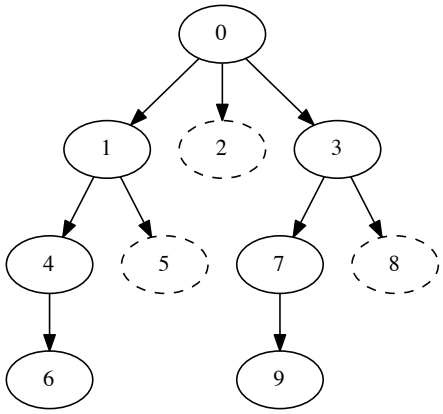

Figure 1: Execution Tree Example

```
AddNode 0 -1
Explore 0
Start
AddNode 1 0
AddNode 2 0
AddNode 3 0
Explore 1
Transition 1
AddNode 4 1
AddNode 5 1
Explore 4
Transition 4
AddNode 6 4
Explore 6
Transition 6
Explore 3
End 0.42
Start
Transition 3
AddNode 7 3
AddNode 8 3
Explore 7
Transition 7
AddNode 9 7
Explore 9
Transition 9
End 0.29
```

Figure 2: Exploration Trace Example

is started from the root node. The second execution proceeds in a similar fashion, visiting node 3, exploring a branch in its subtree, and requiring a total of 0.29 time units. Note that nodes 2, 5, and 8 are never marked for exploration and consequently never explored.

3 Methods

In this section we present a number of techniques for estimating the runtime of stateless exploration. During stateless exploration each execution starts from the root node of the execution tree (the initial state of the program) and ends in a leaf node (a terminal state of the program). This allows us to describe an estimation technique as an algorithm that operates over the exploration trace model. In particular, a sequential scan of the exploration trace can be used to build up the execution tree, recording which nodes have been marked for exploration, visited, and explored.

The estimation techniques proposed by this paper consist of several components. First, the *strategy* component is used to determine how to treat nodes that have not been marked for exploration yet. Second, the *estimator* component is used to determine how to combine a strategy and the exploration information gathered so far to compute an intermediate runtime estimate. Third, the *fit* component is used to aggregate the sequence of intermediate runtime estimates computed so far to produce the actual runtime estimate.

3.1 Strategies

In general, stateless exploration does not explore all subtrees of an execution tree because dynamic partial order reduction identifies the exploration of some parts of the tree as redundant. Further, the information about which subtrees need to be explored is revealed only as stateless exploration progresses through the state space. Consequently, to estimate the runtime of stateless exploration, one needs a *strategy* that determines how to treat nodes that have not been marked for exploration.

In this paper we consider three strategies for computing the function $F : V \to \mathbb{N}$, which for a node v of a partially explored execution tree, estimates the number of unmarked (and thus unexplored) children of node v that the strategy expects to be marked (and thus explored) in the future.

- *Hindsight* – Assumes perfect knowledge about which nodes will be explored, computing the function F by pre-processing the exploration trace. This strategy is infeasible in practice but we consider it as a best case scenario for comparison.
- *Lazy* – Assumes that a node will not be explored unless it has been marked for exploration. In other words, $F(v) = 0$ for all nodes v irrespective of the exploration status of the execution tree.
- *Eager* – Assumes that a node will be explored unless the exploration has already backtracked from the parent without exploring the node. In other words, $F(v)$ is equal to the number of unmarked children if the exploration has not backtracked from v and 0 otherwise.

Space and Time Complexity

The hindsight strategy, which we use for benchmarking purposes only, requires linear pre-processing time and has $O(n)$ space overhead, where n is the size of the execution tree. Neither the lazy nor the eager strategies has any overhead.

3.2 Estimators

Having estimated which parts of a partially explored execution tree remain to be explored, we pass this information onto an *estimator*. The estimator is responsible for producing an intermediate runtime estimate. In this paper we consider two different estimators based on previous work by Kilby et al. [8]: the weighted backtrack estimator and the recursive estimator.

3.2.1 Weighted Backtrack Estimator (WBE)

The *weighted backtrack estimator* WBE) is an online variant of Knuth's offline technique [10] for tree size estimation. WBE uses the length of each explored branch weighted by the probability it is explored (assuming uniform distribution over edges) to predict the size of the tree. To adapt WBE to runtime estimation, we replace the length of each branch with the time required to explore it.

Formally, the WBE updates its estimate every time it explores a branch, setting the estimate to:

$$estimate = \frac{\sum_{b \in B} t(b)}{\sum_{b \in B} p(b)}$$

where B is the set of explored branches, $t(b)$ is the time needed to explore a branch, and $p(b)$ is the probability of exploring the branch. For a branch $b = (v_1, \ldots, v_n)$, where v_i is the i-th node along the branch b, the probability $p(b)$ is defined as:

$$\prod_{i=1}^{n-1} \frac{1}{M(v_i) + F(v_i)}$$

where $M(v_i)$ is the number of marked children of node v_i and $F(v_i)$ is determined by the strategy.

3.2.2 Recursive Estimator (RE)

The *recursive estimator* (RE) is an online technique that estimates the size of an unvisited subtree as the average of the sizes of its visited siblings. To adapt RE to runtime estimation, we replace the size of each subtree with the time required to explore its branches.

Formally, when RE explores a branch $b = (v_1, \ldots, v_n)$, it updates the runtime estimate for every node along the branch in a bottom-up manner, setting the estimate to:

$$estimate(v_i) = \begin{cases} t(b) & \text{if } i = n \\ E_i \cdot \left(1 + \frac{F(v_i)}{n_{v_i}}\right) & \text{otherwise} \end{cases}$$

where E_i is the sum of the runtime estimates of subtrees rooted at visited children of v_i, n_{v_i} is the number of visited children of v_i, and $F(v_i)$ determined by strategy.

Space and Time Complexity

The WBE estimate needs to be updated upon two events: 1) when a new branch is explored and 2) when a new node is marked for exploration. To avoid recomputation of all of the values $p(b)$ and $t(b)$ for each update to the WBE, we store in each node v of the execution tree the sums:

$$\sum_{b \in B(v)} p(b) \text{ and } \sum_{b \in B(v)} t(b)$$

where $B(v)$ is the set of explored branches that contain the node v. When a new branch is explored, the aggregate probability and runtime values enable us to update the WBE estimate by changing only the values for the nodes along the current branch. Thus, although the time complexity of updating the WBE estimate for a new branch is linear in the length of the branch, it is amortized to $O(1)$ over the time needed to explore the branch. When a new node is marked for exploration, we need to update the aggregate probability and runtime values of the nodes along the current branch. The worst time complexity of this operation is $O(d)$, where d is the depth of the execution tree. Contrary to the case of exploring a new branch, the cost associated with marking node for exploration does not have constant amortized complexity. Thus, the overhead of computing the WBE estimate is potentially non-constant.

In contrast to the WBE estimate, the RE estimate needs to be updated only when a new branch is explored. The time complexity of this operation is linear in the depth of the execution tree but, as explained above, it is amortized to $O(1)$ over the time needed to explore the branch.

3.3 Fits

As the exploration progresses, new and presumably more accurate intermediate runtime estimates are computed. Previous work [8] considers the intermediate runtime estimates produced at distinct time points in a stateless exploration run in isolation, using only the latest estimate for decision making. In comparison, this paper treats the intermediate runtime estimates obtained in the course of a stateless exploration run as a sequence. We fit a function $f(t)$ to the sequence and then compute the actual runtime estimate by solving the equation $f(t) = t$.

Both intuition and experience suggests that the estimates produced by our techniques are initially inaccurate but converge to the correct value. Consequently, we choose a method that interpolates the estimates' behavior over time.

The method used in this paper is based on the Marquardt-Levenberg algorithm [13, 12] for weighted non-linear least-square fitting. We use the algorithm to find the values for coefficients of the function that

best fit the sequence of intermediate runtime estimates. To reflect the increasing confidence in estimates over time, the least-square fitting is weighted, using t as the weight for an estimate at time t.

In the context of this paper we consider four different fits:

- *Empty* fit: This scheme actually does no fitting. Rather, it emulates previous work [8], using the latest intermediate runtime estimate as the actual runtime estimate.
- *Constant* fit: $f(t) = c$. The advantage of using a constant fit is that the solution to the equation $f(t) = t$ is very likely to be a positive number (and thus meaningful). The disadvantage of using a constant fit is that it does not detect trends.
- *Linear* fit: $f(t) = a * t + b$. The advantage of using a linear fit is its ability to detect linear trends in the sequence of intermediate runtime estimates. However, the solution to the equation $f(t) = t$ can be negative and thus of no value.
- *Logarithmic* fit: $f(t) = a * ln(t) + b$. The advantage of using a logarithmic fit is its ability to detect non-linear trends in the sequence of intermediate runtime estimates. Further, if a solution to the equation $f(t) = t$ exists, it is guaranteed to be positive. The disadvantage of using a logarithmic fit is that a solution to the equation $f(t) = t$ may not exist.

Space and Time Complexity

The space and time complexity of the empty fit is $O(1)$. The space complexity of the other fits is linear in the length of the sequence being fitted. As for the time complexity of the other fits, the Marquardt-Levenberg algorithm uses a hill climbing technique. A single iteration of the algorithm is linear in the number of length of the sequence being curve. The number of iterations is potentially unbounded and depends on the desired precision. In other words, re-computing a fit with every new intermediate runtime estimate results in time complexity that is quadratic in the length of the sequence being fitted. For long sequences of intermediate runtime estimates, this overhead can be reduced by employing reservoir sampling [24].

3.4 Resource Allocation Mechanisms

Because stateless exploration might need a considerable amount of time to complete and a test suite of a large-scale system developer is expected to consist of many tests, it is not unreasonable to expect that the resources available for testing purposes are insufficient to complete all tests by a deadline. In this subsection we describe how to use runtime estimation to intelligently allocate machine cycles to a collection of stateless explorations to maximize testing objectives in the context of limited testing resources.

The generic mechanism maintains a priority queue of stateless exploration runs and the queue is used to identify which run to advance next. After a stateless exploration run is identified, a new branch of its underlying execution tree is explored, and its runtime estimate and its priority queue position are updated. This process is repeated for as long as there are unexplored branches and machine cycles available.

The order of the priority queue depends on the testing objective in use. In the context of this paper, we consider two intuitive objectives:

- *Maximize number of completed tests*: This testing objective is motivated by the guarantee realized upon completion of a stateless exploration run.
- *Achieve even coverage across tests*: This testing objective is motivated by balancing the values different tests provide towards establishing the confidence in the correct operation of the program under test.

For the first objective, stateless explorations are ordered by the remaining time estimate, which is computed by subtracting the elapsed time from the runtime estimate, following the "shortest remaining time first" policy.

For the second objective, stateless explorations are ordered by the coverage estimate, which is computed by dividing the elapsed time by the runtime estimate, following the "smallest coverage first" policy.

Space and Time Complexity

The space complexity of maintaining a priority queue is $O(k)$, where k is the number of stateless explorations. The time complexity of identifying the top element of a priority queue is $O(1)$, while the time complexity of updating the value of the top element is $O(\log k)$, where k is again the number of stateless explorations.

4 Evaluation

The goal of this section is to evaluate the following two hypotheses:

1. **Accuracy**: The estimation techniques described in Section 3 generate accurate runtime estimates.
2. **Efficiency**: The resource allocation mechanisms described in Section 3 outperform a baseline mechanism based on a round-robin policy.

To evaluate these hypotheses, we use a set of 10 exploration traces recently released by Google [18]. We wrote a trace simulator that reads these traces, simulates the exploration of the execution tree, and computes an intermediate runtime estimate every time a branch of the execution tree is explored. Additionally, we also created a program that inputs a sequence of intermediate runtime estimates and a fit type and uses the Marquardt-Levenberg algorithm to determine a function of that type with the best fit to the sequence.

The rest of this section first provides details about the exploration traces used for the evaluation and then addresses the above hypotheses.

4.1 Exploration Traces

The set of exploration traces used for our evaluation is summarized in Table 1. The table identifies the name of a test, the number of nodes of its execution tree, the number of branches of its execution tree, and the total time needed for the exploration. The unit of time is abstract as the timing of the exploration traces has been scaled by a magic constant as part of the trace anonymization process [18].

TEST NAME	# NODES	# BRANCHES	TIME
RESOURCE(2)	110	8	2.42
RESOURCE(3)	4,914	279	86.15
RESOURCE(4)	248,408	12,054	4,438.54
SCHEDULING(6)	29,578	720	250.80
SCHEDULING(7)	237,528	5,040	1,956.32
SCHEDULING(8)	2,142,164	40,320	19,868.90
STORE(3,3,7)	20,577	924	392.78
STORE(3,3,8)	88,386	3,790	1,715.49
STORE(3,3,9)	230,747	9,230	2,613.85
TLP	4,201,044	27,200	24,197.60

Table 1: Test Statistics

The RESOURCE(x) tests are representative of a class of tests that evaluate interactions of x different users that acquire and release resources from a pool of x resources. The SCHEDULING(x) tests are representative of a class of tests that evaluate handling of x concurrent scheduling requests. The STORE(x,y,z) tests are representative of a class of tests that evaluate interactions of x users of a distributed key-value store with y front-end nodes and z back-end nodes. Finally, the TLP test is representative of a class of tests that perform scheduling work.

4.2 Accuracy Evaluation

To evaluate the first hypothesis, we compare several estimation techniques based on different combinations of strategies, estimators, and fits. In the context of this subsection, accuracy of an estimation technique is evaluated by advancing a stateless exploration run for some time, generating intermediate runtime estimates after each test execution, using a fit to compute a prediction, and comparing the prediction to the correct value.

The experiment collected sequences of estimated runtimes computed during the simulation of the exploration traces RESOURCE(4), SCHEDULING(8), STORE(3,3,9), and TLP that are representative of the full set. For each test a sequence of runtime estimates was collected for all possible combinations of the RE and WBE estimators and the eager, hindsight, and lazy strategies.

For each combination of a test, a strategy, an estimator, and a sequence of runtime estimates, we then computed the empty, constant, linear, and logarithmic fit using the initial 1%, 5%, and 25% of the sequence.

Results

Figure 3 depicts the results of our initial accuracy measurements. The accuracy of the runtime estimate e with respect to the actual runtime a is computed as follows:

$$accuracy(e,a) = \begin{cases} 100 * (a/e)\% & \text{if } e > a \\ 100 * (e/a)\% & \text{otherwise} \end{cases}$$

For example, the accuracy of a runtime estimate that is 50% of the actual runtime is 50% while the accuracy of a runtime estimate that is 400% of the actual runtime is 25%. In other words, the graphs do not distinguish between under-estimation and over-estimation.

Figure 3 consists of three bar graphs, one for each percentage at which the fit was computed. Each of the bar graphs depicts the accuracy achieved for the 96 different combinations of test, strategy, estimator, and fit, with the results clustered by fit and test. Note that the vertical axis depicting the accuracy is in logarithmic scale. In some cases, the application of a fit did not generate a positive solution for the runtime estimate and in such cases the bar is missing.

Analysis

Our evaluation indicates that, unlike the eager strategy, the lazy strategy is consistent with the hindsight strategy. In other words, the lazy and the hindsight strategies tend to agree on which unmarked children will be marked in the future; in fact, in the case of the SCHEDULING tests and the TLP test, the lazy strategy and the hindsight strategy are indistinguishable. At the same time our evaluation indicates that the the hindsight strategy does not always produce the most accurate results. Surprisingly, the eager strategy occasionally produces the most accurate results and we attribute this fact to inaccuracy introduced by estimators and fits.

As for the estimators, our evaluation does not indicate that either of the two estimators consistently outperforms the other one. The weighted backtrack estimator, however, produces estimates that are, except

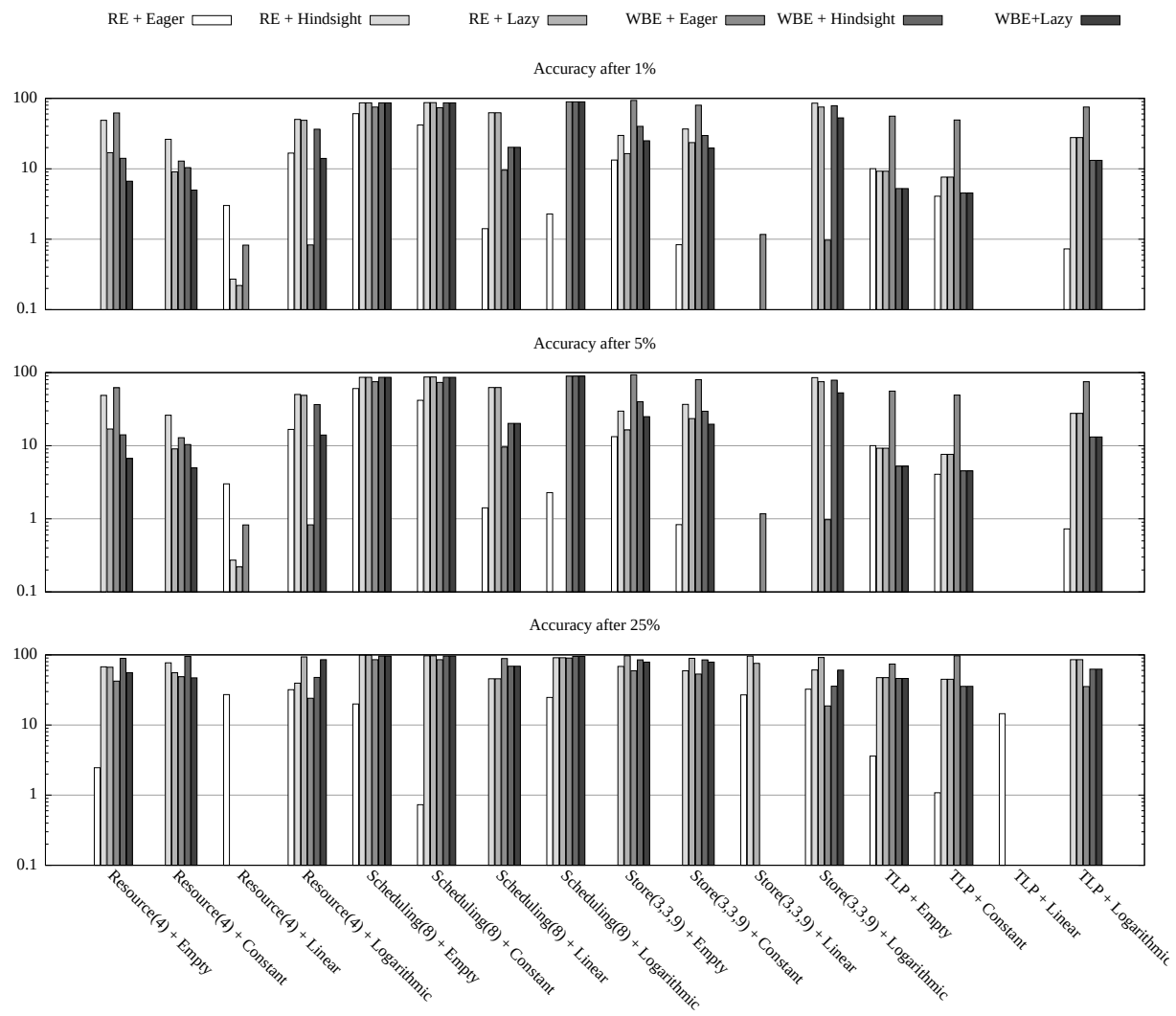

Figure 3: Accuracy of Runtime Estimation Techniques

for two cases, within an order of magnitude from the actual runtime. This cannot be said about the recursive estimator, which makes the weighted backtrack estimator more dependable.

Turning now to fits, the linear fit often fails to generate a positive solution, which makes it undependable. Comparing the empty fit to the constant fit, the empty fit produces better or equivalent results in most of the cases, which confirms our intuition that the intermediate runtime estimates grow more accurate with time. The logarithmic fit generates a solution in all but two cases, and compared to the empty fit, produces better or equivalent results in most of the cases.

To analyze the overall accuracy, we took a closer look at the best-performing techniques. Tables 2, 3, and 4 report the estimation error of the best performing estimation techniques after 1%, 5%, and 25% of the exploration respectively. The acronyms in the TECHNIQUE column have the following meaning: E+W+E means eager strategy, WBE, and empty fit, L+R+E means lazy strategy, RE, and empty fit, L+R+L means lazy strategy, RE, and logarithmic fit, and L+W+L means lazy strategy, WBE, and logarithmic fit.

Interestingly, the accuracy of the E+W+E technique does not improve over time, while the accuracies of the other techniques do. To understand this phenomenon, Figure 4 depicts the sequence of runtime estimates

Technique	Resource(4)	Scheduling(8)	Store(3,3,9)	TLP	Mean
E+W+E	62.11%	75.19%	93.46%	55.87%	68.97%
L+R+E	16.95%	86.21%	16.45%	9.24%	16.69%
L+R+L	48.78%	0.00%	75.19%	27.70%	42.92%
L+W+L	14.03%	89.29%	52.91%	13.18%	22.57%

Table 2: Accuracy of Best Techniques after 1%

E+W+E	45.66%	80.65%	91.74%	84.03%	69.93%
L+R+E	30.68%	92.59%	94.34%	15.22%	33.44%
L+R+L	63.69%	87.72%	40.82%	23.42%	42.37%
L+W+L	51.02%	94.34%	41.15%	23.58%	41.32%

Table 3: Accuracy of Best Techniques after 5%

E+W+E	42.19%	85.47%	59.17%	74.07%	60.61%
L+R+E	66.67%	99.01%	97.09%	47.39%	70.92%
L+R+L	93.46%	90.91%	91.74%	85.47%	90.09%
L+W+L	85.47%	95.24%	60.61%	62.50%	72.99%

Table 4: Accuracy of Best Techniques after 25%

over time for selected tests and best-performing techniques. Each graph also includes a horizontal line that identifies the correct runtime.

As expected, the eager strategy leads to over-estimation, while the lazy strategy leads to under-estimation. Further, the value of the runtime estimate can change quickly over time, which suggests that the performance of the estimation techniques based on the empty fit is sensitive to the time at which the fit is evaluated.

Thus, although at first glance the E+W+E technique seems to be the best, our further investigation suggests that the L+W+L technique might be a more robust choice. We base this conclusion on two facts: 1) the resilience to spikes and drops in the sequence of the runtime estimate provided by the logarithmic fit and 2) the accuracy of fitting a logarithm to a sequence of runtime estimates generated by the combination of the lazy strategy and WBE.

In summary, the most accurate techniques examined in this paper consistently achieve average accuracy above 60% after exploring as little as 1% of the state space. If we were to choose a technique to deploy in production, our recommendation would be to use a technique that combines the lazy strategy, the weighted backtrack estimator, and the logarithmic fit.

4.3 Efficiency Evaluation

In this subsection, we evaluate the potential of runtime estimation to implement allocation policies that target the testing objectives introduced in Section 3.

4.3.1 Maximizing # of Completed Tests

To evaluate how well runtime estimation techniques aid in maximizing the number of completed tests, we extended our trace simulator with a priority queue that tracks the remaining time for each stateless exploration run and a scheduler that advances stateless exploration runs according to the priority queue.

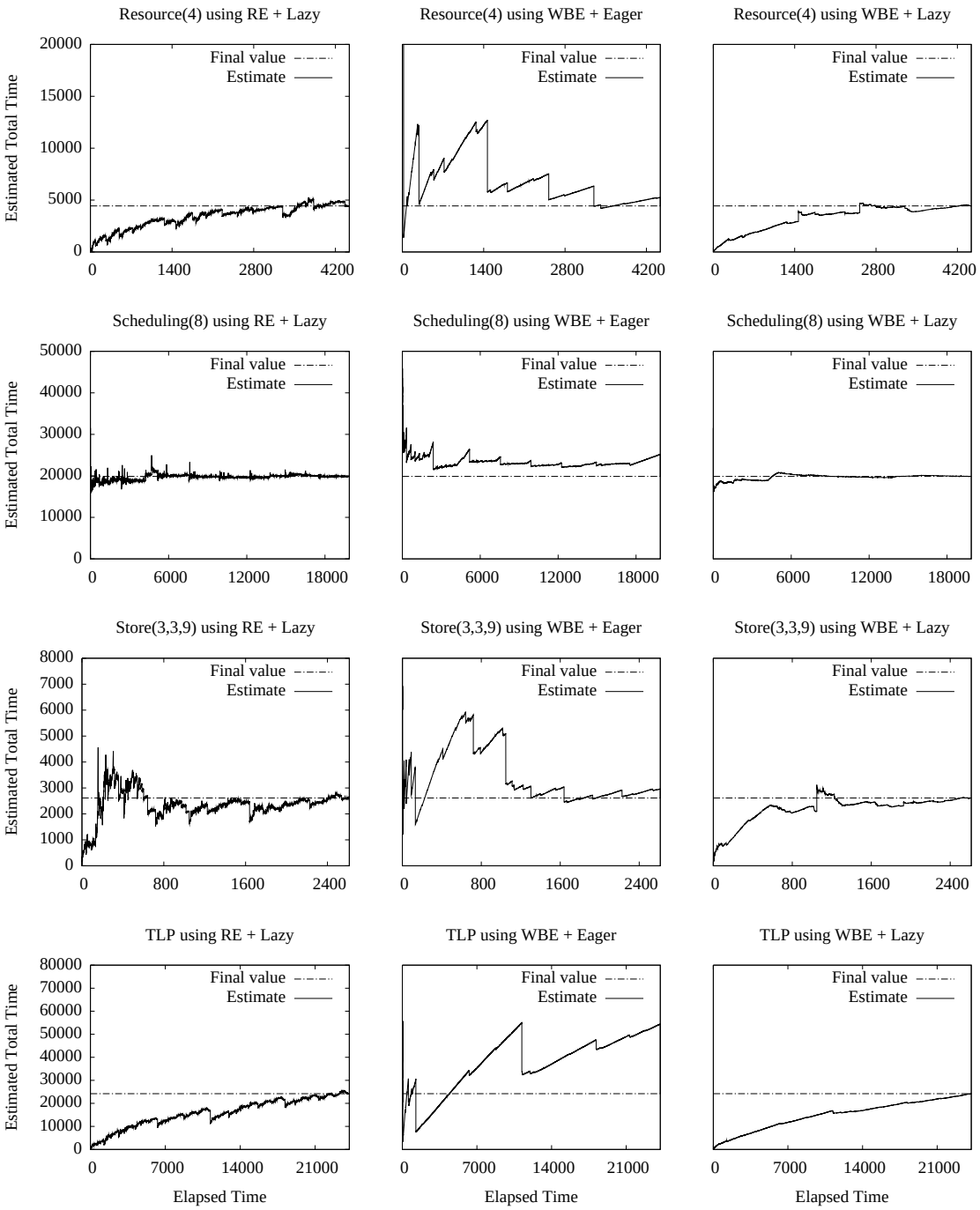

Figure 4: Evolution of Runtime Estimates Over Time

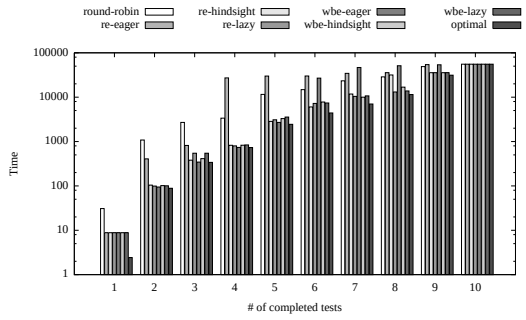

Figure 5: Maximizing # of Completed Tests

COMPLETED	RE-EAGER	RE-LAZY	WBE-EAGER	WBE-LAZY	OPTIMAL
1	0.29	0.29	0.29	0.29	0.08
2	0.38	0.09	0.09	0.09	0.08
3	0.30	0.20	0.13	0.20	0.13
4	8.08	0.24	0.22	0.25	0.22
5	2.58	0.27	0.23	0.31	0.21
6	2.03	0.49	1.82	0.50	0.30
7	1.47	0.45	2.00	0.46	0.30
8	1.24	0.46	1.79	0.48	0.40
9	1.11	0.73	1.10	0.73	0.64
10	1.00	1.00	1.00	1.00	1.00
MEAN	1.85	0.42	0.87	0.43	0.34

Figure 6: Performance of Allocation Algorithms Relative to the Baseline Algorithm (lower is better)

We examine all combinations of the RE and WBE estimators and the eager, hindsight, and lazy strategies. Given the frequency of runtime estimate computations in this experiment, we only consider the empty fit, which is the only fit with constant time and space complexity.

For the sake of comparison, we also examine two additional scheduling algorithms: 1) a *baseline* algorithm, which selects the next test to run using round-robin, representing an approach commonly used in practice, and 2) an *optimal* algorithm, which cheats by knowing the actual runtime of each tests and executes tests from the shortest to the longest.

Lastly, we provided the simulator with unlimited time and the 10 exploration traces and recorded the times at which different techniques completed each. In other words, instead of using a fixed time budget, we recorded data that allows us to derive the results for an arbitrary fixed time budget.

Results

Figure 5 presents a bar graph, which for each of the allocation algorithms plots the time needed to complete a number of tests. The horizontal axis shows the number of completed tests, while the vertical axis shows time in logarithmic scale. Note that the measurement is indifferent to the order in which the tests finish and only compares the times required by different algorithms to complete a certain number tests.

Analysis

Our results indicate that the algorithms that incorporate the eager strategy tend to perform poorly and, in some cases, need more time to complete a certain number of tests than the baseline algorithm. In contrast to that, the algorithms that incorporate the hindsight and the lazy strategies perform comparably to the optimal algorithm. In most cases, these algorithms require a fraction of the time required by the baseline algorithm to complete a certain number of tests. Further, our results indicate than in the context of this experiment, the choice of the estimator is not significant.

Figure 6 reports the fractions of the time required by the best performing algorithms with respect to the baseline algorithm. The rows of the table report these fractions for each number of completed tests and the last row reports the average.

Thus, our experiments indicate that the allocation algorithms based on a combination of the lazy strategy, either of the two estimators, and the empty fit, reduce the time needed by the baseline approach to complete a certain number of tests by 2.38× on average, while the theoretical maximum is 2.94×.

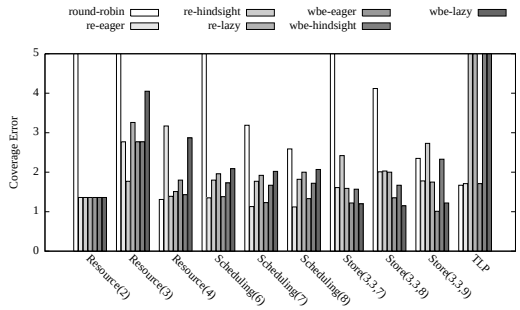

Figure 7: Achieving Even Coverage

TEST	ROUND-ROBIN	RE-EAGER	RE-LAZY	WBE-EAGER	WBE-LAZY
RESOURCE(2)	11.11	1.36	1.36	1.36	1.36
RESOURCE(3)	11.11	2.77	3.26	2.77	4.05
RESOURCE(4)	1.31	3.17	1.51	1.80	2.87
SCHEDULING(6)	11.11	1.35	1.96	1.38	2.09
SCHEDULING(7)	3.19	1.13	1.92	1.23	2.02
SCHEDULING(8)	2.59	1.12	2.00	1.33	2.07
STORE(3,3,7)	11.11	1.61	1.59	1.22	1.20
STORE(3,3,8)	4.12	2.01	2.00	1.35	1.15
STORE(3,3,9)	2.35	1.78	1.75	1.01	1.22
TLP	1.67	1.71	8.15	1.71	7.12
MEAN	5.97	1.80	2.55	1.52	2.51

Figure 8: Coverage Error of Allocation Algorithms

4.3.2 Achieving Even Coverage

To evaluate how well the runtime estimation techniques aid in achieving even coverage across the test suite, we modified the ordering the priority queue uses to order elements from an ordering based on estimated remaining time to an ordering based on estimated coverage.

Similarly to the previous experiment, we examined all combinations of the strategies and estimators discussed in this paper and the empty fit. Further, we examined a baseline algorithm that selects the tests to run using a round-robin policy. In contrast to the previous experiment, we simulated the scarcity of testing resources by setting the machine time budget to $5,000$ time units.

For each test, we measured the *coverage error* that is, the relative difference between the realized and the optimal coverage. Formally, let t be sum of the actual runtimes of all tests in a test suite and $m \leq t$ be the amount of available machine time, then the *even coverage* is $e = \frac{m}{t}$. Further, let t be the time needed to fully explore the test and t^* be the time spent exploring the test, then the *realized coverage* is $r = \frac{t^*}{t}$ and the *coverage error* of the realized coverage r with respect to the even coverage e, denoted $error(r,e)$ is defined as:

$$error(r,e) = \begin{cases} r/e & \text{if } e > r \\ e/r & \text{otherwise} \end{cases}$$

Results

Figure 7 presents a bar graph, which for each of the allocation algorithms and each of the tests plots the achieved coverage error. The horizontal axis is used to cluster the results by test, while the vertical axis plots the coverage error. Note that the graphed data does not distinguish between under-shooting or over-shooting the even coverage.

Analysis

Our results indicate that the algorithms that incorporate the hindsight and the lazy strategies tend to perform poorly and in some cases even achieve higher coverage error than the baseline algorithm. In contrast to that, the algorithms based on the eager strategy often produce the best result, always achieving a coverage error of less than 3. Further, the algorithms that incorporate the weighted backtrack estimator tend to perform marginally better than those that incorporate the recursive estimator.

Table 8 compares the coverage errors of the best performing algorithms to that of the baseline algorithm. The rows report the coverage errors for individual tests and the last row reports the average. Thus, our experiments indicate that allocation algorithms based on a combination of the eager strategy, either of the

two estimators, and the empty fit, reduce the coverage error of the baseline approach by 3.31× on average, while the theoretical maximum is 5.97×.

5 Related Work

The related work comes from two categories: research on state space estimation and research on resource allocation.

State Space Estimation

The estimators in this paper adapt the work on estimating search tree size of Kilby et al. [8] to the problem of estimating stateless exploration runtime. In their evaluation, Kilby et al. present the accuracy of their estimation techniques achieved during exploration of search trees corresponding to both decision and optimization problems.

Similar to our work, the techniques of Kilby et al. can be described as online and passive. However, in contrast to search tree size estimation of Kilby et al., our techniques address the non-linear nature of stateless exploration due to DPOR. Nonetheless, comparing the accuracy of our estimation techniques on stateless exploration runs to Kilby et al. estimation techniques on search trees, our techniques perform equally well on a harder problem.

In related work, Taleghani and Atlee [19] studied the problem of state space coverage estimation for explicit-state model checking. Their solution is based on Monte Carlo techniques and complements a state space exploration with random walks to estimate the ratio between visited and unvisited states. In their evaluation, Taleghani and Atlee implemented their technique inside of Java PathFinder (JPF) [23] and evaluated it using a collection of Java programs.

In contrast to our work, the technique of Taleghani and Atlee is limited to stateful approaches and can be described as offline and active. More precisely, the estimate is computed only once at the end of the exploration and the computation relies on a particular state space exploration algorithm. Although Taleghani and Atlee do not explicitly mention whether their experiments were carried out in the context of state space reduction, since JPF supports it, we assume they were. If that is the case, our technique performs equally well on a similar problem but does not rely a specific exploration algorithm.

Resource Allocation

Dynamic allocation of resources to a collection of independent tasks is both a well studied theoretical problem [21] and a practical problem addressed by a range of systems ranging from batch schedulers such as the Maui scheduler [7] to platforms for sharing resources in a data center [6].

While in practice [6, 7] tasks are usually running concurrently on a cluster, this paper experiments with a simple model that schedules tasks sequentially. This simplification is justified by the unique nature of a stateless exploration run, which typically consists of many executions of the same test. Recording progress of each stateless exploration run using an execution tree thus enables low-overhead fine-grained interleaving of concurrent stateless exploration runs. In addition, different executions of the same test can be explored in parallel, enabling linear speed-up [16]. In other words, representing a collection of machines as a sequence of machine cycles is a reasonable abstraction for a large collection of very small independent tasks.

Another unique aspect of allocation of resources among stateless exploration runs is how value accrues. In general, value of a task accrues only upon its completion, which is reflected in scheduling objectives that minimize average task latency [7] or maximize task throughput [21]. In the case of a stateless exploration run, value can accrue at any point in time and a part of our evaluation is based on an objective that uses coverage as a measure of value.

Interestingly, the problem of deciding which stateless exploration run to advance next is similar to the problem of multi-armed bandit [22]. In short, a multi-armed bandit problem for a gambler is to decide which arm of an n-slot machine to pull to maximize his total reward in a series of trials. To bridge the research on multi-armed bandits with our work, we need to define a reward function for stateless exploration runs. The search for a good reward function definition seems to be an interesting avenue for future work.

6 Conclusion

This paper presents a solution to the problem of allocating resources to a collection of stateless exploration runs. The solution comes in the form of different allocation algorithms that are based on testing objectives and techniques for estimating the runtime of a stateless exploration run.

In this paper, an estimation technique consists of three components: a strategy, an estimator, and a fit. We investigated three strategies: eager, hindsight (infeasible in practice), and lazy; two estimators: weighted backtrack and recursive; and four fits: empty, constant, linear, and logarithmic; for a total of 16 feasible estimation techniques.

In the course of our experimental evaluation of these estimation techniques, we arrived at a number of conclusions: 1) the eager strategy tends to over-estimate the total runtime, 2) an estimate based on the eager strategy can decrease sharply if DPOR backtracks from a node with unexplored children, 3) the lazy strategy tends to under-estimate the total runtime, 4) a sequence of runtime estimates produced by WBE in the course of a stateless exploration run is more stable than a sequence of runtime estimates produced by RE, 5) the empty fit is sensitive to spikes and drops in the sequence of runtime estimates, while the other fits are not, 6) the empty fit generates more accurate estimates than the constant fit, 7) the linear fit often fails to generate an estimate, 8) the logarithmic fit is a good match for the combination of the lazy strategy and the weighted backtrack estimator.

Further, we observed the overall average accuracy of our best estimation techniques to be above 60% after exploring as little as 1% of the state space. Although such accuracy is sufficient for making manual or automated decision based on the estimated runtime, we would like to do better in the future. We believe the sub-optimal accuracy of our best estimation techniques stems from 1) the non-linear nature of DPOR progress through an execution tree and 2) from unexpected patterns in the structure of an execution tree.

We conjecture that, similar to power-law in random networks graphs [1], execution trees generated by stateless exploration of concurrent programs have common structural patterns. In our future work, we plan to identify these common structural patterns and use them to improve the accuracy of our estimation techniques.

Besides evaluating the accuracy of our estimation techniques, this paper also investigated the ability of these techniques to allocate resources to a collection of stateless exploration runs heeding a testing objective. We considered two testing objectives: 1) maximizing the number of completed tests and 2) achieving even coverage between different tests. For each of these objectives we designed an allocation algorithm parametrized by an estimation technique. We then evaluated the performance of our estimation techniques against a baseline algorithm based on a round-robin policy and against the theoretical optimum.

In the course of our experimental evaluation of these allocation algorithms, we arrived at the following conclusion. While, the lazy strategy outperformed the eager strategy at meeting the first testing objective, the eager strategy outperformed the lazy strategy at meeting the second testing objective. We believe that this is an indication of the absolute estimates being more accurate when generated using the lazy strategy, while the ratios between these absolute estimates for different stateless exploration being more accurate when generated using the eager strategy.

Further, our evaluation demonstrated that, for the two testing objectives considered in this paper, our allocation algorithms narrow the gap between the baseline approach and the theoretical optimum. In future

work, we would like to experiment with other testing objective, such as maximizing the number of bugs found, which we did not consider in this paper since the exploration traces from Google do not contain such information.

References

[1] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286:509–512, 1999.

[2] Katherine E. Coons, Sebastian Burckhardt, and Madanlal Musuvathi. GAMBIT: Effective Unit Testing for Concurrency Libraries. *SIGPLAN Not.*, 45(5):15–24, January 2010.

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

[4] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*. Springer-Verlag, 1996.

[5] P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *POPL 1997*, pages 174–186. ACM, 1997.

[6] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: a platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, Berkeley, CA, USA, 2011. USENIX Association.

[7] David B. Jackson, Quinn Snell, and Mark J. Clement. Core Algorithms of the Maui Scheduler. In *Proceedings of the 7th International Workshop on Job Scheduling Strategies for Parallel Processing*, JSSPP '01, pages 87–102, London, UK, UK, 2001. Springer-Verlag.

[8] Philip Kilby, John K. Slaney, Sylvie Thiébaux, and Toby Walsh. Estimating search tree size. In *AAAI*, pages 1014–1019, 2006.

[9] C. E. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *NSDI 2007*, 2007.

[10] Donald E. Knuth. Estimating the Efficiency of Backtrack Programs. *Mathematics of Computation*, 29(129):121–136, 1975.

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

[12] Kenneth Levenberg. A Method for the Solution of Certain Non-linear Problems in Least Squares. *Quart. J. Appl. Maths.*, 2(2):164–168, 1944.

[13] Donald W. Marquardt. An Algorithm for Least-Squares Estimation of Nonlinear Parameters. *Journal of the Society for Industrial and Applied Mathematics*, 11(2):431–441, 1963.

[14] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. In *OSDI 2008*, pages 267–280, 2008.

[15] J. Simsa, R. Bryant, G. Gibson, and J. Hickey. Efficient Exploratory Testing of Concurrent Systems. *CMU-PDL Technical Report*, 113, November 2011.

[16] J. Simsa, R. Bryant, G. Gibson, and J. Hickey. Scalable Dynamic Partial Order Reduction. In *Proceedings of the 3rd International Conference on Runtime Verifications*, RV'12, 2013.

[17] J. Simsa, G. Gibson, and R. Bryant. dBug: Systematic Evaluation of Distributed Systems. In *SSV 2010*, 2010.

[18] Jiri Simsa and John Wilkes. ETA exploration traces. Google research blog, September 2012. Posted at http://code.google.com/p/googleclusterdata/wiki/ETAExplorationTraces.

[19] Ali Taleghani and Joanne M. Atlee. State-Space Coverage Estimation. In *ASE*, pages 459–467, 2009.

[20] S. S. Vakkalanka, S. Sharma, G. Gopalakrishnan, and R. M. Kirby. ISP: A tool for model checking MPI programs. In *PPoPP 2008*, pages 285–286, 2008.

[21] Rob van Stee. *Combinatorial algorithms for packing and scheduling problems*. PhD thesis, Universität Karlsruhe, June 2008.

[22] Joannès Vermorel and Mehryar Mohri. Multi-armed bandit algorithms and empirical evaluation. In *Proceedings of the 16th European conference on Machine Learning*, ECML'05, pages 437–448, Berlin, Heidelberg, 2005. Springer-Verlag.

[23] Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, April 2003.

[24] Jeffrey S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, March 1985.

[25] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. MoDist: Transparent Model Checking of Unmodified Distributed Systems. In *NSDI 2009*, pages 213–228, April 2009.