Parallel symbolic execution for structural test generation — Source link ⧉

Matt Staats, Corina S. Păsăreanu

Institutions: University of Minnesota, Ames Research Center

Published on: 12 Jul 2010 - International Symposium on Software Testing and Analysis

Topics: Symbolic execution, Symbolic trajectory evaluation, Concolic testing, Speedup and Test case

Related papers:

- Symbolic execution and program testing

- DART: directed automated random testing

- KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs

- CUTE: a concolic unit testing engine for C

- Parallel symbolic execution for automated real-world software testing

Parallel Symbolic Execution for Structural Test Generation[*]

Matt Staats
Dept. of Computer Science and Eng.
University of Minnesota
staats@cs.umn.edu

Corina Păsăreanu
Carnegie Mellon University/NASA Ames
Research Center
Moffett Field, CA, 94035
Corina.S.Pasareanu@nasa.gov

ABSTRACT

Symbolic execution is a popular technique for automatically generating test cases achieving high structural coverage. Symbolic execution suffers from scalability issues since the number of symbolic paths that need to be explored is very large (or even infinite) for most realistic programs. To address this problem, we propose a technique, *Simple Static Partitioning*, for parallelizing symbolic execution. The technique uses a set of pre-conditions to partition the symbolic execution tree, allowing us to effectively distribute symbolic execution and decrease the time needed to explore the symbolic execution tree. The proposed technique requires little communication between parallel instances and is designed to work with a variety of architectures, ranging from fast multi-core machines to cloud or grid computing environments. We implement our technique in the Java PathFinder verification tool-set and evaluate it on six case studies with respect to the performance improvement when exploring a finite symbolic execution tree and performing automatic test generation.

We demonstrate speedup in both the analysis time over finite symbolic execution trees and in the time required to generate tests relative to sequential execution, with a maximum analysis time speedup of 90x observed using 128 workers and a maximum test generation speedup of 70x observed using 64 workers.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Experimentation,Verification

Keywords

symbolic execution, Java Pathfinder, parallel

1. INTRODUCTION

Software testing is a commonly used technique for validating the quality of software. It is typically a manual process that accounts for half of the total cost of software development and maintenance [4]. The automation of testing not only reduces the cost of producing software but also increases software reliability by enabling more thorough testing.

Symbolic execution [16] is a fully automated technique for generating test cases to achieve high testing coverage, and (together with variations such as concolic execution) has become a popular approach in recent years [14, 21]. Symbolic execution is performed by executing programs with symbolic, rather than concrete, inputs. Execution branches when a condition (e.g., an *if* statement) is encountered, with the condition constraining the inputs on one branch and the negation of the condition constraining the inputs on the other branch. Test generation is performed by solving the resulting constraints using a constraint solver.

The paths followed during symbolic execution form a *symbolic execution tree*, representing all the possible executions through the program. However exploring *all* the possible program executions is generally infeasible (the symbolic execution tree may be very large or even infinite) thus limiting the application of symbolic execution in practice. One way to address this scalability problem is to use a compositional approach [11] that analyzes functions in isolation, encodes the analysis results as summaries and reuses these summaries to analyze high-level functions.

We explore here a different, complementary approach that improves the scalability of symbolic execution via parallelization. Our approach is motivated by the increased availability of multi-core computers and the inherently parallelizable nature of symbolic execution. Just as exploring a binary tree can be parallelized with little to no inter-process communication, so can exploring a symbolic execution tree. This is an important advantage, as previous work in parallel model checking has indicated that synchronization overhead can be significant, potentially negating any benefits [12, 24]. Consequently, we investigate approaches to parallel symbolic execution requiring little or no inter-process communication.

More specifically, we use a set of pre-conditions to *partition* the symbolic execution tree, allowing us to effectively distribute symbolic execution and decrease the time needed to explore the symbolic execution tree. The pre-conditions are computed by first performing a "shallow" symbolic execution of the user code to automatically collect a large list of path conditions. These path conditions are then processed to produce a list of constraints to be used as pre-conditions. We refer to this technique as Simple Static Partitioning (SSP).

We also propose a generic client-server framework to perform test generation using multiple machines and/or cores. The frame-

work is designed primarily for flexibility and extensibility, and thus can operate on a variety of architectures, ranging from multi-core machines with shared memory, to networked desktop machines, to cloud or grid computing environments. Our framework is built on top of the Java PathFinder (JPF) [26] tool-set, using Symbolic PathFinder (SPF) [17], a JPF extension that performs symbolic execution for Java bytecode. Using this framework, we have developed listeners for coordinating parallel automatic test generation, including coverage measurement and test suite reduction.

We have implemented Simple Static Partitioning within this framework. We have evaluated its effectiveness in terms of (1) the speedup in the time required to completely explore a finite symbolic execution tree relative to sequential symbolic execution, and (2) the speedup in the time required to generate a test suite achieving a predetermined level of *Modified Condition/Decision Coverage* (MC/DC) [6] relative to one or more parallel instances of random depth first search. We used six case examples in our evaluation: an Altitude Switch from the avionics domain, a Wheel Brake System from the automotive domain and four data structures commonly used in conjunction with Java Pathfinder [27].

Our results demonstrate analysis time speedups up to 30x for 3 of 6 systems using 64 workers, with a maximum speedup of 90x observed using 128 workers. For small numbers of workers (2-8) we demonstrate speedups consistently larger than 90% of the maximum (linear) speedup for 3 of 6 systems, with the other three systems demonstrating speedups of 30% to 90% of the maximum speedup depending on the number of workers. Finally, we demonstrate consistent and significant speedup in automatic test generation over parallel random depth first search for systems requiring on average at least 10 minutes when using random depth first search, with speedups ranging between 5.3x and 70x using 64 workers.

Our approach is related to previous work on parallelizing software model checking [9, 12, 13, 24]. However, all of these are done in the context of explicit state space exploration. There is little work on parallelizing symbolic execution. We are aware of two such approaches [7, 15], both of them very recent. Unlike our approach, they both operate primarily by *dynamically* partitioning the symbolic execution tree for load balancing. The work closest to ours is King's master thesis [15], which uses a queue of subtrees for exploration. However this queue is populated dynamically, during execution, and not statically as in our case. King demonstrates consistent speedup in analysis time for a small number of workers (2-4), but with decreases in speedup as the number of workers increases to 6 or 8. In contrast, our approach does not exhibit drops in overall speedup as the number of cores increases. We provide a more extensive comparison in Section 7.

Our contributions are thus: (1) the description and implementation of an effective technique for statically partitioning a symbolic execution tree and distributing the partitions across parallel instances, (2) the development of a flexible, extensible framework for parallelizing Java Pathfinder, and (3) an evaluation of our work in terms of the speedup when exploring a finite symbolic execution tree and the performance of automatic test generation.

The paper is organized as follows: Section 2 gives background on Java Pathfinder and Symbolic Pathfinder, Sections 3 and 4 outline our partitioning technique and the parallel framework, Sections 5 and 6 evaluate our technique and discuss the implications, and finally Sections 7 and 8 discuss related work and conclude.

2. JAVA PATHFINDER

Java Pathfinder (JPF) is an open-source tool-set for verifying Java bytecodes. It includes an explicit-state model-checker (core-JPF) and several extensions, e.g. Symbolic PathFinder and Com-plexCoverage, that we use in our work. The model checker consists of an extensible custom Java Virtual Machine (JVM), listener support for monitoring and influencing JPF's search, and a set of Java methods for instrumenting Java programs. JPF's default mode of execution, termed *concrete execution*, performs explicit-state model checking over Java bytecode.

JPF has been designed to be extensible, and consequently much of JPF's execution can be monitored or replaced. Our work extends JPF via two mechanisms: *ChoiceGenerators* and *JPF listeners*. *ChoiceGenerators* are the mechanism by which JPF explores the state space. *ChoiceGenerators* correspond to non-deterministic choices made during execution, and often are generated by instrumentation in the source code being explored. *JPF listeners* monitor JPF's exploration of the search space, the operation of JPF's custom JVM, and the creation and execution of *ChoiceGenerators*. Often, JPF listeners are used to selectively influence execution – a listener can monitor for a specific event and modify decisions made by JPF, including modifying the operation of *ChoiceGenerators*.

Improvements or enhancements that build upon JPF are referred to as *JPF extensions* or simply *extensions*. There exist several extensions to JPF relevant to automatic test generation. In this work, however, only two extensions are used: Symbolic Pathfinder (SPF), an extension to JPF for performing symbolic execution, and Com-plexCoverage, an extension which improves upon JPF and SPF's test generation functionality by adding functionality for measuring complex structural coverage criteria such as MC/DC. We briefly describe the relevant aspects of these extensions below; see [17] and [23] for details.

2.1 Symbolic Pathfinder (SPF)

Symbolic execution is a form of program analysis in which symbolic values are used instead of concrete input values. The program state is represented by the symbolic values of the program variables, the program counter (i.e., next bytecode to be executed) and the path condition (*PC*). The path condition is a boolean formula representing the constraints that must be satisfied by the symbolic values for execution to progress on the current path, i.e. it encodes the *pre-condition* to follow that path.

SPF implements symbolic execution on top of JPF using a non-standard bytecode interpretation. JPF's search mechanism is used to generate and explore the symbolic execution tree. The key implementation details are (1) replacing JPF's standard bytecode interpretation, (2) using variable attributes to store symbolic information, and (3) generating *PCChoiceGenerators* when executing conditional bytecodes.

JPF is implemented with an abstract bytecode factory, allowing developers to change how bytecodes are interpreted. The standard bytecode interpretation performs concrete execution, using real values (e.g., integers, floats, etc.) to explore the state space of the bytecodes. SPF extends these bytecodes to allow variables to be represented by symbolic values and expressions, and propagates symbolic values and expressions when executing said bytecodes. Storage of symbolic values and expressions is accomplished by assigning symbolic attributes to variables, fields, and stack operands. Note that SPF's bytecodes are a true extension of the standard concrete bytecodes, and thus both concrete values and symbolic values can be used during the same execution. We term this feature *mixed-mode execution*.

When a conditional bytecode (e.g., those compiled from *if* statements, *switch* statements, etc.) is executed in SPF, execution branches to explore the result of the bytecode evaluating to *true* or *false*. A choice generator, *PCChoiceGenerator*, is used to nondeterministically choose which branch to explore. By default, two choices, *true*

```
[1]   int r;
[2]   while (true) {
[3]     boolean b = Debug.getSymbolicBoolean("b");
[4]     int x = Debug.getSymbolicInteger("x");
[5]     int y = Debug.getSymbolicInteger("y");
[6]
[7]     if (b) {
[8]         Debug.storeTraceIf(x>y, "ob1");
[9]         if (x > y)
[10]            r = x;
[11]        else
[12]            r = y;
[13]    }
[14] }
```
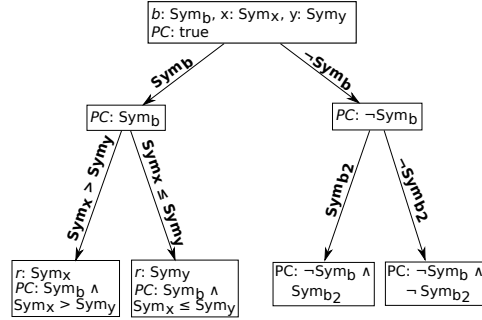
Figure 1: Symbolic Execution Example (Depth of 3)

and *false*, are generated by a *PCChoiceGenerator*. Each choice generated is associated with a path condition – the bytecode's condition if *true* and the negation of the bytecode's condition if *false*. When a choice is explored, the bytecode evaluates to this choice and the associated path condition is appended to the *PC*. When branching execution, the satisfiability of the path condition is checked using off-the-shelf constraint solvers. If the *PC* is satisfiable, JPF continues along the associated path; otherwise, JPF backtracks. By default, SPF always explores the branch corresponding to *false* first.

We present an example in Figure 1. For symbolic execution, on lines *3-5*, the program variables *x, y* and *b* are assigned symbolic variables using the ComplexCoverage extension described in Section 2.2. As execution progresses, the variable *r* is expressed in terms of *x* and *y*. At each conditional bytecode, execution branches to explore the bytecode evaluating to *true* and *false*. Upon branching, the appropriate constraint, shown on each arrow, is appended to the current *PC* (see Figure 1, right). For example, two paths lead from the top of the symbolic execution tree. The left path corresponds to line *7* evaluating to *true* and adds Sym_b to the *PC*; the right path corresponds to line *7* evaluating to *false* and adds $\neg Sym_b$ to the *PC*. The left path adds additional constraints relating x and y; the right path returns to the top of the loop and generates new symbolic variables (indicated by the subscript *2*). Solving the constraints encoded in a *PC* give the *test inputs* that exercise that path.

SPF, unlike standard JPF, does not perform state matching as this is in general undecidable for path conditions on unbounded data. To prevent SPF from attempting to explore a potentially infinite search space, an upper limit is placed on the search depth or on the number of constraints in the path condition. Thus SPF's search is characterized as a finite *symbolic execution tree* as shown in Figure 1, for a depth of 3 (choices). This is in contrast to concrete execution in JPF, which is characterized as a potentially cyclic graph. This property, as we will see in Section 3, is the key to partitioning symbolic execution across multiple parallel instances.

2.2 ComplexCoverage

The ComplexCoverage extension provides functionality for both JPF and SPF related to generating test suites satisfying complex structural coverage criteria. The functionality includes improved support for instrumentation, measuring coverage achieved by test suites, and reducing the size of test suites while maintaining coverage. Generating tests using ComplexCoverage relies on an instrumentation-based approach, in which each *test obligation* required by the coverage metric is specified as instrumentation. A *test obligation* for a structural coverage metric is a condition that must evaluate to *true* at a specific point in the source code. For example, on line *8* in Figure 1, the obligation *x > y at line 8* is expressed

as *Debug.storeTraceIf(x>y, "ob1")*. If this statement is executed when *x>y* is satisfiable, it will store test inputs satisfying the obligation. When symbolic execution terminates, the *coverage* achieved is computed as the percentage of satisfied obligations.

The ComplexCoverage extension provides support for "tagging" instrumentation with identifying strings, thus providing a link to symbolic variables used by SPF and program variables in the code. Additionally, it supports measuring the wallclock or CPU time required to generate each test, needed for our evaluation.

We perform our evaluation in Section 5 using *MC/DC coverage* [6], a rigorous structural metric widely used in critical system domains such as avionics software. Test suites providing MC/DC coverage must cause each condition to evaluate to *true* and *false* and must demonstrate that each condition affects the value of the expression. Our use of MC/DC is motivated in part by an ongoing NASA project that aims at analyzing Simulink/Stateflow models using JPF/SPF via translation into Java [18]. Furthermore, we use MC/DC as achieving MC/DC is more difficult than achieving other structural coverage criteria (such as statement or condition coverage) and thus represents a more significant challenge.

3. PARTITIONING FOR PARALLEL SYMBOLIC EXECUTION

Our goal is to parallelize symbolic execution in a manner useful for a variety of environments, including fast multi-core machines with shared memory architectures, networked desktop machines, and cloud or grid computing environments. As previous work in parallelized model checking has indicated that synchronization overhead can be significant, potentially negating any benefits [12, 24], we are only interested in methods that require very little communication.

We describe here our method of parallelization that statically partitions and distributes execution by (1) generating a queue of constraints and (2) distributing these constraints (as part of a standard JPF configuration) to workers, which use the constraints to direct symbolic execution. We term this method *Simple Static Partitioning (SSP)*. In our evaluation, we compare SSP against another method of parallelization, parallel *random depth first search (RDFS)*, described next. Both techniques, given sufficient time, will explore an entire finite symbolic execution tree. Both techniques are implemented as JPF listeners.

3.1 Random Depth First Search

Random depth first search (RDFS) performs depth first search while randomly selecting the order branches in the tree are explored. It differs from SPF's standard search technique only in the use of randomization. While a simple technique, it has very low overhead and is amenable to parallel execution via different

random seeds. Furthermore, previous research has shown parallel RDFS to be an effective method of detecting errors [9, 13]. RDFS acts as a baseline for evaluating other techniques – if a technique cannot outperform RDFS, it is unlikely to be of practical use.

Whenever a *PCChoiceGenerator* is created, the technique randomly decides the order each choice will be explored using Java's default random number generator (seeded with the current system time or with a user-provided seed). By default, both orderings (*true*, *false* and *false*, *true*) are equally likely; however, the technique can be configured to favor one ordering over another.

3.2 Simple Static Partitioning (SSP)

We partition the traversal of a symbolic execution tree by using a set of constraints over the input variables of the program under analysis. These constraints are distributed to parallel *workers* which use them as pre-conditions for the symbolic execution performed with SPF. For each pre-condition, SPF explores only a subset of the symbolic execution tree, representing only the program executions for the inputs that satisfy the pre-condition. To compute the set of constraints, we first perform a "shallow" symbolic execution of the code under analysis and process the path conditions (PCs) that are generated automatically. SPF's standard depth first search or RDFS can be used to explore the tree.

To effectively partition and distribute execution, we require a set of constraints to be *disjoint* and *complete*. We also desire the set to be *useful*. If for any two constraints A and B in the set, $A \wedge B$ is false, then each worker will potentially explore different parts of the symbolic execution tree, and we state the set of constraints is *disjoint*. If the disjunction of all the constraints in the set simplifies to true, then every possible path will be explored by at least one instance of symbolic execution, and we state our set of constraints is *complete*. Finally, if each constraint in the set will cause symbolic execution to ignore some, but not all of the symbolic execution tree it would usually explore – i.e, each constraint will affect execution – we state the set contains *useful* constraints. Ideally, each partition will require the same amount of time to explore. Creating such partitions is difficult without a priori knowledge of the entire symbolic execution tree; consequently, we heuristically partition the tree.

Our parallelization technique, Simple Static Partitioning (SSP), is described in detail below. The user provides two inputs to the SSP technique: the depth of the shallow execution step, and the minimum number of constraints desired. Following this, SSP then performs a shallow symbolic execution of the system to the depth provided by the user, collecting all PCs observed at said depth. These PCs are then broken into a set of individual constraints (i.e. no conjunctions). Using this set, SSP operates by (1) selecting a symbolic variable commonly used in the set, (2) generating a set of constraints that use the variable (using the set of individual constraints for guidance), (3) removing all the variables used in the newly generated set of constraints from future consideration and (4) repeating until we can (combinatorially) generate the number of constraints desired. These constraints are then placed in queue and distributed to parallel workers.

SSP with inputs depth D and suggested queue size SQS:

1. Perform shallow execution, collecting all PCs at depth D.

2. Split each PC into individual constraints and count the frequency of each constraint. Place every constraint in a set *IndCons* along with its frequency.

3. Count the frequency of each symbolic variable in *IndCons*. Place every variable in a set *AvailableVars* along with its frequency.

4. Using *IndCons*, label symbolic variables as "expensive" or "cheap", where "cheap" variables are used *only* in constraints that are "cheap".

Constraints are "expensive" if they use multiplication and/or division, and "cheap" otherwise.

5. Create an empty set *GeneratedConstraints*, containing sets of constraints generated by SSP. The product of the size of each set of constraints represents the number of constraints we can generate (we term this *numGeneratable*).

6. While *numGeneratable* < *SQS*:

   (a) Choose the cheap variable V with the highest frequency from *AvailableVars*. If no cheap variable is in *AvailableVars*, choose the expensive variable V with the highest frequency.

   (b) If V (1) is only referenced in equality constraints, (2) a symbolic integer and (3) the range of V is no more than twice the number of equalities using V in *IndCons*, then:

   Add {All constraints referencing V} to *GeneratedConstraints*

   else:

   C = Most common constraint referencing V in C
   Add {$C, \neg C$} to *GeneratedConstraints*

   (c) Remove every variable referenced in *GeneratedConstraints* from *AvailableVars*.

7. Generate every combination of constraints using the sets in *GeneratedConstraints* (i.e, Cartesian product of *GeneratedConstraints*). Each combination represents a conjunction of constraints. Order the conjunctions (process described under *Constraint Queue*) to create *ConstraintQueue*.

The key intuitions behind this technique are: (1) more commonly used variables are likely to partition the state space in useful ways, (2) the performance of constraint solvers suffers when multiplication and division are used; avoid those constraints, and (3) if V can only take on a small range of values, each value is likely to be important, so use all possible constraints for V rather than picking only one. Steps 6a and 6b handle intuitions 2 and 3, respectively. Also note that 6c prevents overconstraining a variable (conservatively), i.e., selecting multiple constraints for the same variable such that the constraints cannot all be satisfied.

The resulting set of partitions meets our stated goals. Each conjunction differs from the others, and thus the set of conjunctions is *disjoint*. Each set of constraints in *GeneratedConstraints* is *complete*, and thus the Cartesian product of these constraints is obviously complete. Finally, the constraints are drawn from observed PCs and are thus *useful*, provided every symbolic variable in the constraints is generated (some dynamically generated variables may be referenced but not generated, but this is rare in practice).

User Control The user's control over SSP is limited to the depth of the shallow execution and the queue size minimum (which we term the *suggested queue size*). Both of these can influence the performance of SSP. The larger the depth of shallow execution, the better SSP's knowledge of the underlying symbolic execution tree will be, which may lead to better constraints. For example, branches that terminate at a depth less than the user-specified depth are ignored. Naturally, large depths increase the time required by SSP. Thus selection of depth must be balanced between creating effective partitions and controlling the overhead incurred.

Similarly, the suggested queue size determines the number of constraints in the queue. A larger number of partitions increases the overhead required (e.g., more communication, greater overlap in partitions), but allows more freedom in load balancing. This is discussed in greater detail in Section 6. Note that the suggested queue size is simply a *minimum* size; the partitioning process may (and often does) produce more partitions and thus a larger queue.

Constraint Queue As we will see in Section 4, our parallelization framework uses a queue to distribute JPF configurations. If
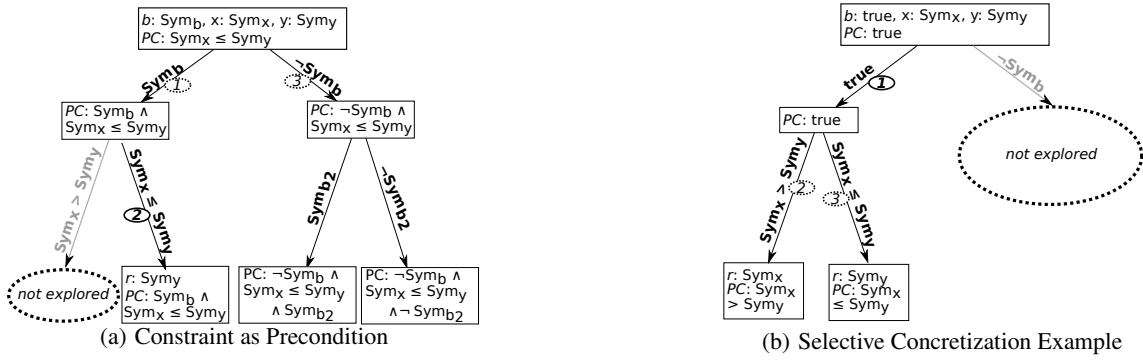
(a) Constraint as Precondition   (b) Selective Concretization Example

Figure 2: Constraining Path Condition (PC) in Simple Static Partitioning

the set of constraints is larger than the number of workers, ordering is relevant, as it determines which constraints are distributed first. SSP orders *ConstraintQueue* such that constraints on the ith symbolic variables chosen in Step 6a change every 2^{i-1} items in the queue. In other words, constraints on the first symbolic variable differ between sequential items in the queue, constraints on the second symbolic variable chosen differ between every 2 items in the queue, etc. This ordering was chosen so that constraints over more common symbolic variables change more frequently in the queue.

Example Let's suppose we run SSP using the example from Figure 1 with a depth of 3 and a suggested queue size of 4. Execution to depth of 3 yields the PCs corresponding to the 4 paths shown in Figure 1. SSP then proceeds producing the following sets:

Step 1: **Collected PCs =** $\{b \wedge x > y, b \wedge x \leq y, \neg b \wedge b_2, \neg b \wedge \neg b_2 \}$

Step 2: **IndCons =** $\{(b, 2), (\neg b, 2), (x > y, 1), (x \leq y, 1), (b_2, 1), (\neg b_2, 1) \}$

Step 3: **AvailableVars=** $\{(b, 4), (x, 2), (y, 2), (b_2, 2) \}$

Step 6b: **GeneratedConstraints (1st iteration) =** $\{\{b, \neg b\}\}$

Step 6c: **AvailableVars (1st iteration) =** $\{ (x, 2), (y, 2), (b_2, 2)\}$

Step 6b: **GeneratedConstraints (2nd iteration) =** $\{\{b, \neg b\}, \{x > y, x \leq y\} \}$

Step 6c: **AvailableVars (2nd iteration) =** $\{ (b_2, 2)\}$

Step 7: **ConstraintQueue =** $\{b \wedge x > y, \neg b \wedge x > y, b \wedge x \leq y, \neg b \wedge x \leq y \}$

As we can see, the symbolic variable b is the most commonly referenced symbolic variable in *AvailableVars*, and is thus selected first. The most common constraints referencing b are b and $\neg b$ (seen in *IndCons*); we arbitraily choose one and add it and its negation to *GeneratedConstraints*. We remove b from *AvailableVars*, leaving x, y and b_2, each with a frequency of 2. We arbitrarily choose x, and select the most common constraint referencing x, $x > y$, and add it and its negation ($x \leq y$) to *GeneratedConstraints*. We then remove both x and y from *AvailableVars*, leaving b_2. We can now generate 4 conjunctions using the sets in *GeneratedConstraints*. The suggested queue size is 4, so we terminate, generate the *ConstraintQueue* and order it according to the rules above.

Distributing and Using Constraints The technique above only generates constraints. Once this is complete, we must distribute these constraints to workers. Distributing constraints is handled using the general purpose parallel JPF framework, described in Section 4. The queue of constraints is transformed into a queue of *worker configurations*, with each worker configuration containing a constraint to be used in the exploration of a subtree.

When a worker configuration is retrieved by a worker, the con-

straint is effectively used as an initial precondition. An example of this is given in Figure 2(a). As shown, an initial precondition of $x \leq y$ is used. As a result, line 9 cannot evaluate to *true* and some of the symbolic execution tree shown in Figure 1 is not explored. Two separate instances of SPF conducted in parallel can be used to divide computation. One instance would use the precondition $x \leq y$, and another instance would use the precondition $x > y$. Taken together, these two instances explore the entire tree shown in Figure 1 while exploring different parts of said tree. (Clearly, this would lead to less than optimal partitioning, as significant overlap exists between the two workers.)

Selective Concretization In symbolic execution, the use of the constraint solver can be quite expensive. Consequently, we have implemented an optimization when using constraints of the form $var == value$, called *selective concretization*. This optimization leverages SPF's ability to handle both concrete and symbolic values, i.e., mixed-mode execution. In this technique, one or more variables normally assigned symbolic values are instead assigned a single *concrete* value. While functionally equivalent to directly using the constraint, replacing a symbolic variable with a single concrete value reduces the size of path constraints, potentially improving the performance of the constraint solver.

An example of when selective concretization occurs is shown in Figure 2(b). In this example, the constraint used is $b == true$, and the variable b is concretized as *true*. As a result, line 7 cannot evaluate to *false*, and only roughly half of the symbolic execution tree explored in Figure 1 is explored. We expect this technique to be effective when working with systems that contain important inputs with a discrete domain, such as boolean values, flags, flight modes etc. These type of inputs are often present in NASA and avionics software.

PCs as Constraints Readers familiar with symbolic execution may note that we can statically partition by simply using the set of PCs collected at a specified depth (plus all PCs for leaves at smaller depths) as the set of constraints. Such a set would be disjoint and complete, and the cost of constructing such a set is very low. However, this method presents problems: namely, unless we wish to create a very large number of partitions (> 1000), we must collect PCs from a very shallow depth, thus partitioning using very limited information.

For example, for the TreeMap case study presented in Section 5 a very shallow execution (maximum depth of 9) requiring less than 1 second and produces 21 PCs, with only 3 symbolic variables (out of 28 potentially produced) referenced. If we increase the depth to 21, symbolic execution requires only 10 seconds, but produces 2,511 PCs, with 10+ symbolic variables referenced. As the cost to collect large numbers of PCs is negligible, but the cost of using each

PC directly as a constraint is potentially high (due to overhead), we desire a method of transforming a large set of PCs into a smaller set of partitions. SSP represents such a method.

4. PARALLEL JPF FRAMEWORK

We present here a general framework for parallelizing JPF. This framework allows us to not only parallelize SPF, but also core-JPF and other JPF extensions. Our framework is built using a simple client-server model, with coordination and communication across parallel instances of JPF handled by an extension of JPF listeners. By using JPF listeners for inter-process communication, our framework has the same flexibility and extensibility present in JPF. This allows for both parallel approaches incorporating significant inter-process communication, or (as with SSP) approaches with very little communication.

Parallel JPF Instances To run parallel instances of JPF, the user first starts the server, known as the *JPF manager*, and provides a *parallel configuration* to the JPF manager. The user then starts one or more clients, directing the clients to connect to the JPF manager. We hereafter refer to these clients as *workers*. The parallel configuration specifies how parallel instances of JPF should be configured and is similar to a standard JPF configuration. The configuration can specify global configuration used by *all* parallel instances, such as the system to explore, and can specify *individual* configuration for a single worker, such as the random seed to use. The parallel configuration is processed by the JPF manager to create a list of individual JPF configurations, known as *worker configurations*. These worker configurations are placed in a queue, and sent to clients upon request. Each client, upon receiving an worker configuration, creates and starts a JPF instance using the configuration. Upon termination of the JPF instance, the client will request a new worker configuration; if the queue is exhausted, the client exits.

Communication between Workers Communication between workers is accomplished by remote listeners and remote listener managers. Figure 3 illustrates communication between objects in our framework for one worker and one remote listener. *Remote listeners* are JPF listeners extended to communicate with a remote listener manager. Remote listeners are specified in the worker configurations, and are registered with the worker's JPF instance. *Remote listener managers* are specified in the parallel configuration given to the JPF manager, and are created and maintained by the JPF manager. Once created, remote listener managers generate and append configuration information for one or more remote listeners for each worker configuration, and communicate with these remote listeners during execution. Thus remote listener managers and remote listeners are associated in a one to many relationship – each remote listener manager communicates with every remote listener instance it configures. Communication between a remote listener and a remote listener manager is bidirectional – the listener can send information to the remote listener manager, and the remote listener manager can send information in response. As with JPF listeners, multiple types of remote listeners and remote listener managers can be used simultaneously.

JPF worker-server communication is implemented using Java Remote Method Invocation (Java RMI) [8]. The JPF manager (i.e. the server) is implemented as a remote object, and workers and remote listeners communicate with the JPF manager via the object's methods. This is a very flexible method of communication – workers can be run simultaneously on the same computer (e.g, multi-core computers), on different computers (e.g. cloud computing) or any combination thereof.
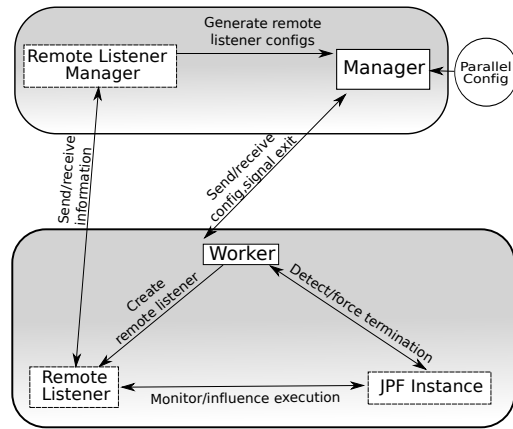

Figure 3: Parallel JPF Support (1 Worker Shown)

Parallel Search Implementing a parallel search technique in our framework can be done in two ways. The simplest method is to implement a configurable JPF listener or heuristic. Running the parallel search technique is then done by running the same listener or heuristic, configured differently, on each worker. For example, to implement parallel RDFS, we implemented a JPF listener configurable with a "random seed" parameter to perform RDFS. To run parallel RDFS, we use a configuration globally specifying that workers should use RDFS and explore the same program, but that *each* worker should use a different seed. We use a remote listener to collect the results produced by each worker.

While simple, this method has the drawback of requiring each worker to be separately configured. A different method would be to implement the technique as a remote listener, using the remote listener manager's ability to generate configuration info. For example, SSP is implemented in this fashion; the remote listener manager runs the partitioning heuristic, and uses the resulting constraints to generate configuration info. Note that parallel search techniques that require communication *during* JPF's execution must be implemented using a remote listener.

Example Remote Listener Consider Figure 4 illustrating a remote listener we use for monitoring MC/DC test generation. We wish to measure the coverage achieved, collect the MC/DC test suites generated by potentially remote workers, and combine and reduce the MC/DC test suites without reducing the coverage achieved. To implement this, each remote listener detects when its corresponding JPF instance generates a test satisfying an obligation, sends coverage information to the remote listener manager, and sends the generated MC/DC test suite when JPF execution terminates. The remote listener manager processes and displays coverage information to the user, and combines and reduces the generated MC/DC test suites. The remote listener manager also generates the configuration information for the remote listeners. The end result of using this remote listener is (1) the total MC/DC coverage achieved is displayed to the user and (2) an MC/DC test suite achieving this coverage is stored on the user's machine.

5. EVALUATION

We evaluate SSP using two metrics: the time to completely explore a finite symbolic execution tree, and the the time to generate tests meeting the MC/DC structural coverage criterion (this includes the time to collect, combine and reduce the MC/DC test suites). Specifically, we investigated two questions:

Reduction in Analysis Time: How does the (1) size of the con-
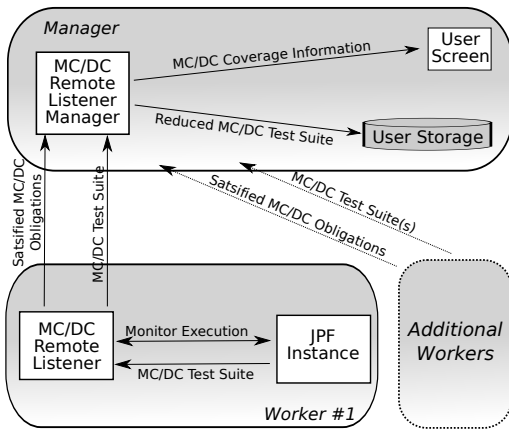
Figure 4: MC/DC Remote Listener

straint queue and (2) the NPW (number of parallel workers) influence the time to completely explore a finite symbolic execution tree?

Reduction in Test Generation Time: How does the NPW influence the time to generate tests satisfying the MC/DC coverage criterion?

The former is a general performance metric: how much faster can we explore a symbolic execution tree when using our parallelization technique, and how well does it scale? The latter measures how well our parallelization technique specifically improves automatic test generation.

5.1 Experimental Setup

To investigate these questions, we vary both the size of the constraint queue and the NPW independently. Based on the length of time required to analyze our case examples, we use constraint queues of suggested sizes 8, 64, and 128. Larger constraint queues were too large, as they would reduce the average time required to analyze each partition to very small values for some case examples (1-5 seconds), potentially making overhead a major factor in the time required. Recall from Section 3 the suggested size is the *minimum* queue size; the actual queue size is larger for most case examples. Actual queue size is indicated in parentheses next the case example name in Tables 2, 4 and 3. We examined NPWs of 2, 4, 6, 8, 16, 32, 64, 92, and 128. We use six case examples in our evaluation. Both the server and the workers were run on a Dual Quad Core Intel Xeon 2.33 GHz with 16 GB of RAM running on SLED 10.1. Our implementation along with the case examples, are freely available from JPF's Sourceforge web-site[1].

Automatic test generation is a problem of reachability in the state space, and there has been work indicating that other reachability problems (specifically detecting concurrency errors) can be improved through parallel RDFS [9]. We therefore use parallel RDFS as a baseline when evaluating question 2, using a large number of runs with 64 different random seeds to prevent bias.

Given the number of NPWs, case examples, queue sizes, and the number of RDFS runs, we wished to run 42,066 test generation runs and 120 analysis runs, using up to 128 cores in parallel. The time required to perform such an experiment is infeasible, and we do not have access to such a large number of cores (even using several multi-core machines). We therefore control costs via simulation and resampling. We describe our evaluation of questions 1 and 2 in Sections 5.3 and Section 5.4, respectively.

---

[1]At time of writing, we are using the SVN repository at https://javapathfinder.svn.sourceforge.net.

5.2 Case Examples

We used six case examples in our experiment: two synchronous reactive systems developed in Simulink and translated to Java, and four Java data structures. Metrics for each case example are shown in Table 1.

Case Example	# Classes	SLOC	# MC/DC Obs.	BAS	SED	# Iterations
ASW	15	425	94	7:57	26	2
WBS	1	231	90	10:22	17	5
BinHeap	2	268	98	9:26	16	6
BinTree	2	115	44	117:34	18	6
FibHeap	2	258	76	47:16	16	7
TreeMap	2	447	172	47:33	16	7

Table 1: Case Examples
BAS = Base Analysis Time (in minutes), SED = Shallow Execution Depth

The first system, the Altitude Switch (ASW), is a synchronous reactive component from the avionics domain. This component turns power on to a Device Of Interest (DOI) when the aircraft descends below a threshold altitude above ground level (AGL). The second system, the Wheel Brake System (WBS), is a a synchronous reactive component from the automotive domain. This determines what pressure to apply to braking based on the environment. Both components were developed in Simulink. The ASW was automatically translated to Java using tools developed at the Vanderbilt University [25]. The WBS was translated to C using tools developed at Rockwell Collins and manually translated to Java.

The remaining four systems are Java data structures. These systems are used in previous related work [2, 27] and provide examples from an additional domain. We perform analysis and test generation for these data structures by exploring all possible sequences of data structure operations up to a finite length. Invalid sequences of data structure operations can generate exceptions, which we catch and ignore.

The number of iterations explored and the length of sequences explored for the data structures are finite and vary between systems. Each length was chosen to allow for reasonably long, but still feasible analysis time when using a single instance of SPF, and to use no more than 1 GB of memory. The depth for the shallow execution was selected for each case example (1) to allow the suggested queue size of 128 to be generated and (2) to allow SSP to complete in less than 10 seconds. As noted in Section 3, larger depths increase the amount of information available to SSP and can favorably improve the quality of the partitions. We intentionally avoid selecting large depths to avoid biasing our SSP results.

For the evaluation in Section 5.4, we used JPF's MC/DC instrumentation Eclipse plugin [23] together with the ComplexCoverage JPF extension to automatically instrument our case examples for MC/DC test generation. The evaluation in Section 5.3 was conducted without instrumentation.

5.3 Evaluation of Analysis Performance

For each case example, to evaluate the analysis performance of SSP we measured the time required to (1) explore the symbolic execution tree using a single instance of SPF, and (2) explore the symbolic execution tree for each combination of NPW and queue size. For a case example C, we term the analysis time required for a single instance of SPF as the *base analysis time* B for C, and term the analysis time required for a specific NPW x and specific queue size y as the P_x^y *analysis time* for C. We define the *startup time* S as the time required to run SSP and construct the queue. For a case example C, we define the *speedup* for NPW x and queue size y as $\frac{B}{P_x^y}$. Ideally, the speedup will be close to the NPW, as this represents linear speedup. We define the % *maximum speedup* as

$\frac{100B}{xP_x^y}$. Ideally, this will be 100%. Finally, we define the *overhead* as $100\frac{\Sigma Q - B + S}{B}$, where Q is the set of worker runtimes, and thus ΣQ is cumulative time across all workers. Ideally, this will be 0%.

Measuring the analysis time required using a single instance of SPF is simple – run SPF over the case example and measure the total runtime. As noted above, however, we have limited resources and cannot run each combination of NPW and queue size. Instead, we ran each suggested queue size *once* and simulated the results for each NPW. For each suggested queue size, we ran SSP using a single worker and measured the following for each item in the queue: the time required to retrieve the configuration (i.e., communication overhead), the time required to instantiate SPF, and the time to run SPF. Together, these times represent the total time required to run the item. The time needed to instantiate and run SPF was measured using Java's *ThreadMXBean* class, while the communication overhead was measured using wallclock time (to ensure time spent blocking is counted). We also measured the time required by SSP and the JPF manager's startup time (i.e., time until the constraint queue is built).

For each queue size, this resulted in a list of runtimes for each item in the queue and the overhead incurred by the queue-building process. Using these times, we simulated the total time for each NPW of interest using the following process. First, we create a runtime for each simulated parallel worker, with each runtime initialized to S, i.e. the time spent building the constraint queue. We then (1) remove the first element of queue, (2) add the entire time required by said element to the runtime for the simulated parallel worker with lowest current runtime (i.e., the next idle worker) and (3) repeat until the queue is empty. Finally, we select the maximum runtime from the list of simulated runtimes: this is the longest time required by any simulated worker, and thus represents the time required to explore the symbolic execution tree in parallel. Data collected is then used to compute the speedup, the % maximum speedup, etc.

This simulation ignores competition between simulated workers; we address this in Section 5.5. We list the speedup results for analysis in Table 2, and metrics over individual queue items in Table 3. Note we have omitted BinTree using a suggested queue size of 128, as results for smaller queue sizes indicated the time to run would be prohibitive.

5.4 Evaluation of Test Generation Performance

For each case example, to evaluate the test generation performance, we generated tests automatically using both parallel RDFS and the partitioning technique. As with our evaluation of analysis time, we are exploring a finite symbolic execution tree, and RDFS and SSP both completely explore the tree. Both techniques thus achieved the same MC/DC coverage, but differ in the time required to achieve this coverage. Our evaluation of test generation performance is based on the difference in time required to reach this coverage. We used the MC/DC remote listener described in Section 4 to collect the generated test suites.

The performance in parallel RDFS can vary depending on the random seeds used. As in Section 5.3, we do not have the resources to run large numbers of test generation runs for each NPW and case example. Instead, for each case example, we ran 64 RDFS runs using a single worker, each run with a different seed. This produced 64 test suites, one for each run. Each test suite achieved the same coverage, but the time required satisfy individual obligations varies between test suites.

We then sampled from these test suites to simulate running parallel RDFS for each NPW. When generating a test, the Complex-Coverage extension records the CPU time required to generate the

test and the obligation the test satisfies. Thus for each NPW x, we can randomly sample x test suites and determine for each coverage obligation o the earliest time o is satisfied by any test suite. This information is used to determine the earliest time all obligations o are satisfied by any of the sampled test suites. We term this the *time to finish (TTF)* for the run. We performed resampling for each NPW 1,000 times (or the maximum number of possible times) and average the resulting TTFs.

We simulated test generation for the partitioning technique using the same basic approach from the previous subsection. For each item in the queue, this resulted in the same timing information used in the previous subsection, as well as a test suite. We then simulated each NPW using the process outlined in the previous section, with one exception: when an item is removed from the queue and added to a simulated worker, we modified the test suite timing information for the item. Specifically, we increased the time required to generate each test, adding the (1) stored runtime for the simulated worker (i.e., the total runtime for earlier queue items run on the simulated worker) and (2) the time required to retrieve the configuration and instantiate JPF. Once this process is completed, we determine the TTF for the NPW simulated.

We list the test generation results in Table 4. In our analysis, we explored only a suggested queue size of 64. This size was selected based on the results from Section 5.3, as we felt it represented a medium between too small a queue (leading to poor load balancing) and too large a queue (leading to high overhead).

5.5 Threats to Validity

We performed our experiments primarily as a pilot study to evaluate SSP. We do not perform any statistical analysis, and our results should therefore not be viewed as a rigorous empirical study. Nevertheless, there are several points to consider when interpreting our results. We used six case examples in our evaluation. We selected these examples because they are drawn from two domains and because four of the examples (the data structures) are often used in studies related to JPF. The shape of the symbolic execution tree for these examples undoubtedly influences the effectiveness of SSP, and may not be representative. Additionally, test generation using these examples may not be representative of Java programs in general.

We selected bounds for our systems to yield reasonably long, but feasible analysis times. Additionally, we have selected shallow execution bounds (for partitioning) to yield low overhead for constraint generation. It is possible longer bounds may produce different results (shorter bounds would prevent us from generating the largest queues and were thus not possible). However, we feel these bounds are fair and believe larger bounds would only improve our results.

To control costs, we simulate running parallel workers with parallel RDFS using 64 random seeds and 1,000 samples. It is possible that the pool of 64 test generation runs do not accurately represent the possible runs RDFS. It is also possible that 1,000 simulated runs does not accurately represent the possible set of parallel runs.

Additionally, we simulate running each NPW using one run for each queue size, and ignore the possibility of blocking. It is possible that during parallel execution, multiple workers may request a configuration at the same time, which causes some workers to wait, thus lengthening the time required for analysis. However, the actual time required to retrieve a configuration is very small, less than 5 ms on average. Furthermore, the total number of requests is small (equal to the queue size), and the time required to run each configuration is at a *minimum* a few seconds for all case examples and queue sizes. Given this, the rate of requests will be on aver-

age low with respect to the time required to respond to the requests for the NPW explored. We therefore do not believe this is a major concern in our results.

Finally, we do not employ randomization when generating tests using our partitioning technique. It is possible SPF's default search order is particularly good at finding tests when used with our partitioning technique. However, it seems unlikely to be the case for *all* the systems where improvement was observed.

SQS 8	P_2	P_4	P_6	P_8	P_{16}			
ASW(8)	1.88	3.71	3.79	7.17				
WBS(25)	1.83	3.4	4.75	5.92	11.7			
BinHeap(25)	1.94	1.94	1.94	1.94	1.94			
BinTree(18)	0.95	1.89	2.47	3.0	3.0			
FibHeap(8)	1.89	2.81	3.71	3.71				
TreeMap(25)	1.49	1.69	1.82	1.83	1.84			
SQS 64	P_2	P_4	P_6	P_8	P_{16}	P_{32}	P_{64}	
ASW(64)	1.93	3.82	5.67	7.46	14.45	27.01	47.52	
WBS(100)	1.87	3.73	5.49	7.23	13.47	23.76	44.82	
BinHeap(125)	1.76	2.06	2.17	2.22	2.27	2.28	2.29	
BinTree(162)	0.83	1.56	2.42	3.02	4.57	5.45	5.58	
FibHeap(64)	1.96	3.94	5.72	7.54	10.68	13.55	14.81	
TreeMap(125)	1.93	2.75	3.13	3.33	3.48	3.55	3.55	
SQS 128	P_2	P_4	P_6	P_8	P_{16}	P_{32}	P_{64}	P_{128}
ASW(128)	0.95	1.91	2.78	3.79	7.38	13.75	24.39	42.9
WBS(200)	1.9	3.77	5.67	7.42	14.77	27.56	48.57	90.7
BinHeap(625)	1.8	2.25	2.42	2.5	2.66	2.72	2.74	2.74
FibHeap(512)	1.67	3.33	4.99	6.64	13.06	23.57	32.84	41.15
TreeMap(625)	1.61	3.14	4.42	5.18	6.17	6.75	7.07	7.14

Table 2: Speedup for Suggested Queue Sizes 8, 64 and 128
P_x = Initial Precondition with NPW = x, SQS Y = Suggested Queue Size

	Avg Time	Max Time	Std Dev	Overhead (%)
ASW (8)	63.1	66.3	1.3	6.0%
ASW (64)	7.7	10.0	0.3	3.0%
ASW (128)	7.8	11.1	0.5	108.9%
WBS (25)	23.1	28.5	0.6	5.1%
WBS (100)	6.6	8.9	0.5	6.3%
WBS (200)	3.3	5.6	0.2	4.8%
BinHeap (25)	21.7	290.4	67.6	0.1%
BinHeap (125)	4.5	247	27.7	5.8%
BinHeap (625)	0.9	205.8	11.0	40.7%
BinTree (18)	809.3	2349.7	880.5	106.55%
BinTree (162)	104.3	1262.74	264.0	140.51%
FibHeap (8)	373.5	763.8	225.2	5.5%
FibHeap (64)	43.9	191.4	43.7	0.2%
FibHeap (512)	6.6	60.7	8.6	19.8%
TreeMap (25)	119.7	1547.2	303.9	4.9%
TreeMap (125)	23.0	801.7	84.1	2.34%
TreeMap (612)	5.6	397.9	25.1	23.9%

Table 3: Runtime Metrics for Queue Items (in seconds)

6. DISCUSSION

In this section, we discuss (1) how various factors, including NPW and constraint queue size, influence the time required by SSP to completely explore a finite symbolic execution tree and (2) how the NPW influences the time required to generate tests satisfying the MC/DC coverage criterion.

6.1 Effectiveness of Parallelization

Our results indicate that speedup can be realized for all systems. In particular, for NPW in the range of modern multi-core machines (2, 4, and 8), the % maximum speedup is relatively high for most systems, with many observed values higher than 90%. This is illustrated in Figure 5, showing the maximum observed speedup for each case example. As we can see, several case examples are close to the maximum speedup, thus achieving near-linear speedup.

However, our results do demonstrate variability between case examples. For two systems (WBS, FibHeap), significant speedup was observed for all queue sizes and NPW, with speedup nearly always increasing as NPW and queue sizes increased. However, for the other systems speedup plateaued at certain NPWs (e.g., BinHeap). Furthermore, a larger queue size sometimes led to *worse* performance than a smaller queue size for some NPWs, while simultaneously *improving* the performance for other NPWs (e.g. FibHeap, TreeMap). This leads to two key observations: (1) speedup with respect to NPW is *monotonic*, and (2) speedup with respect to queue size is *not* monotonic, and thus care must be taken when suggesting a queue size.

The first observation is perhaps unsurprising. SSP requires little communication by design, and thus while increasing the NPW adds a small amount of communication overhead, the gains from having another worker nearly always outweigh it. Indeed, this is one of the primary benefits of using only static partitioning – additional workers can be added without worries of decreased performance.

The second observation is more interesting. Load balancing in SSP is determined largely by the presence, quantity, and position in the queue of large partitions. Given a queue with a small variance in the time required to explore each partition (relative to the overall time required), we know that few, if any partitions, will require a significant percentage of the overall resources – such large partitions likely do not exist. Of course, finding such a set of partitions is difficult, and SSP's ability to do so varies depending on the inputs. As shown in Table 3, the data structures tend to have a few partitions that require substantially more time to explore than the average partition. These partitions occur because certain operations (namely adding values to the data structure) are expensive, and make subsequent operations (e.g., find) more expensive as well. However, if large partitions exist, we can still achieve good load balancing by placing such partitions earlier in the queue. Larger partitions are explored early, and smaller partitions essentially "fill the gaps" as larger partitions complete.

Consequently, improvements to load balancing when using static partitioning must (1) reduce the variation in partition size or (2) improve the queue ordering. One method of reducing variance in the constraint queue is to simply increase the size of the queue generated. Ideally, each partition in the original queue will be divided into two or more smaller partitions, and the cumulative time to explore these smaller partitions will be roughly equal to that of the original partition. Thus the average time to explore a partition will decrease in the larger queue, while the cumulative time for both queues will be roughly the same, reducing the variance. Indeed, as shown in Table 3, the standard deviation (along with other runtime metrics) decreases as the queue size increases. However, by increasing the size of the queue, we increase the overhead. This increase in overhead reduces or negates the performance gains for most systems when using large queues. For example, the overhead for the FibHeap system jumps from 0.2% to 19.8% when the queue size is increased from 64 to 512, leading to reduced performance for NPW of 2-8, but *increased* performance due to better load balancing for NPW of 16-64. This suggests that queue size should be selected with respect to the NPW, perhaps using the ratio of queue size versus NPW as a guide.

Furthermore, by increasing the queue size, we increase the likelihood that any "expensive" constraints will be selected, which can cause the constraint solver to significantly slowdown. This factor is present in the ASW case example – when using a queue size of 128, constraints containing multiplication and division are used. While these constraints maintain the low variability, the overhead jumps to over 100%.

	R_1	R_2	R_4	R_6	R_8	R_{16}	R_{32}	R_{64}	P_2	P_4	P_6	P_8	P_{16}	P_{32}	P_{64}
ASW (64)	2686	2656	2632	2621	2613	2598	2581	2557	175	84	44	41	40	40	38
WBS (100)	28	13	7.3	4.7	3.5	1.5	0.8	0.5	10	10	10	0.5	0.5	0.5	0.5
BinHeap (125)	2075	1250	620	366	273	182	168	158	35	32	32	32	28	23	23
BinTree (64)	14	3.8	1.1	0.7	0.6	0.4	0.4	0.3	30	0.21	0.17	0.16	0.15	0.14	0.13
FibHeap (64)	948	901	873	862	851	838	828	819	503	220	157	108	60	26	26
TreeMap (125)	632	605	584	574	568	554	543	531	481	266	194	158	133	121	118

Table 4: Test Generation TTF (in seconds) for Suggested Queue Size 64
R_x = Randomized Depth First Search with NPW = x, P_x = Initial Precondition with NPW = x
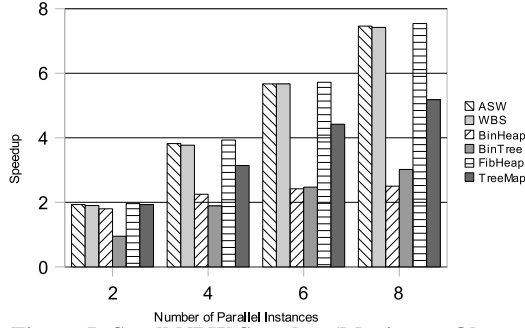
Figure 5: Small NPW Speedup (Maximum Observed)

A final note: the BinTree system is the only system in which a speedup less than 1.0 was observed, indicating that exploring Bin-Tree's symbolic execution tree is faster using *one* instance, rather than two; all larger NPWs produce speedup. The reason is that when the inputs are generated dynamically, as is the case for the method parameters, some of the collected constraints are useful only for certain executions (e.g., executions where that method is called); they are useless for other executions. These constraints are usually not selected in favor of more generally applicable and thus more numerous constraints; however, if selected, they may lead to useless constraints over symbolic variables that are not generated. In the future, we plan to address this problem by considering the relationship between constraints and generation of symbolic variables. This phenomenon was not observed in the other examples.

6.2 Influence of NPW on Test Generation

Our results indicate that SSP significantly improves test generation performance for each case example except BinTree. In particular, we observe that for the case examples where a single instance of SPF requires at least 10 minutes to achieve the maximum achievable coverage, the performance improvements with respect to NPW are generally more scalable when using SSP than when using parallel RDFS, and for a given NPW, SSP outperforms parallel RDFS.

Both observations indicate that SSP is a more effective method of improving test generation performance than parallel RDFS. Indeed, the performance improvements observed in test generation are larger than the analysis time speedup – for example, for the FibHeap system and a NPW of 64, using SSP reaches the achievable maximum coverage 31x faster than 64 parallel RDFS runs, despite a corresponding analysis time speedup of only 15x. Furthermore, for the ASW system and a NPW of 64, using SSP reaches the achievable maximum coverage 70x faster than the average RDFS run – a *greater* than linear speedup.

This results from the use of depth first search. When exploring a subtree in the symbolic execution tree, depth first search explores the entire partition. If the subtree is large but contains no unsatisfied obligations, depth first search may spend significant time exploring the partition without improving coverage. We have observed that obligations are often distributed throughout the symbolic execution tree. In this situation, a single instance of RDFS will inevitably ignore obligations on the *left* side of the symbolic execution tree

while thoroughly exploring the *right* side (or vice versa). SSP, which partitions execution, is less prone to this.

Note that unlike most case examples used, BinTree performs well using parallel RDFS, finding the maximum achievable coverage in at most 14 seconds – a difficult feat to top. This, in conjunction with the SSP problem discussed above, is responsible for the poor performance of SSP relative to parallel RDFS.

7. RELATED WORK

We have already discussed some closely related work in sections 1. We add more discussion here.

Research on automatically generating tests to satisfy structural coverage criteria has been conducted for many years [19]. Recent approaches for Java programs include Korat by Boyapati et al. [5], an explicit state enumeration technique for generating test inputs for a Java method. Korat has been parallelized by Siddiqui and Khurshid as PKorat [22] using a master/slave configuration to explore candidate vectors in parallel. Korat was also parallelized by Misailovic et al. [10]. That work describes parallel test generation and execution and it is similar in spirit with our approach, as it partitions the search space, but in the context of explicit state space exploration. Also relevant is jCute by Sen et al. [21], a concolic execution engine for Java; jCute has not been parallelized however we are aware of a parallel version of a concolic execution engine in Microsoft's SAGE system.

Our approach to automatic test generation is similar to counterexample based test generation, first developed by Ammann et al. [1] using mutation operators and specifications. The approach has been modified to generate tests satisfying structural coverage metrics by Rayadurgam and Heimdahl [20] and to generate tests via JPF-based symbolic execution by Khurshid et al. [14].

Stern and Dill present in [24] a parallel version of the explicit state model checker Murϕ suitable for verifying safety properties. Holzmann and Bosnacki present in [12] a multi-core version of the explicit state model checker SPIN suitable for verifying liveness properties, as do Barnat et al. [3]. Dwyer et al. in [9] and Holzmann et al. in [13] explore the use of non-communicating parallel randomized search for fault detection. These approaches were shown to be effective and are very similar to the RDFS technique presented in this work. However, these approaches are examples of explicit state model checking, rather than symbolic execution.

Finally, in his master thesis [15], King parallelized symbolic execution using a dynamic load balancing approach. The approach dynamically populates a queue of subtrees, adding subtrees when nondeterministic choices are encountered. Idle workers poll for work and active workers contribute to this queue. The effectiveness of his approach was measured in terms of speedup in analysis time (similar to Section 5.3) and the number of state paths explored in a specific amount of time, with NPI of 2, 4, 6 and 8. As case studies, the same data structures were used (though not the exact same code), but he analyzed individual methods in isolation while we analyzed method sequences. Speedups reported tend to be close to linear for NPW of 2, often reasonable for NPW of 4, but relatively poor for NPW of 6 and 8. For several case examples, the speedups

for NPW of 2-4 exceed the speedup for NPW of 8 (presumably due to overhead). Furthermore, for at least one case example, a single instance of symbolic execution outperforms a run with NPW of 8. In contrast, SSP does not exhibit a drop in speedup with increased NPW. Furthermore, speedup with SSP seems more consistent with NPW of 2-8 than King's approach.

8. CONCLUSIONS AND FUTURE WORK

We have presented a general framework for parallelizing Java Pathfinder and techniques for parallel symbolic execution developed using this framework. We have evaluated these techniques in terms of analysis time and MC/DC test generation time using six case examples. We demonstrate up to 90x speedup in analysis time using 128 workers, and 70x speedup in automatic test generation time using 64 workers. Furthermore, for small numbers of workers (2-8) we demonstrate speedups in analysis time consistently larger than 90% of the maximum (linear) speedup for 3 of 6 systems, with the other three systems demonstrating speedups of 30% to 90% of the maximum speedup.

In the future, we would like to evaluate the partitioning techniques in other contexts, such as fault detection. Given the similarity between searching for states satisfying coverage obligations and searching for states violating assertions/properties, we believe the partitioning techniques will perform well in this context. Furthermore we plan to investigate other static partitioning techniques (e.g. based on the control flow graph) and study their effectiveness in conjunction with dynamic partitioning, e.g. [7, 15]. We believe that such a combination will be most effective for parallelizing symbolic execution.

9. REFERENCES

[1] P. E. Ammann, P. E. Black, and W. Majurski. Using model checking to generate tests from specifications. In *Proc. of 2nd IEEE Int'l. Conference on Formal Engineering Methods*, pages 46–54. IEEE Computer Society, Nov. 1998.

[2] S. Anand, C. Pasareanu, and W. Visser. JPF-SE: A symbolic execution extension to Java PathFinder. In *Proc. of 13th TACAS Conference*, volume 4424, page 134. Springer, 2007.

[3] J. Barnat, L. Brim, and J. Chaloupka. Parallel breadth-first search LTL model-checking. In *Proc. of 18th IEEE Int'l. Conference on Automated Software Engineering*, pages 106–115, 2003.

[4] B. Bezier. *Software Testing Techniques, 2nd Edition*. Van Nostrand Reinhold, New York, 1990.

[5] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated Testing Based on Java Predicates. In *Proc. of the 2002 ACM SIGSOFT Int'l. Symposium on Software Testing and Analysis*, pages 123–133. ACM New York, NY, USA, 2002.

[6] J. Chilenski and S. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9:193–200, September 1994.

[7] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea. Cloud9: A Software Testing Service.

[8] T. Downing. *Java RMI: remote method invocation*. IDG Books Worldwide, Inc. Foster City, CA, USA, 1998.

[9] M. Dwyer, S. Elbaum, S. Person, and R. Purandare. Parallel randomized state-space search. In *Proc of 29th Int'l. Conference on Software Engineering*, pages 3–12, 2007.

[10] B. Elkarablieh, D. Marinov, and S. Khurshid. Efficient solving of structural constraints. In *Proc. of the 2008 International Symposium on Software Testing and Analysis*, pages 39–50. ACM New York, NY, USA, 2008.

[11] P. Godefroid. Compositional dynamic test generation. In *Proc. of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 47–54. ACM New York, NY, USA, 2007.

[12] G. Holzmann and D. Bosnacki. The design of a multicore extension of the spin model checker. *IEEE Transactions on Software Engineering*, 33(10):659–674, 2007.

[13] G. Holzmann, R. Joshi, and A. Croce. Tackling large verification problems with the swarm tool. *Proc. 15th Int'l. SPIN Workshop*, 5156:134–143, 2008.

[14] S. Khurshid, C. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. *Proc. of the 9th TACAS*, pages 553–568, 2003.

[15] A. King. Distributed parallel symbolic execution. Master's thesis, Kansas State University, 2009.

[16] J. King. Symbolic execution and program testing. *"Communications of the ACM"*, 1976.

[17] C. Păsăreanu, P. Mehlitz, D. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In *Proc. of the 2008 Int'l. Symposium on Software Testing and Analysis*, pages 15–26. ACM New York, NY, USA, 2008.

[18] C. Păsăreanu, J. Schumann, P. Mehlitz, G. Karsai, H. Nine, and S. Nima. Test-case generation for Simulink / Stateflow models of software-intensive mission-critical systems. In *3rd IEEE Int'l. Conference on Space Mission Challenges for Information Technology*, 2009.

[19] C. Ramamoorthy, S. Ho, and W. Chen. On the automated generation of program test data. *IEEE Transactions on Software Engineering*, pages 293–300, 1976.

[20] S. Rayadurgam and M. P. Heimdahl. Coverage based test-case generation using model checkers. In *Proc. of the 8th IEEE Int'l. Conference and Workshop on the Engineering of Computer Based Systems*, pages 83–91. IEEE Computer Society, April 2001.

[21] K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. *Computer Aided Verification*, 4144:419, 2006.

[22] J. Siddiqui and S. Khurshid. Pkorat: Parallel generation of structurally complex test inputs. In *Proc. of the 2009 International Conference on Software Testing Verification and Validation*, pages 250–259. IEEE Computer Society Washington, DC, USA, 2009.

[23] M. Staats. Towards a framework for generating tests to satisfy complex code coverage in Java Pathfinder. In *Proc. of NASA Formal Methods Symposium 2009*, 2009.

[24] U. Stern and D. Dill. Parallelizing the Murϕ verifier. *Formal Methods in System Design*, 18(2):117–129, 2001.

[25] J. Sztipanovits and G. Karsai. Generative programming for embedded systems. In *Proc. of Int'l. Conference on Principles and Practice of Declarative Programming*, volume 6, pages 180–180. Springer, 2002.

[26] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. In *Proc. of Automated Software Engineering*, pages 203–232. Springer, 2003.

[27] W. Visser, C. Păsăreanu, and R. Pelánek. Test input generation for java containers using state matching. In *Proc. of the 2006 Int'l. Symposium on Software Testing and Analysis*, pages 37–48. ACM New York, NY, USA, 2006.