DISTRIBUTED PARALLEL SYMBOLIC EXECUTION

by

ANDREW KING

B.S., Kansas State University, 2005

A THESIS

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Computer Science
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas
2009

Approved by:

Major Professor
Robby

Copyright

Abstract

Software defects cost our economy a significant amount of money. Techniques that can detect software defects before the software begins its operational lifecycle are therefore highly valuable. Unfortunately, as software is becoming more ubiquitous, it is also becoming more complex. Static analysis of software can be computationally intensive, and as software becomes more complex the computational demands of any analysis applied increase also. While increasingly complex software entails more computationally demanding analysis, the computational capabilities provided by computers have increased exponentially over the last half century of computing. Historically, the increase in computational capability has come by increasing the clockspeed of the computer's central processing unit (CPU.) In the last several years, engineering limitations have made it increasingly difficult to build CPU's with progressively higher clock speeds. Instead, processor manufacturers now provide increased capability in the form of 'multi-core' CPUs; where each processor package contains two or more processing units, enabling that processor to execute more than one task concurrently. This thesis describes the design and implementation of a parallel version of symbolic execution which can take advantage of modern multi-core and multi-processor systems to complete analysis of software units in a reduced amount of time.

Table of Contents

List of Figures

List of Algorithms

Acknowledgments

This thesis would not have been possible without all my teachers, family, and friends that have supported me during the development of the many versions of distributed Kiasan and the writing of this thesis. First of all I want to thank my advisor Robby for putting up with all the times I wandered into his office to ask him questions about how different modules in Bogor or Kiasan actually worked, not to mention the multiple hour debugging sessions where he helped me quash particularly stubborn programming bugs. Next I want to thank my family for understanding when I would spend more time on family vacations working on my project than interacting with the rest of the family. Finally, I'm glad my friends put up with me when I was much more focused on this project than spending time with them.

Chapter 1

Introduction

Computing systems are pervasive in modern society. Computers are intimately involved with everything from the micro-controller in a high-end electric shaver to the autopilot of a commercial jet airliner. Computers can be used for entertainment (video games, streaming video). Computers are also used for standard day-to-day business dealings (digital database systems, word processors, video teleconferencing).

What every computer needs in order to operate is software, which is the logical description of some process or calculation. It is no surprise then, that the production of software is big business. The 2007 total revenue for the Software 500, a list of the largest 500 software companies, exceeded 394 billion U.S. dollars, and continues to grow each year[4]. Software defects have been calculated to cost the United States' economy $59.5 billion.[5] As the demand for software grows, so has the demand for techniques to reduce software defects. A software defect can be loosely defined as an aspect of a software system that causes it to behave in a way contrary to its specification.

Amongst software professionals (developers, engineers, academic researchers) there has always been an intense interest in figuring out how to detect and correct (or mitigate) defects in software systems before those systems go live: defects that have gone undetected end up costing money and in some cases even human lives. The many techniques developed are quite varied both in their scope and in their depth. Each approach has different strengths and weaknesses. The techniques, just to name a few, include: weakest-precondition analysis,

abstract interpretation, model-checking, information-flow analysis, software testing, etc.

A popular approach in the industrial setting has been software-testing. Software testing is simply where software engineers develop a corpus of test cases (a set of inputs to the software). Developers run the software against each test case. If the software produces unwanted output then a defect has been detected, and an engineer will attempt to correct the defect. A recent incarnation of software testing is known as unit testing. Unit tests conceptualize a software system as interconnected discrete units. Software engineers give each unit some functional specification (formally or informally). Unit tests are designed to exercise the functionality of each unit. As with software testing unit testing requires the investment of developer time; a human must come up with the test cases. For a complex unit, a large number of test cases may be required to exercise enough of the unit to detect any defects.

Coinciding with the increased demand for software, computational capabilities in hardware are also growing. While for much of the time since the 1950's computing power has increased primarily through higher clockspeed central processing units, recently semiconductor companies have had difficulty increasing the clockspeed at the same rate. Instead of increased clockspeed, processor manufacturers are leveraging better manufacturing techniques to put more than one processing unit in a single processing package. As of the writing of this thesis, mid-range server class systems commonly have up to 24 processors spread across 4 CPU packages (4 processor Dunnington series Xeon machine.) Major CPU manufacturers have CPU packages on their road maps that will contain 100 or *more* processors sometime in the next 10 years[6].

This means that while computing systems will continue to grow in terms of absolute power, software developers will need to write programs that take advantage of parallelism in order to harness that new power. Not only will that make software more complex in general, it also means that some algorithms probably won't be able to take direct advantage of more processing cores if those algorithms are inherently serial.

One research question is can the computing power of highly parallel systems be exploited for software verification? This thesis explores an analysis technique, symbolic execution, that is particularly amenable to parallelization.

Symbolic execution is a type of static analysis that exhibits a certain form of parallelism. As a symbolic execution analyzes a program; it will explore different possibilities of program execution. These explorations are independent once they bifurcate from one another. In theory, this means that individual explorations can be executed by different processors in parallel once the explorations become independent.

1.1 Contributions

The contributions of this thesis are as follows:

1. A high-level description of how symbolic execution can parallelized.
2. A discussion of how parallel symbolic execution was implemented in Kiasan, a symbolic execution tool for Java.
3. Experimental data and evaluation that captures the range of performance exhibited by parallel Kiasan.

1.2 Thesis Structure

This thesis is organized into 8 chapters. Chapter 1 is this introduction. Chapter 2 contains a survey of techniques that are either commonly used, have been implemented as a parallel algorithm, or are directly relevant to the proceeding sections. Chapter 3 describes *parallel symbolic execution* and some observations on how it will perform in certain instances. Chapter 4 contains details on how parallel symbolic execution was implemented in Kiasan. Chapter 5 is dedicated to experiments: it contains both the experiment descriptions and the experimental data. Chapter 6 reviews related work; it describes tools that implement the

techniques from Chapter 2. Chapter 7 contains the conclusion. Finally, Chapter 8 describes some possible future directions for parallel symbolic execution.

Chapter 2

Background

This chapter contains an overview of some selected static analysis techniques. Of most direct relevence to the following section are the descriptions of the various forms of symbolic execution. Other techniques are described for purposes of comparison, and because they relate to tools described in later chapters. This chapter also summarizes some basic approaches to distributed or parallel processing in addition to some basic terms.

2.1 Program Analysis Background

The first half of this chapter describes various program analysis techniques that are relevent to symbolic execution, including various terms commonly used when discussing these techniques.

2.2 Soundness and Completeness

In this document the terms *soundess* and *completeness* are used to describe certain properties of the various analysis. This usage in formal methods is directly related to these term's usage in the domain of mathematical logic.

2.2.1 Soundness

An analysis is sound if it demonstrates that a program has no bad (contrary to expectation or specification) behavior when it specified or implied preconditions are true.

2.2.2 Completeness

If some program has an error, a complete analysis will find it.

2.3 Kripke Structures

A Kripke structure is a 4-tuple $M = (S, I, R, L)$[7] where

1. S is a finite set of states.
2. $I \in S$ is the set of initial states.
3. R is a transition relation, such that $R \subseteq S \times S$ and $\forall s \in S$ and $\forall s' \in S \; \exists (s, s') \in R$
4. L is a labeling function of type: $S \rightarrow 2^{AP}$

Kripke structures can be used to formally specify a finite state system. Kripke structures support the specification of non-determinism by allowing multiple transition relations from the same state s, (e.g. $(s, s') \in R$ and $(s, s'') \in R$ but $s' \neq s''$).

2.4 Explicit State Model Checking

Model checking is a computationally expensive formal analysis technique used to verify that some system is correct with respect to some specification. A model checker takes as input some model of a finite-state system M, and a specification S for the correctness of the system. The model checker is then used to exhaustively explore all states of the system. If the model-checker encounters a state deemed 'bad' by the specification, the model-checker can produce a counter-example, or trace of states that illustrates how the system could evolve to violate the specification[7].

More formally, the model checking problem can be described as: Given some Kripke structure $M = (S, I, R, L)$ which represents a finite state system, and some specification formula F, find all sets in S that satisfy f: $\{s \in S \mid M, s \models f\}$[7].

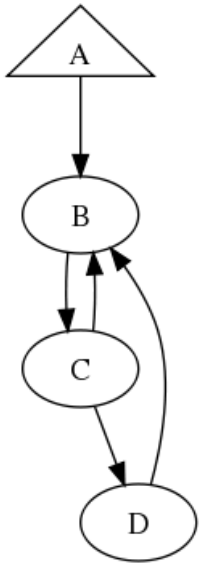


Figure 2.1: *A simple FSM that could represent some Kripke structure, where A is the initial state, and the transition relations are defined by the arcs between the states (nodes)*

Generally, the logic formula f is compiled into another state machine[7]. Techniques exist to do this conversion from formulas in Linear Temporal Logic (LTL), CTL or others but describing those techniques are beyond the scope of this thesis. Instead, for the following example, imagine that the specification has already been compiled into the finite state machine of figure 2.2.

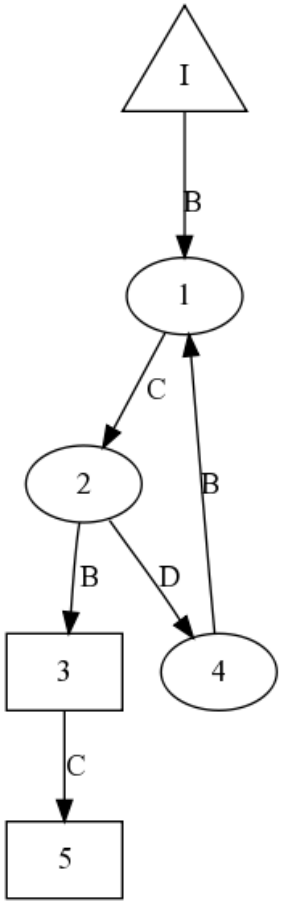


Figure 2.2: *A FSM representation of a specification. I is the initial state, states 3 and 4 are 'bad' states, and the labels of the arcs are from the state labels of figure 2.1*

Figure 2.2 represents the specification that our system should not have a state transition trace that contains $< C, B, C >$ or $< B, C, B >$. The following is the depth first model check of the system M (Figure 2.1) with specification f, represented by FSM S (Figure 2.2):

1. Both FSM are in their initial states.
2. M transitions $A \rightarrow B$.
3. The model checker pushes A onto its backtracking stack. The stack contents are $< A >$
4. S has a transition from I labeled B, so S transitions $I \rightarrow 1$.

5. M transitions $B \rightarrow C$. B is pushed onto the stack, and S transitions $1 \rightarrow 2$, following the arc labeled C. The stack contents are $< A, B >$

6. M has two choices from C, either to B or to D. Assume M transitions $C \rightarrow B$. I transitions $2 \rightarrow 3$. C is pushed onto the stack yielding: $< A, B, C >$.

7. M transitions $B \rightarrow C$, but there is no corresponding label that would allow S to transition from 3, which is a bad state. The model checker detects that S is stuck. The stack is used to construct the counter-example $ABCB$.

8. The model checker will then backtrack M to state C (where there was another unexplored choice) and then continue the search by exploring the transition $C \rightarrow D$.

Algorithm 1 DFS model checking algorithm

```
procedure DFS(s)
markAsSeen(s)
l ← getSpecLabel(s)
specState ← transitionSpecFSM(l)
push(traceStack, s)
if badState(specState) then
  addToCounterExamples(traceStack)
  return
end if
successors ← successorStates(s)
for all succ ∈ successors do
  if notSeen(succ) then
    DFS(succ)
  end if
end for
return
```

2.5 Symbolic Execution

Symbolic execution is a static program analysis approach initially described in *Symbolic Execution and Program Testing*[8]. Symbolic execution reasons about a program symbolically

instead of concretely. For instance, given a small program consisting of one statement S: $x = y + 3$ where $(x \wedge y \in \mathbb{Z}\ )$ In the concrete case, if $y == 0$ prior to the execution of S then it is the case that $x == 3$ after the execution of S.

Since the statement S under consideration is ideal (both x and y are real life integers, or $x \wedge y \in \mathbb{Z}$) it is straightforward to reason about the affect of executing S on the program state symbolically. Before S is executed we know that $y = \alpha$ where α is some unknown value from $\mathbb{Z}$. After executing S, then it is the case that $y = \alpha \wedge x = \alpha + 3$. Program statements involving other (subtraction, multiplication, division, etc) arithmetic can be coupled with symbolic semantics in a similar way. The collection of symbolic relations that are true at a given stage of program execution is known as a path condition (PC).

Program control statements (such as if-else blocks and loops) can also be reasoned about symbolically. Consider the small program from listing 2.1.

```
int x, y, z;
if(x > y)
        z = x + y; //S1
else
        z = x - y; //S2
```

Listing 2.1: *Program fragment p1*

Prior to the if-else branch $y = \alpha \wedge x = \beta \wedge z = \gamma \in PC$. PC does not contain any information that can be used to determine if the true branch of the if-else will be followed or if the false branch will be followed because the values of x and y are unknown. This means that both branches will be explored by a symbolic execution. When the case $x > y$ is explored, the path condition is ammended to $y = \alpha \wedge x = \beta \wedge z = \gamma \wedge \beta > \alpha$ after $S1$ is executed then the PC becomes $y = \alpha \wedge x = \beta \wedge z = \alpha + \beta \wedge \beta > \alpha$. When the case $x \leq y$ is explored PC becomes $y = \alpha \wedge x = \beta \wedge z = \gamma \wedge \beta \leq \alpha$ before $S2$ and

$y = \alpha \wedge x = \beta \wedge z = \beta = \alpha \wedge \beta \leq \alpha$ after. During an analysis the symbolic execution engine will generate (and store) a path condition for each program path explored.

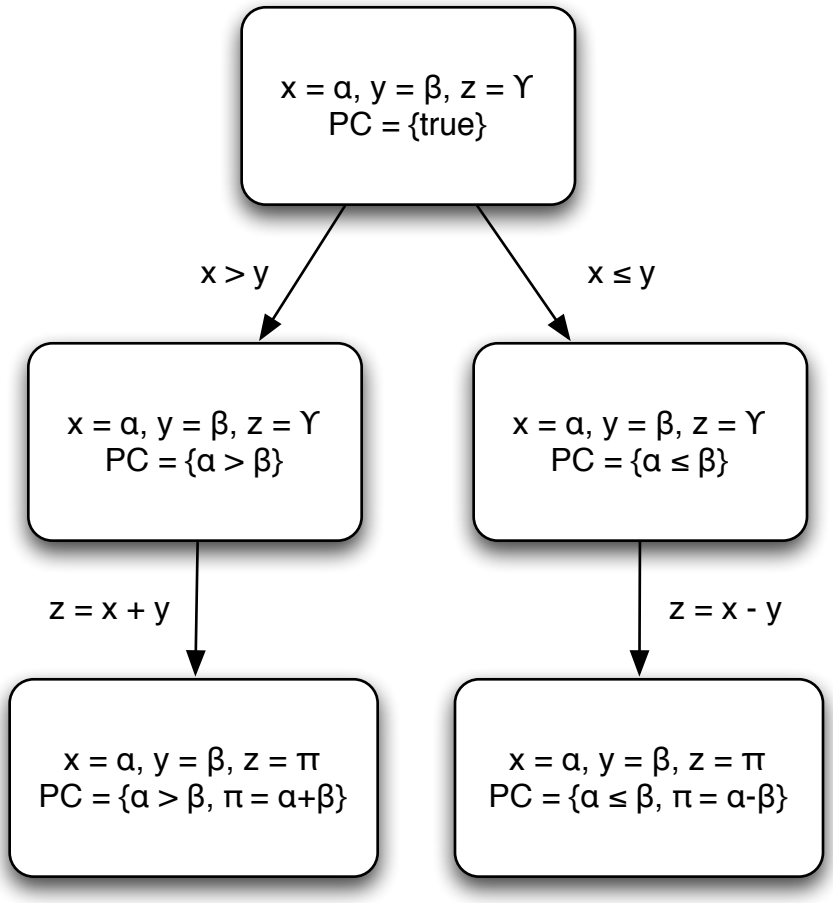


Figure 2.3: *Execution tree for code fragment 1*

A symbolic execution has the property that if each symbol in a PC is instantiated in such a way that satisfies the PC, then the instantiated values are equal to the values that would be produced in a normal, concrete execution when the input variables are assigned the same values as in the *a posteri* instantiation. For example, if for program above α, β, and γ are instantiated such that $\alpha = 2 \wedge \beta = 1$ then $\gamma = 3$, which is equivalent to concretely executing the program where $x = 2 \wedge y = 1$ resulting in $z = 3$. Satisfying assignments to each PC will result in a collection of program inputs that can be used as program or unit test cases.

2.5.1 Symbolic Execution of Imperative Programs with Heap Objects - Lazy Initialization

The method of executing a program symbolically described in the previous section is adequate when the program being checked is written in a language whose data storage (program variable) constructs are clearly bounded in size. (For example, a simplified version of Ada or Pascal where dynamically allocating memory from the heap is disallowed.) Modern programming languages such as Java make heavy use of heap allocated objects and arbitrarily sized and structured datatypes. In the purest sense, the conceptually unbounded heap in the computation model of these languages poses a problem: How can the infinite number of possible heap configurations be characterized symbolically?

A proposed solution (or concession to, depending on your perspective) for this problem, is Lazy Initialization, which was first implemented in conjunction with symbolic execution in JPF[9] (Lazy initialization used in this manner to analyze structured data was first done with the abstract interpretation tool TVLA[10]). Lazy Initialization provides a method for systematically exploring heap configurations in a language like Java that enforce disciplined manipulation of the heap. In Java a variable may be a container for some numerical value (such as float, integer, double, boolean) or a reference to an address in the heap where a structured object is stored. A structured object is a composition of some number of named simple values and some number of named references to other objects. Lazy Initialization semantics for heap object reference access and manipulation is as follows (see algorithm 2 for a more formal description):

1. If a named variable is dereferenced for the first time, non-deterministically consider each possible object that that reference could refer to, append that assignment to the current PC and proceed.

2. If a new object is allocated during the consideration of potential derefence possibilities, add it to the symbolic heap for future considerations on that symbolic path.

Algorithm 2 Lazy Initialization[3]

```
input(f) {f is the field variable under inspection.}
if unintialized(f) then
  if isType(f, T) then
    choose( initialize(f, null) ⋁ initialize(f, new(T)) ⋁ initialize(f, priorObject())
  end if
  if preconditionViolation() then
    backtrack()
  end if
end if
if isPrimitive(f) then
  initPrimitive(f)
end if
```

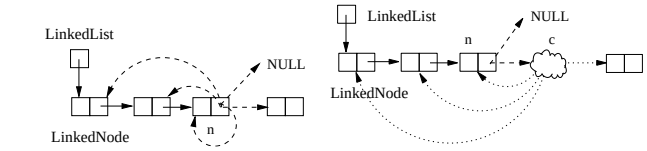
Of course, recursively defined object types and array based datastructures allow for unbounded heap graphs, so a bound on the heap graph can (and should, since it is often the case that software contains inductively defined data structures) be artificially imposed. Different approaches to bounding have been used. Such approaches include bounding the number of objects in the heap directly, bounding the number of method invocations and loop iterations, and bounding the length of a heap reference chain (known as the k-bound, where k is the maximal length of any given reference chain in the heap graph) Imagine the linked-list implementation from listing 2.2.

```
public class LinkedNode{

        private LinkedNode next;
        private Object value;

        public void add(Object o){
                if(next == null){
                        LinkedNode n = new LinkedNode();
                        n.setValue(o);
                        this.next = n;
                }
                else{
                        next.add(o);
                }
        }

        public void setValue(Object o){
                this.value = o;
        }
}
```

Listing 2.2: *Simple linked list implementation*

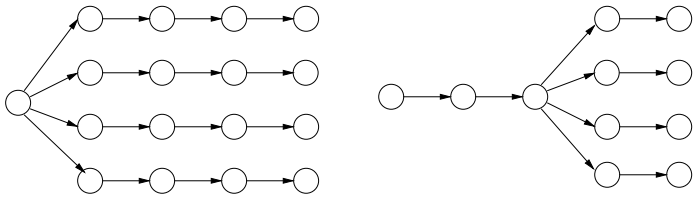
This data-structure is inductively defined, and is unbounded in length. A lazy initialization symbolic execution of the add method with a k-bound of 2 would not consider any heap configurations larger than the ones visualized in figure 2.4(a).

This arbitrary bound on the heap means that in general k-bounded symbolic execution is not considered sound because all possible heap configurations will not be considered during the analysis. However a k-bounded symbolic execution is sound for all heap configurations that have been explored so k-bounded symbolic execution is considered 'relatively sound.' Also, as an added benefit, Lazy initialization and its derivatives (lazier and lazier♯, explained later) will only explore non-isomorphic heap configurations.



(a) Linked list non-deterministic choices with Lazy Initialization

(b) Linked list non-deterministic choices with Lazier Initialization



(c) execution trace for lazy initialization

(d) execution trace for lazier initialization

Figure 2.4: *Visual comparision of Lazy and Lazier Initialization*[1]

2.5.2 Lazier Initialization

While lazy initialization provides a technique to systematically explore all possible heap configurations relevant to a given program, its inherent non-determinism can result in a significantly expensive analysis as the number of potential heap objects grows. In order to reduce the degree of exponential blowup created by the non-deterministic choices Lazy Initialization was refined by Deng, et al[1] into Lazier Initialization.

The purpose of the original Lazy Initialization was the systematic unfolding of possible heap configurations as the enumeration of those possibilities became needed by the underlying symbolic execution. As it turns out, Lazy Initialization non-deterministically unfolds heap configuration possibilities too early. For a given object reference o that has field f, Lazy Initialization will non-deterministically choose among all possibilities for f the first time f is accessed in any way. In fact, it is possible to delay the expansion of the possibilities for $o.f$ until either $o.f$ is used in a non-primitive field access (such as $o.f.g$), an equality test ($o.f = z$) or as a receiver object in a method call. The initialization of $o.f$ to a concrete

heap object is delayed by instead temporarily initializing $o.f$ to a symbolic object place holder until any of the previously described three conditions are met.

A beneficial consequence of this delay is that the symbolic computation tree becomes thinned out at a given height, greatly reducing the total number of symbolic states that must be explored during the course of the analysis. The following example was first presented in *Towards A Case-Optimal Symbolic Execution Algorithm for Analyzing Strong Properties of Object-Oriented Programs*[11]:

```
public class Node<E>{
        private Node<E> next;
        private E data;
        //@ensuresdata == \old(n.data) && n.data == \old(data);
        public void swap(@NonNull Node<E> n){
                E e = data;
                data = n.data;
                n.data = e;
        }
}
```

Listing 2.3: *Simple swap example*

Figures 2.5 and 2.6 compare the execution trees for the symbolic execution of the swap unit from figure 2.3. Using standard lazy initialization the symbolic execution must explore more heap configurations than with lazier initialization. When using lazier initialization symbolic execution has fewer possibilites for reference assignment at each nondeterministic choice. As the tree is explored more deeply, the difference in the number of current choices resulting from the decrease in past choices is dramatic. For example, in lazy initialization the analysis starts with 3 choices, while with lazy initialization there is two. At the last level there are 20 choices with lazy initialization but only 6 with lazier!

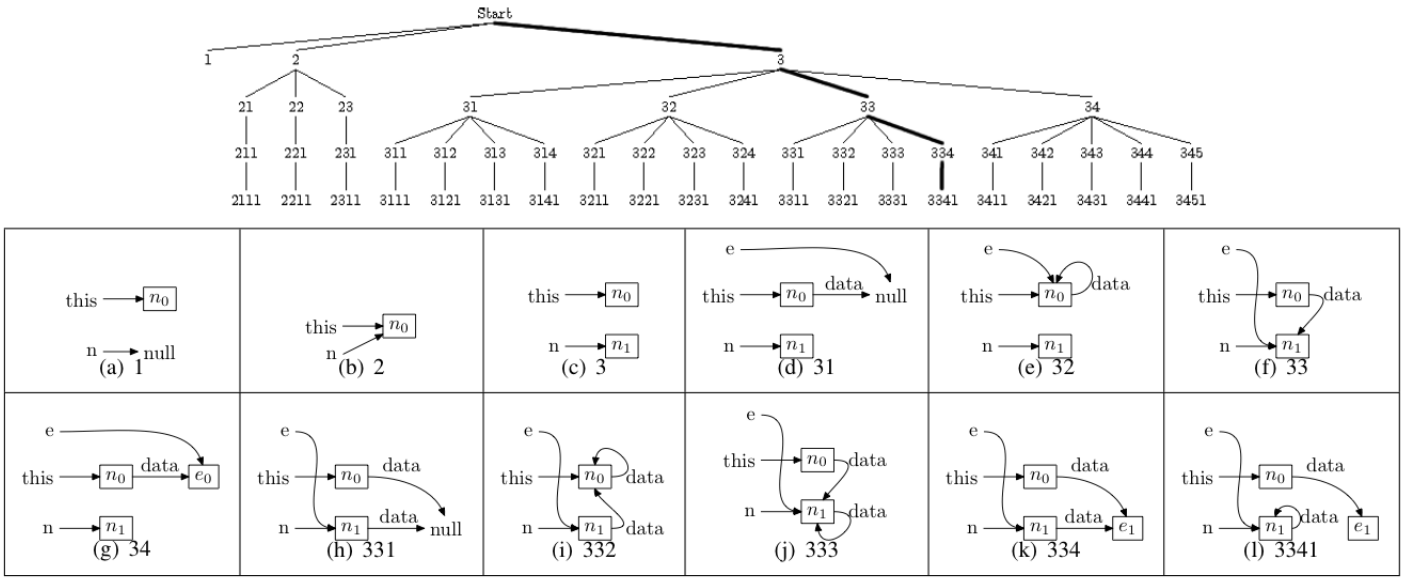


Figure 2.5: *Lazy Symbolic Execution Tree and An Example Trace (3-33-334-3341 and Sibling States)*

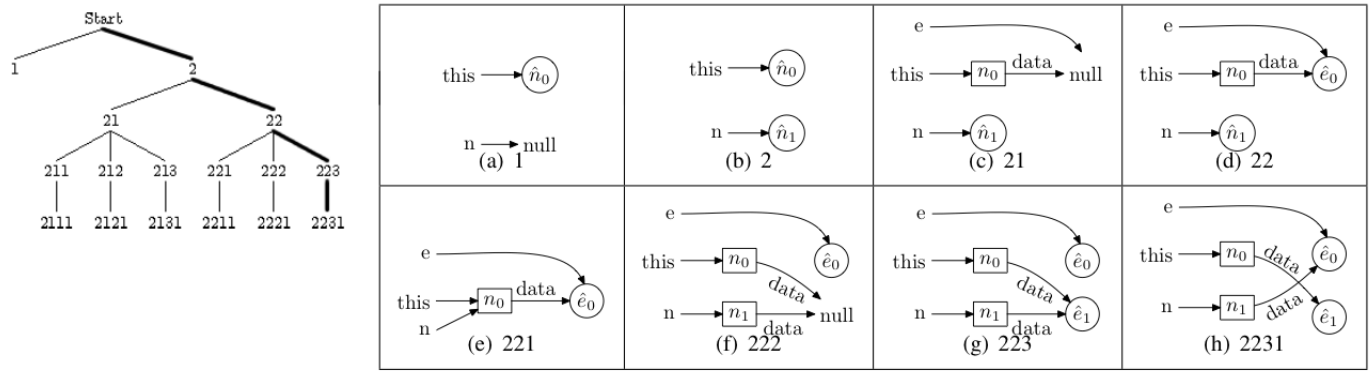


Figure 2.6: *Lazier Symbolic Execution Tree and An Example Trace (2-22-223-2231 and Sibling States)*

2.5.3 Lazier♯ Intialization

Lazier Initialization is not optimal with respect to the number of heap configurations explored[11]. It prematurely expands certain heap configurations leading to an unecessarily

bloated computation tree. The cause of the inefficiency is Lazier Initialization's handling of the initialization of null values. Once one of the three initialization conditions is met (see section 2.5.1) Lazier Initialization immediately initializes the accessed field to either `null` or any of the other type correct objects in the heap. In many situations during execution it is inconsequential whether the object in question is `null` or not, and this choice may be deferred until later.

Thus Lazier♯ Initialization improves upon Lazier Initialization by returning a temporary place holder that indicates a given reference may be `null` when one of the three conditions from section 2.5.1 are met. Later, if the `null`ness of the object is in question (for instance in a `null`ness test) then the symbolic execution can non-deterministically choose whether the reference is `null` or not according to the constraints in the current PC.

As with Lazier Initialization this heap configuration unfolding delay results in a sparser computation tree and less symbolic cases to explore. In Deng, et al[11] it was shown that Laziest Initialization is case optimal. Figure 2.7 shows the execution tree of 2.3 when analyzed with lazier♯ initialization.

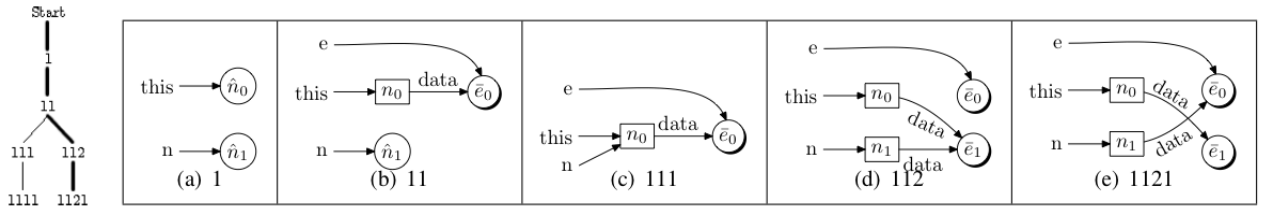
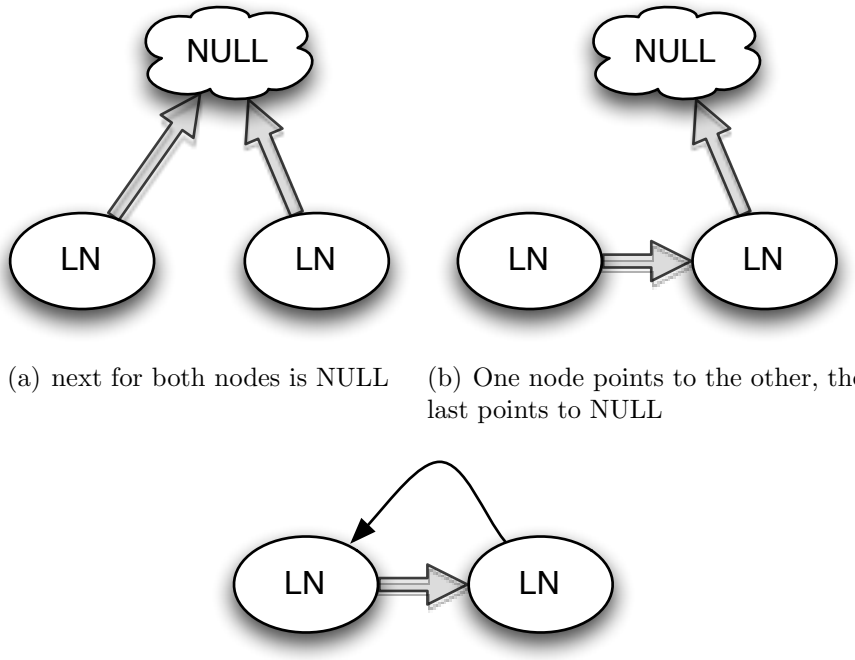


Figure 2.7: *Lazier♯ Symbolic Execution Tree and An Example Trace (1-11-112-1121 and Sibling States)*

2.6 Structure Analysis for Testing

Structure analysis for testing is a technique that aids in the creation of unit test cases. Given some complex type definition (e.g., a Java type), structure analysis will automatically generate object graphs according to that type specification up to a certain bound. Sometimes

a *representation predicate* is used to prune test cases from the test corpus that are not relevant to the unit's functional behavior (i.e. the test case in question does not conform to the units pre-condition).



(a) next for both nodes is NULL

(b) One node points to the other, the last points to NULL

(c) Nodes are in a cycle

Figure 2.8: *Three structures generated by a structure analysis on LinkedNode*

Consider the *LinkedList* from listing 2.2. Based on *LinkedNode*'s type specification, there are 3 possible non-isomorphic *LinkedList* structures with 2 *LinkedNode*s (assuming the generation does not allow self-cycles) Refer to figure 6.2. If the representation predicate returns *false* when there is a cycle 6.2(c) would be pruned from the test corpus.

2.7 Parallel and Distributed Computation

The remainder of this chapter describes different forms of distributed and/or parallel computation that are most relevent to my implementation of parallel Kiasan.

Algorithm 3 Distributed Bag of Tasks Coordinator Process

```
while true do
  workerConn ← receiveCon()
  req ← reqType(workerConn)
  if req = getTask then
    send(workerConn, getTaskFromBag())
  else
    putTaskIntoBag(getTaskFromConn(workerConn))
  end if
end while
```

Algorithm 4 Distributed Bag of Tasks Worker Process

```
while true do
  task ← getTaskFromCoord()
  result ← execute(task)
  if containsTask(result) then
    sendTaskToCoord(result)
  end if
end while
```

2.8 Distributed Bag of Tasks / Coordinator-Workers

The 'Distributed Bag of Tasks'[12], also known as Coordinator-Worker, is a programming pattern that can be used to implement parallel algorithms. The coordinator is a special process that maintains a collection of independent tasks. One or more workers are connected to the coordinator, either via shared variable or some message passing framework. A worker retrieves a task from the coordinator, executes the task, and then possibly adds more tasks to the bag.

2.9 MapReduce

MapReduce is a simple low level programming model developed at Google to support massively parallel programming[13]. MapReduce requires that programmers factor computation into two distinct phases: A *Map* phase, and a *Reduce* phase:

1. $Map(k1,v1) \rightarrow list(k2, v2)$

2. $Reduce(k2,list(v2)) \rightarrow list(v3)$

Each of these phases are loosely[14] inspired[13] by the Map and Reduce operators provided by functional programming languages like LISP or Haskell. In Google's MapReduce, the programmer implements a Map function, which takes as input some key value pair, $(k1,v1)$, and then emits one or more intermediate key value pairs: $list(k2,\ v2)$. The programmer must also implement a Reduce function, which takes as input some $(k2,list(v2))$ and returns some $list(v3)$. A MapReduce framework applies the Map function in parallel to all input $(k1,v1)$. The resulting intermediate $list(k2,\ v2)$ is then grouped by keys into a collection of $list(k2,\ v2)$, which is then processed by the Reduce function in parallel. The following pseudo-code demonstrates how a an algorithm for counting the frequency of words in a collection of documents could be implemented in MapReduce.

Algorithm 5 Map function for word frequency counting in a document

```
input (name, document)
for each word w in document do
  EmitIntermediate(w,1)
end for
```

Algorithm 6 Reduce function for word frequency counting in a document

```
input (word, list(counts))
for each count in list(counts) do
  result ← result + count
end for
Emit(result)
```

In addition to the programming model described here, MapReduce also is a framework designed to support the programming model. The framework is responsible for determining how to partition the input data and the intermediate keys, as well as how to dispatch data to the various Map or Reduce functions. There are no stringent requirements on how to implement a MapReduce framework except that it must support the MapReduce programming model.

The MapReduce programming model is particularly straightforward to use if the algorithm the developer is trying to implement can easily be conceptualized as working on a large set of initial data.

Chapter 3

Parallel Symbolic Execution

3.1 Paralellization

Symbolic execution as described in 2.5 can be very effective in uncovering bugs and generating test cases for a wide range of complex code. The practicality of applying symbolic execution is dependent on the bounding settings required to achieve sufficient code coverage (the loop call, method call, and k-bound). It is not uncommon that some examples of real world code won't be fully covered by an analysis with conservative bound settings. For example, during the symbolic analysis of the redblacktree.put() unit, serial symbolic analysis is capable of exploring all 12,000 heap configurations that are possible with k=4 in under 2 minutes on a reasonably powerful workstation (i.e. 3Ghz Core2 Xeon.) This bound is not adequate for complete code coverage, and there could still be unfound programming errors. For k=5, there are more than 170 million possible heap configurations. The amount of time required to complete a k=5 analysis is more than 3 orders of magnitude greater than a k=4 analysis. The lower bound on the time required for a k=5 analysis is on the order of several days, which is too long to integrate into something like a nightly build process.

While previously implemented versions of symbolic execution have been engineered as serial algorithms, there exists inherent parallelism in the symbolic execution algorithm. It makes sense to leverage this inherent parallelism to take advantage of the recent move towards multi-core (multi CPU) systems and clusters of computers a software development

organization may have at its disposal. In the following sections I will discuss how symbolic execution can be processed by a parallel system. Chapter 4 will contain the details concerning the implementation of parallel Kiasan.

3.2 Parallelism in Symbolic Execution

A nice characteristic of any symbolic execution tree is that any two non-overlapping subtrees in the larger tree are completely independent from one another. Effectively this means that each path condition PC for a given program is independent from any other PC for that same program. Essentially, each PC represents a different causally independent execution of that program. Therefore each PC for some program p can be explored independently by separate processors or computers.

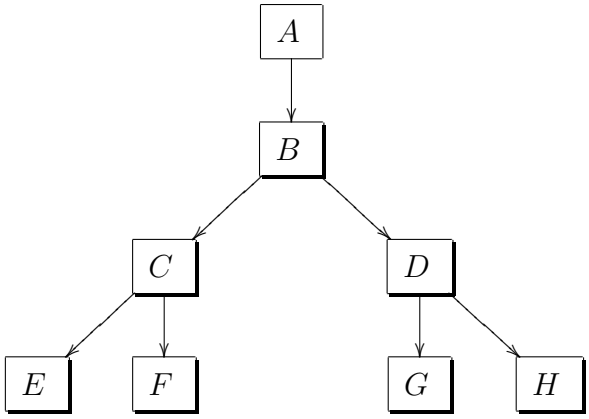


Figure 3.1: *A small execution tree. The sub-trees with roots C and D do not overlap and are therefore independent from one another.*

Unfortunately, each PC is not known *a priori* to the analysis, so there is not a straight forward way to assign the analysis of independant PC to separate processors; The number of PC that will result from a given analysis is unknown. Instead, other techniques must be employed in order to exploit symbolic execution's parallelism while still making efficient use of multiple processors.

3.2.1 Parallelization

While the total work set of a given analysis is not known *a priori*, symbolic execution can be parallelized when it becomes clear that the current path condition being developed bifurcates into two independent path conditions. For example, refer to the program in listing 2.1 and its execution tree in figure 2.3. Informally, parallelization occurs as follows.

1. Symbolic execution starts in the initial state. The shape of the tree is unknown. (Only the root node in figure 2.3 exists)

2. Symbolic execution proceeds and uncovers a non-deterministic choice (either $x > y$ or $x \leq y$.)

3. With more than one non-deterministic choice, check to see if any workers are idle. If there are idle workers, then one choice is assigned to an idle worker. The other choice is taken by the current worker.

In this way, different sub trees of a given program's symbolic execution tree can be assigned to different processors.

Consider figure 3.2. Each node of the tree represents a location of non-deterministic choice. Each arc leaving a node represents a choice in particular. Imagine the symbolic execution of a program whose execution tree is structurally similar to the one represented in figure 3.2. If the symbolic execution had just uncovered the first location of non-deterministic choice (the root node of the tree) then it would know that there are three non-deterministic choices that could be made. Any two of those choices could be assigned to other free workers, and the remaining choice would be computed by the current worker. However, the size and structure of the sub-trees are unknown (the symbolic state space has not been sufficiently unfolded yet.)

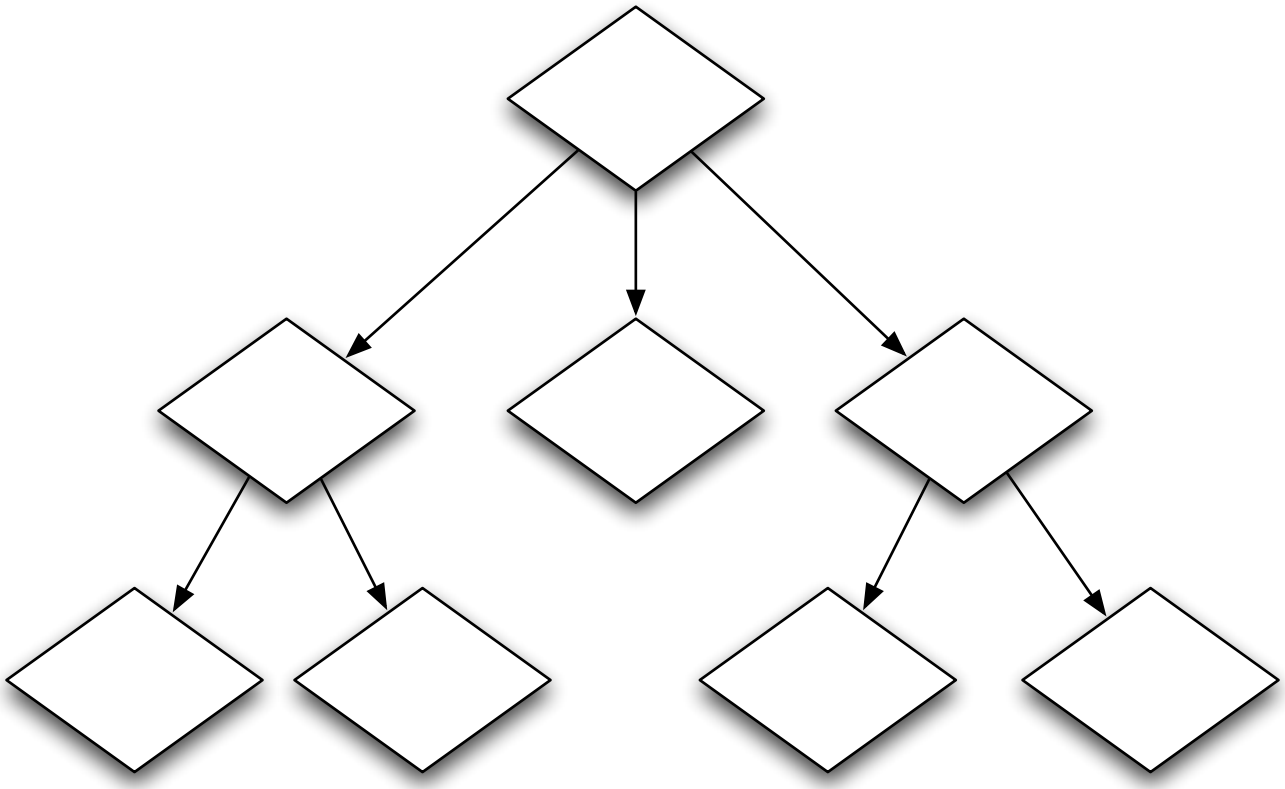


Figure 3.2: *Execution tree for some hypothetical program*

3.2.2 Parallelization - Load Balancing

Like any parallel algorithm, parallel symbolic execution incurs processor and memory overhead managing the parallelization of the analysis. Because the cost (in processing time) of any subtree of the execution tree is not known in advance there is no guarantee that a given analysis will benefit from parallelization. In fact, there are some situations where parallelizing the analysis will increase the amount of time required to complete that analysis. These situations occur when the cost of coordinating the parallelization of a subtree dominates the cost of simply performing symbolic execution on that subtree. Less obviously, attempting to parallelize every non-deterministic choice will result in a drastic slowdown of the total analysis as the sum of the resources expended during coordination activities will always exceed non-parallel analysis time.

While there is no known way in general to predict the computational resource demands of a given symbolic execution, simple commonsense heuristics can be used to choose when to parallelize a non-deterministic choice, especially if it is likely the symbolic execution as a whole will be very computationally intensive.

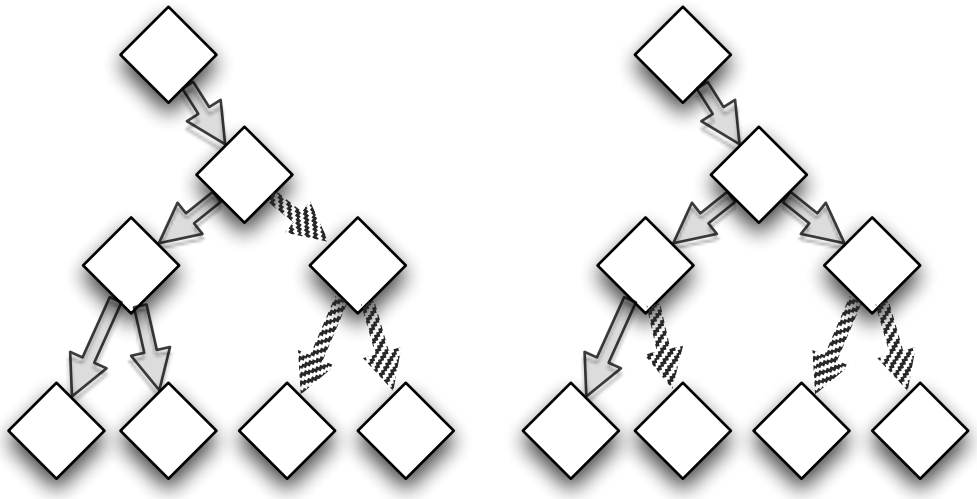
Instead of choosing when to parallelize based on the structure of the symbolic execution tree, the analysis could instead parallelize when computational resources become free. Imagine some computer that is able to run 8 threads simultaneously (e.g. An 8-CPU computer.) Prior to analysis the machine has the capacity to analyze 8 subtrees concurrently. Once analysis begins but before the first non-deterministic choice location, 1 processor is involved in the analysis and 7 are free. If the program from 3.2 is under analysis, then 2 of the three choices from the root node may be assigned to other workers (the current worker continues processing one of the choices.) At this stage there are 3 threads active and 5 threads free. If the thread executing the left-most or right-most subtrees encounters one of the two remaining non-deterministic choice locations they will be able to parallelize up to three of those choices. If any worker thread completes its analysis of its assigned sub-tree then that thread becomes idle, and can later be assigned a different subtree. Equations 3.1

and 3.2 formalize these different parallelization conditions.

$$| \, remainingChoices \, | > 0 \tag{3.1}$$

$$| \, remainingChoices \, | > 0 \wedge \, | \, idleWorkers \, | > 0 \tag{3.2}$$

Figure 3.3: *Parallelization hueristics*

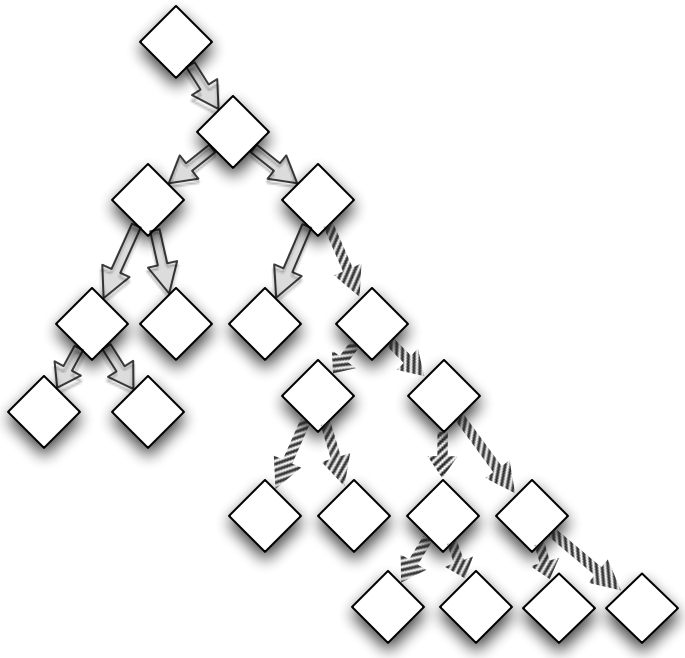


(a) Good parallelization (b) poor parallelization

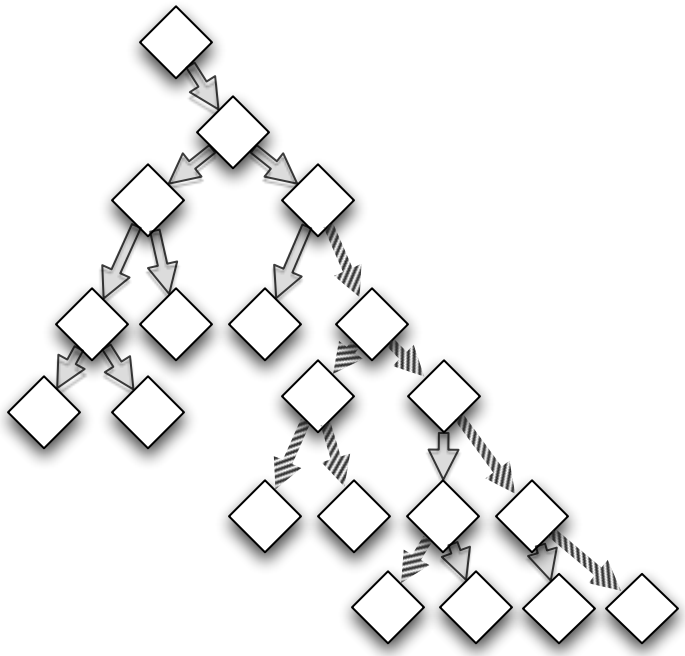
Figure 3.4: *'Optimally' shaped execution trees for parallelization*

3.2.3 Execution Tree Structure

As alluded to in the previous section, the structure of the symbolic execution tree can have a large impact on the effectiveness of load-balanced parallelism. Two major properties of the symbolic execution tree will affect how effective parallelization will be:



(a) Good parallelization



(b) poor parallelization

Figure 3.5: *'Realistically' shaped execution trees for parallelization*

1. Compute time between non-deterministic choice.

2. The density of 'extremely' inner nodes.

1 simply means that if parallelization can only occur at a non-deterministic choice, if it takes less time to compute the sub-tree for that choice than to parallize it then the parallelization is probably wasted effort. 2 is a little more subtle. Optimally, one would want to parallelize subtrees whose cost to analyze is high, or whose cost is such that the new work-unit will complete when other active workers complete. This implies that trees who have lots of inner nodes near the root(Figure 3.6) will probably parallelize much better than trees with 'sparse foliage' (Figure 3.7.)

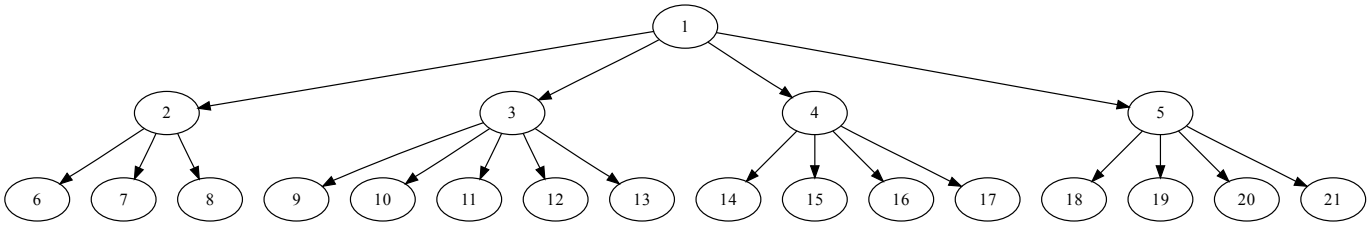


Figure 3.6: *A 'dense' execution tree*

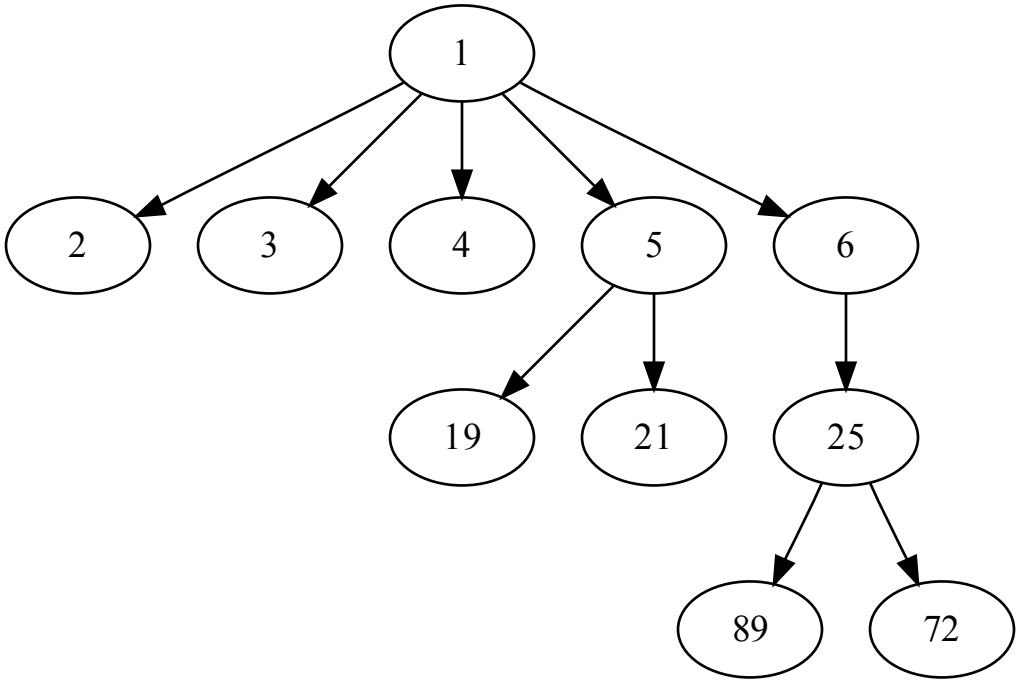


Figure 3.7: *A 'sparse' execution tree*

3.3 WorkUnits

What defines a work unit? Remember that a work-unit might contain an analysis where the analysis engine will have to start somewhere in the middle of an execution tree. Therefore the work unit needs to contain enough information that a worker could reconstruct an internal state appropriate to the parallelization point of that workunit. This section describes two ways of encoding work units.

3.3.1 State Based Workunits

Each worker carries some state S which describes the state of the model it is analyzing. Typically S contains a collection of information such as:

1. H is the model's heap.

2. S is a stack of StackFrames.

3. PC is the current Path Condition.

4. Sym is the set of symbolic variables used.

A state based workunit is the more conceptually simple method of encoding a workunit. A state based work unit encodes the relevant state S of a worker at the time of parallelization. When a worker first receives a state based workunit it must appropriately initialize its symbolic execution core with the encoded state information.

3.3.2 Schedule Based Workunits

Another way of encoding a work unit is to simply carry the *execution schedule* for a particular point of the execution tree.

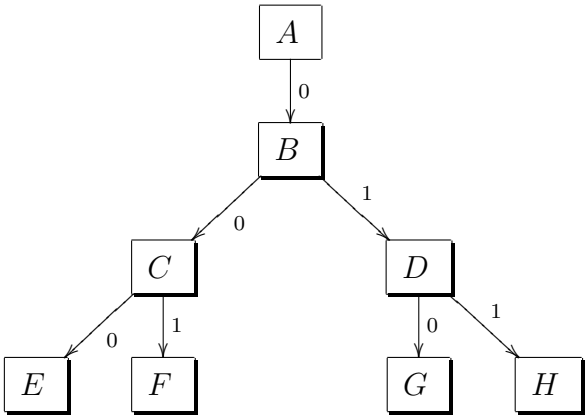


Figure 3.8: *A small execution tree with each choice labeled by its index*

Refer to the simple execution tree in figure 3.8. Imagine that a worker is in state D and wants to parallelize. The worker could parallelize choice 0 or choice 1. Let's say that the worker parallelizes choice 1. This means the workunit will contain the execution schedule: $< 0, 1, 1 >$. To reconstruct the model state H, the other worker simply needs to choose the choice defined to the encoded execution schedule rather than what the symbolic execution

would normally provide. While this approach yields a very simple workunit, the longer the execution schedule is the longer it will take a worker to reconstruct the appropriate state.

3.4 Coordinator Process

Parallel symbolic execution is implemented using the Bag of Tasks distributed programming model of section 2.8. Thus, a central coordination process is required to manage the 'task bag', as well as perform any other accounting tasks relevant to the overall analysis. At its simplest, the coordinator should be able to:

1. Manage the bag of tasks
2. Detect when a given analysis is complete.

Detecting the completion of the analysis is not straight forward as the workers themselves do not know anything about the analysis as a whole, only the suubtree they are currently processing. Instead, a simple invariant can be used to describe the state of the analysis at a high level: If $numWUCheckedOut > 0 \bigvee \|WUQueue\| > 0$ then the analysis is not complete. Thus, in addition to the task queue, a distributed symbolic execution coordinator needs to maintain the number of work units that are checked out.

Chapter 4

Implementation Details

Over the course of this project there have been two major versions of Kiasan: Kiasan/Bogor and Kiasan/Sireum. Kiasan/Bogor was developed first. Kiasan/Bogor extended the Bogor model checking framework. Kiasan/Sireum is a fresh implementation of symbolic execution for Java. Kiasan/Sireum is the most recent version of Kiasan and is still under development as of the writing of this thesis.

The ability to parallelize analyses was added to both Kiasan/Bogor and Kiasan/Sireum. The basic conceptual machinery underlying each implementation is equivalent, however Kiasan/Sireum is much more optimized and generally exhibits drastically better performance characteristics.

4.1 Kiasan/Bogor

Distributed symbolic execution was first implemented on Kiasan/Bogor. Kiasan/Bogor is an implementation of symbolic execution for Java built on top of the Bogor model checking framework. Bogor implements a Java byte-code interpreter that directly model-checks Java bytecode. Kiasan/Bogor directly extends the built in interpreter component to operate with symbolic semantics. Kiasan/Bogor also does not fully support symbolic execution of multi-threaded programs, as the symbolic interpreter did not have semantics for the `MONITORENTER` and `MONITOREXIT` bytecodes. Kiasan/Bogor was used as a testbed for the distributed analysis concepts that are the basis of this thesis and were implemented in a much more robust

fashion for Kiasan/Sireum.

4.1.1 Kiasan/Bogor - Pull Mode coordination

The 'pull-mode' coordinator is the lightest weight coordinator, and most closely resembles the bag of tasks model as workers are completely responsible for connecting and retrieving tasks. The coordination server maintains a queue of pending work units, tracks the number of threads that are idle, and a 'completion condition' which is a boolean expression that becomes true when the total symbolic analysis is complete. In Pull-Mode coordination each worker thread is assigned the same program to analyze. Instead of commencing analysis, the workers communicate amongst one another and elect the 'start worker.' The start worker then commences analysis. When the start worker encounters a point of parallelization, it generates a work unit and transmits it to the coordination server. One of the other workers will pull that workunit from the queue and do likewise until all workers are not idle.

4.1.2 Kiasan/Bogor - Schedule Encoded Work-Unit

Kiasan/Bogor supported schedule-based workunits: schedules were encoded and decoded using a simple hand tuned serializer/deserializer into and from comma delimited integer lists. As a performance increasing measure, Kiasan/Bogor would disable the theorem prover while it was replaying a schedule (The theorem prover would not be needed to make decisions that had already been made.)

4.1.3 Kiasan/Bogor - State Encoded Work-Unit

Kiasan/Bogor supported state-based workunits: a large portion of the Kiasan/Bogor implementation was the creation of hand tune serializers and deserializers for the state-based work units. These hand tuned serializers provided low-latency serialization and deserialization compared to general frameworks like XMLBeans at the time but they tended to be very complicated. (The fully implemented serializer/deserializer consisted of over 1000 lines

Algorithm 7 Kiasan/Bogor Coordinator Server worker thread process; used to manage a connection between a worker process and the server

```
input(workerConnection)
id ← getNextAvailableWorkerID()
addWorkerToSystem(id, workerConnection)
addToIdleQueue(workerConnection)
runLoop ← true
while runLoop do
  workerMessage ← blockForMessage(workerConnection)
  if workerMessage = PING then
    enterMonitor(idleWorkerQueue)
    if numIdleWorkers() > 0 then
      sendMessage(workerConnection, IDLE)
    else
      sendMessage(workerConnection, NOTIDLE)
    end if
    exitMonitor(idleWorkerQueue)
  else if workerMessage = CLIENT-COMPLETES-WORKUNIT then
    taskId ← getTaskIdFromWorker(workerConnection)
    task ← getTask(taskId)
    enterMonitor(task)
    checkInWU(task) {synchronized on the individual task datastructure}
    exitMonitor(task)
  else if workerMessage = GET-WORKUNIT then
    enterMonitor(wuQueue) {idle workers can continue to poll asking for tasks creating
    some contention on the wuQueue}
    workUnit ← dequeue(wuQueue)
    exitMonitor(wuQueue)
    sendMessage(workerConnection, workUnit)
  else
    skip
  end if
end while
```

Algorithm 8 Kiasan/Bogor worker process implementing pull mode

```
serverConnection ← connectToServer()
while true do
  workUnit ← getWorkUnit(serverConnection)
  while !received(workUnit) do
    workUnit ← getWorkUnit(serverConnection) {Poll server for workunit}
  end while
  setWorkerState(workUnit) {initialize worker with either state or schedule}
  explore()
  sendMessage(serverConnection, CLIENT-COMPLETES-WORKUNIT)
end while
```

of Java string-manipulation code). This made it very difficult to change the structure or contents of the workunits as Kiasan/Bogor evolved.

4.1.4 Kiasan/Bogor - Asynchronus Work Unit Commits

Late in the development of Kiasan/Bogor asynchronous commits were added as an optimization to the coordinator server. This optimization was primarily designed to aid in the scalability of the coordinator system. The coordinator contains several data-structures and variables that are used to account for the overall progress of a parallel symbolic execution. These variables and data-structures are guarded by monitors, a basic concurrent locking mechanism provided by the Java run-time. As the number of worker threads increase, the contention for these shared resources also increases. If a worker is committing a workunit to the coordination system and becomes blocked while waiting for access to a resource, then the worker thread will idle wait until the resource is free, preventing it from working on its subtree.

In order to reduce the need for direct locking on the shared queue, each worker is paired with a thread running in the coordinator server. This coordinator thread, and the worker thread share a local queue. Instead of posting work unit commits to the central shared queue, the worker thread posts the commit to the local queue. Once the size of the local queue is greater than 0, the coordinator thread comes alive, and transfers the workunits

from the local queue to the shared queue, performing all changes to the accounting variables previously driven by the worker thread.

4.2 Kiasan/Sireum

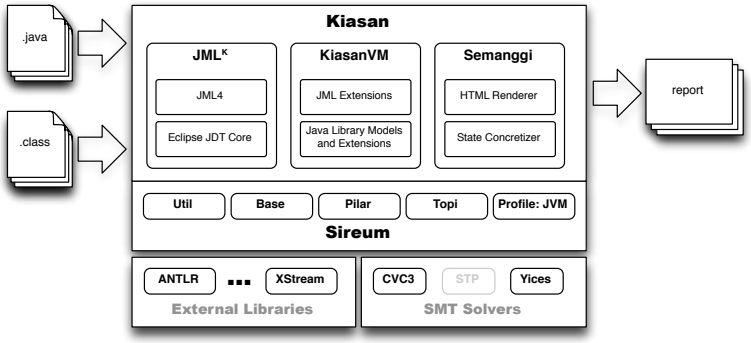


Figure 4.1: *Kiasan's pipelined architecture*

Kiasan/Sireum is a fresh implementation of the algorithms and ideas that were prototyped in Kiasan/Bogor. Kiasan/Sireum improved upon Kiasan/Bogor by having a more robust facility to deal with unit specifications written in the JML specification language (including a compiler that will compile JML annotations into a form Kiasan can check against), by improving integration with various theorem provers (Kiasan/Sireum communicates via the Yices and CVC3 theorem provers via native binary calls as opposed to a UNIX pipe.) a comprehensive HTML report generator that summarizes code coverage and errors discovered, and a much more streamlined implementation of parallel symbolic execution.

Kiasan consists of 4 main modules: (1) JML^{k}, (2)KiasanVM, and (3) Semanggi. The modules process a program for analysis in a pipeline as visualized in figure 4.1.

4.2.1 JML^{k}

JML^{k} is a customized version of JML4 (a Java unit contract specification language) for Kiasan. Given a unit's Java source code and its corresponding bytecode, it translates the JML specifications in the sourcecode to a Kiasan-amenable and executable code (Java source

code) and then JML^k compiles it to Java bytecode form. Both the unit and its specification bytecode (as well as some configuration options, e.g., for contract- code substitutions) are then given to KiasanVM for analysis.

4.2.2 KiasanVM

KiasanVM is a symbolic virtual machine for symbolically executing Java bytecode. As it executes and analyzes the code, KiasanVM notifies users about possible safety violations and points to program locations where such violations may occur. In addition, KiasanVM measures the code instruction and branch coverage as well as other statistics such as execution times. At the end of execution, KiasanVM generates a (XML formatted) report consisting of execution statistics as well as representative (abstract) pre-/post-states of the unit to illustrate the behaviors the analysis has covered. KiasanVM then forwards the generated report to Semanggi for processing.

As of this writing, some Java byte cores are not implemented in KiasanVM: `LDC` (used for class loading), `NEWMULTIARRAY` (allocates memory for multidimensional arrays), and the concurrency operations `MONITORENTER` and `MONITOREXIT`. In practice this limits KiasanVM's ability to find bugs that may arise from errors in concurrency, and may even result in false-positives in situations where the programmer intended to use concurrency monitors to guard against unwanted program states. KiasanVM is the main module that was modified to support parallel symbolic execution.

4.2.3 Semanggi

The Semanggi component of the Kiasan pipeline processes the raw XML unit reports generated by KiasanVM and renders a human readable analysis report in HTML/Javascript. Semanggi's output contains analysis and coverage information such as package level, class level, and method level statistics as well as highlighted source code that visualizes what program statements were analyzed. In addition to coverage statistics, Semanggi uses a constraint solver to construct graphical visualizations of pre-method call and post-method call

heap configurations for each path explored through the unit.

4.3 Kiasan/Sireum Core Components

This next section describes in some detail the main core components of KiasanVM. These components are directly involved with the symbolic execution and are therefore central to the implementation of a parallel version.

4.3.1 KVMExplorer

The KVMExplorer directly controls the search of the symbolic state space. KVMExplorer implements a depth first search (DFS) of the symbolic state space. The KVMExplorer will symbolically execute the instructions of a Java program via the step() method until no progress can be made (step() returns $false$). If step() returns $false$ then KVMExplorer will repeatedly *backtrack* the search until it can *step* again (i.e. explore a different non-deterministic choice). Refer to algorithm 9.

Algorithm 9 explore() method for Kiasan

```
while !shouldTerminate do
  unCovered ← false
  if !step() then
    if !backtrack() then
      break
    end if
  end if
end while
```

Algorithm 9 has been extended into algorithm 10 in order to support parallelism.

4.3.2 State

State objects describe the current state of a model under analysis. Kiasan/Sireum maintains some extra information in the state in order to support the construction of test cases efficiently after the end of a path condition is reached. Implemented as a Java class named *State*, a state can be thought of as a tuple $< S, H, H^p, S, PC >$ where:

Algorithm 10 explore() method for parallel Kiasan

```
while !shouldTerminate do
  unCovered ← false
  if !step() then
    if !backtrack() then
      break
    end if
  end if
  while shouldDistribute() do
    sendWU()
    backtrack()
    step()
  end while
end while
```

Algorithm 11 shouldDistribute() without worker load balancing

```
checkUncovered()
if shouldBacktrack then
  ret ← false
else
  ret ← true
end if
return ret
```

Algorithm 12 shouldDistribute() for load-balanced Lazy Parallelization

```
checkUncovered()
if shouldBacktrack then
  ret ← false
else
  if pingServer() then
    ret ← true
  else
    ret ← false
  end if
end if
return ret
```

1. S is a map $String \rightarrow Value$ of static field names to their values.

2. H is the model's heap.

3. H^p is the pre-heap (the heap before the method being analyzed is invoked).

4. S is a stack of StackFrames.

5. PC is the current Path Condition.

4.3.3 KVMModelManager

The KVMModelManager maintains the state of the model $< S, I^{id}, L^{id}, F^{id}, D^{id}, R^{id}, T^{id}, SF >$ where:

1. S is the current model state.

2. I^{id} is the last used symbolic integer Id.

3. L^{id} is the last used symbolic long Id.

4. D^{id} is the last used symbolic double Id.

5. R^{id} is the last used symbolic reference (Object) Id.

6. T^{id} is the last used symbolic type variable Id.

Whenever the KVMInterpreter symbolically executes an instruction that operates on a symbolic value it uses the KVMModelManager to either locate the correct symbol, or instantiate a new value.

4.3.4 KVMSchedulingStrategist

The KVMSchedulingStrategist chooses what non-deterministic choice the exploration will follow. Choices available at a given state are deterministically indexed (same choices in the same state will always have the index.) The default KVMSchedulingStrategist will always return the next available uncovered choice ($lastChosenIndex + 1$).

4.3.5 KVMInterpreter

The KVMInterpreter implements the symbolic semantics of the Java byte-code. The KVMInterpreter interacts with Topi (the theorem prover), the KVMSchedulingStrategist and the KVMModelManager to symbolically interpret a Java byte-code given the current path condition.

4.4 Additional Details

Parallel symbolic execution in Kiasan/Sireum is much more refined than in Kiasan/Bogor. Distributed Kiasan/Sireum includes:

1. Pull Mode Coordination - The same coordination style that appeared in Kiasan/Bogor.
2. Push Mode Coordination - A simpler, more flexible, and efficient style of central coordination.
3. Xstream State Snapshot - Uses the Xstream library for serializing and deserializing work units.
4. Intra-Unit Parallelization - parallelizes the analysis of a single unit
5. Inter-Unit Parallelization - parallelizes the analysis of a collection of units.
6. Front-end client - takes raw java source code, leverages JML^{k} to compile contracts, and submits an analysis task to the coordination server.

4.4.1 Kiasan/Sireum - Push Mode Coordination

'Push Mode Coordination' was motivated by the desire to let the coordination server exercise more control over work unit dispatch. In Push Mode Coordination the coordination server maintains a queue of idle workers, as opposed to a queue of work units. As new work-units are generated by the workers, the coordinator will de-queue an idle worker and assign

it the new work unit. In this way, some of the synchronization overhead of pull mode is avoided because only a single internal coordinator thread is polling a queue, as opposed to an arbitrary number of worker threads. In addition, the worker thread logic is reduced, as worker threads no longer need to vote a lead worker.

4.4.2 Kiasan/Sireum - XStream State Snapshot

XStream, a library for serializing plain old java objects (POJOs) into XML has performance that is acceptable for use in state snapshot encoded parallelization. All worker to coordinator and coordinator to worker messages are encoded into XML via the XStream library. These *AnalysisStateMessage*s contain:

1. $< S, I^{id}, L^{id}, F^{id}, D^{id}, R^{id}, T^{id}, SF >$ from the KVMModelManager
2. M the model method being analyzed
3. T the current task id. See 4.4.4
4. J the current job id. See 4.4.4

As of the writing of this thesis, the performance of XStream is very good; it approaches the performance of the hand-tuned serializers of Kiasan/Bogor.

4.4.3 Kiasan/Sireum - Intra-Unit Parallelization

Kiasan/Sireum will perform two types of parallelization and load balancing of a Kiasan cluster. An *Intra-Unit Parallelization* is when the analysis of a unit (in this case Java a method.) Kiasan/Sireum performs parallelization of unit analyses the same way Kiasan/Bogor does, at the points of non-determistic choice in the execution tree. Kiasan/Sireum only uses the 'state snapshot' method to communicate the work units between the workers.

4.4.4 Kiasan/Sireum - Inter-Unit Parallelization

Inter-unit Parallelization is type of parallelization not present in Kiasan/Bogor. The coordination server can accept analysis tasks from an arbitray clients, and each client can submit an analysis task that is composed of multiple units. The server will automatically prioritize Inter-unit parallelization of tasks over intra-unit parallelization of tasks in order to more efficiently use cluster resources.

4.4.5 Kiasan/Sireum- Front-end client

The front end client allows a user to submit a unit, or set of units, for analysis in the Kiasan cluster. The front end takes source, compiles it, and then transmits it to the coordination server which then unpacks the task bundle and parallelizes it across the cluster according to available computational resources. If there are any JML contracts annotating the units under analysis, the front end client will compile the contracts into Kiasan executable code and bundles the contract into the task.

4.4.6 Kiasan/Sireum - Coordinator Server

The Kiasan/Sireum coordinator provides push mode coordination (See section 4.4.1) to Kiasan/Sireum worker processes. The coordinator server supports both state snapshot (see section 4.4.2) and schedule encoded (see section 3.3.2) work units. In fact, schedule encoded and state snapshot workunits can be used at the same time. The coordinator server also supports both intra-unit (section 4.4.3) and inter-unit (section 4.4.4) parallelization. The coordinator can manage the analysis of different units simultaneously.

The coordinator server maintains one thread per worker process to manage communication between the server and the worker. A worker management thread is spawned when a worker process connects to the server. Refer to algorithm 13, which highlights the control loop of such threads.

The server maintains a single ‘work unit dispatch thread‘, which handles assigning and

Algorithm 13 Kiasan/Sireum Coordinator Server worker thread process; used to manage a connection between a worker process and the server

```
input(workerConnection)
id ← getNextAvailableWorkerID()
addWorkerToSystem(id, workerConnection)
addToIdleQueue(workerConnection)
runLoop ← true
while runLoop do
  workerMessage ← blockForMessage(workerConnection)
  if workerMessage = PING then
    enterMonitor(idleWorkerQueue)
    if numIdleWorkers() > 0 then
      sendMessage(workerConnection, IDLE)
    else
      sendMessage(workerConnection, NOTIDLE)
    end if
    exitMonitor(idleWorkerQueue)
  else if workerMessage = CLIENT-COMPLETES-WORKUNIT then
    taskId ← getTaskIdFromWorker(workerConnection)
    task ← getTask(taskId)
    enterMonitor(task)
    checkInWU(task)
    exitMonitor(task)
  else
    skip
  end if
end while
```

Algorithm 14 Kiasan/Sireum worker process implementing push mode

```
serverConnection ← connectToServer()
while true do
  workUnit ← getWorkUnit(serverConnection) {worker blocks until server pushes a
  work unit down}
  setWorkerState(workUnit) {initialize worker with either state or schedule}
  explore()
  sendMessage(serverConnection, CLIENT-COMPLETES-WORKUNIT)
end while
```

Algorithm 15 Kiasan/Sireum Coordinator Server workunit dispatch thread. This thread blocks until the work unit queue has contents, deqeues a work unit, then assigns the work unit to an idle worker process.

```
while true do
  enterMonitor(blockingWUQueue)
  workUnit ← dequeue(blockingWUQueue)
  enterMonitor(blockingWUQueue)
  task ← getTaskForWU(workUnit)
  enterMonitor(blockingIdleWorkerQueue)
  idleWorkerConnection ← dequeue(blockingIdleWorkerQueue)
  exitMonitor(blockingIdleWorkerQueue)
  enterMonitor(task)
  checkOutWU(task)
  exitMonitor(task)
  sendMessage(idleWorkerConnection, workUnit)
end while
```

transmitting work units to idle workers. Refer to algorithm 15 for pseudocode of the dispatch thread. The use of the dispatch thread reduces contention for shared data-structures, like the work unit and task queues, which have to be read frequently in pull mode coordination.

In order to fully support inter-unit parallelization, the server must maintain a database of the transitive closure (source code and dependencies) of the units that will be analyzed. Each full analysis constitutes a job, which includes the relevant source-files, JML contracts, and analysis options. The server spawns a job management thread on demand whenever a FrontEndClient (section 4.4.5) connects to submit a job to the server, or worker process connects to retrieve a job file.

Algorithm 16 Kiasan/Sireum Coordinator Server job management connection thread

```
input(clientConnection)
while true do
  clientRequest ← getMessage(clientConnection)
  if clientRequest = GETJOB then
    jobId ← getJobId(clientConnection)
    jobData ← getJobData(jobId)
    sendMessage(clientConnection, jobData)
  else
    jobId ← nextAvailableJobId()
    jobData ← getJobDataFromConnection(clientConnection)
    putJobData(jobId,jobData)
    task ← generateTask()
    initializeTask(task)
    workUnit ← generateInitialWU(task)
    enqueue(workUnit) {add fresh work unit to the central work unit queue}
  end if
end while
```

Chapter 5

Experiments and Evaluation

This section evaluates the distributed Kiasan with respect to scalability. In theory, the best scalability one can expect is when actual compute throughput scales commensurate with the the addition of compute nodes. For instance, if a given analysis takes 2 minutes with 1 processor, a perfectly scalable distributed analysis will at best return a result in 1 minute. In practice, this level of efficiency may not be achievable due to a number of factors, such as hardware concerns (processor interconnect speed, hardware interuppt management, etc) or software concerns (thread synchronization, operating system thread management, the level of parallelism present in the conceptual problem, message serialization overhead). These experiments show how distributed Kiasan behaves when it is used to analyze different types of program units with different analysis (k-bound, array-bound) options.

5.1 Kiasan 1 Experiments

The Kiasan 1 experiments measure how long an analysis takes as the number of worker processes are increased. These experiments were run on nodes isolated from Kansas State Universities Beocat cluster. The configuration of each node consisted of:

1. 2 AMD Opteron 875 Processors (8 cores total)
2. 32 Gigabytes of RAM
3. 2.6 series Linux kernel (Gentoo Linux)

4. Sun JRE 1.6 (64-bit mode)

The number of workers were varied between 1, 2, 4, 6, and 8 for each experiment.

5.2 Results

5.2.1 State Snapshot Work Units

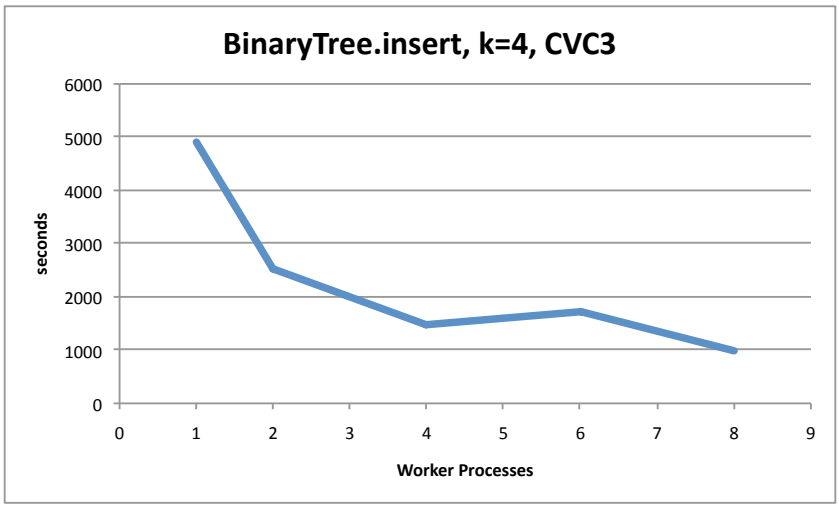


Figure 5.1: *Analysis time vs. number of worker processes, using the CVC3 theorem prover.*

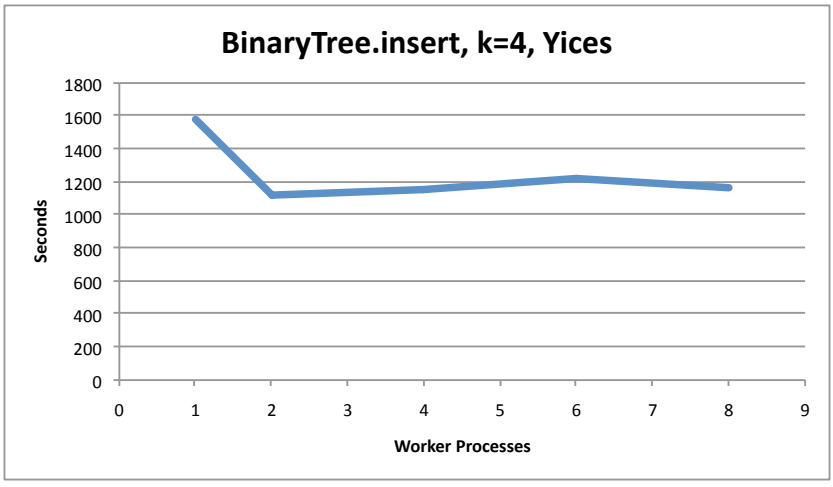


Figure 5.2: *Analysis time vs. number of worker processes, using the Yices theorem prover.*

Figures 5.1 and 5.2 illustrate how inter-choice computation time can have a significant impact of the scalability of distributed Kiasan. Both of those experiments were identical except for the difference in theorem prover. Yices is clearly much faster than CVC3, yielding a dramatic decrease in analysis time regardless of the number of workers. However, even though total analysis time decreased, the amount of computation time between possible parallelizations also decreased, reducing the computation vs. coordination ratio which resulted in much worse scalability as the number of workers were increased.

5.2.2 Schedule-based Work Units

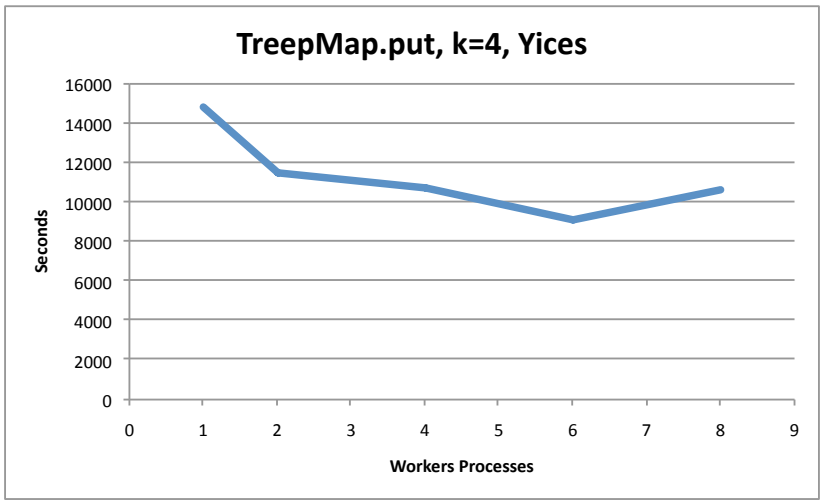


Figure 5.3: *Analysis time vs. number of worker processes, using the Yices theorem prover.*

Figure 5.3 shows the scalabilty of distributed Kiasan analyzing the TreeMap.put unit. The TreeMap tends to be fairly computationally heavy, so even with the use of Yices as the theorem prover the computation vs. coordination ratio stayed large enough to show a general trend of speed-up as the number of workers were increased.

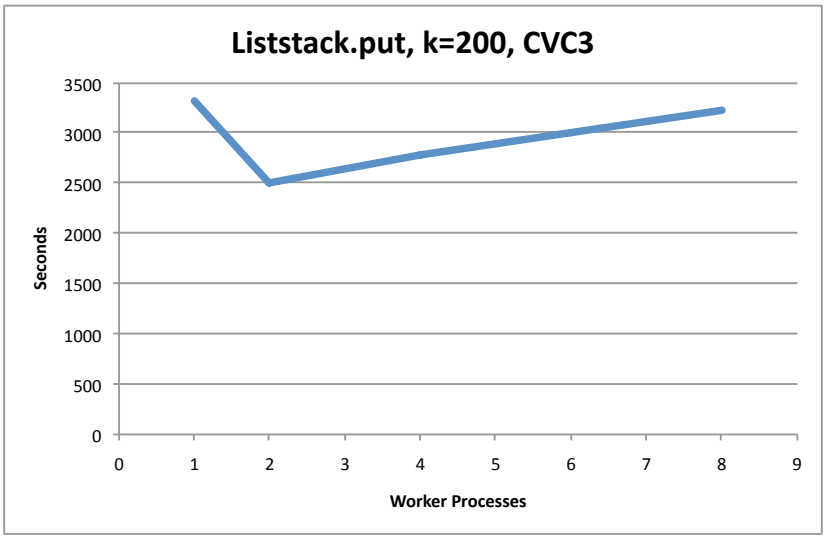


Figure 5.4: *Analysis time vs. number of worker processes, using the CVC3 theorem prover. The k-bound was set to 200 in order to force a long analysis time.*

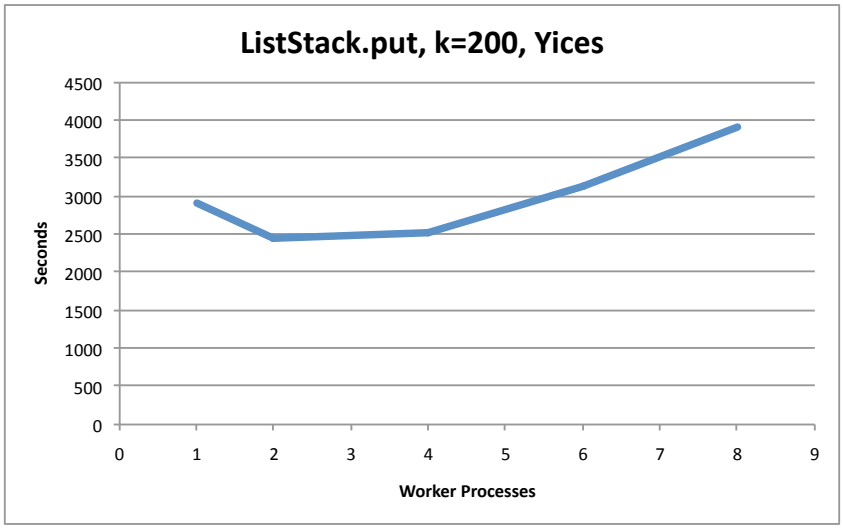


Figure 5.5: *Analysis time vs. number of worker processes, using the Yices theorem prover. The k-bound was set to 200 in order to force a long analysis time.*

Figures 5.4 and 5.5 demonstrate how some units don't exhibit much parallelism. The liststack is a collection from the java.util libraries that uses a linked list to store data for a stack. While the execution tree for this unit and large k-bound is fairly tall, most branches

in this tree are not. This means that there is a high probability that a parallelization may generate a work unit for a computationally light tree. In these situations the computation vs. coordination ratio can drop dramatically as can bee seen in the experimental results.

5.3 Kiasan 2 Experiments

These Kiasan 2 experiments measure, for each unit, computational throughput per minute, total computation completed over a 20 minute period, worker idle time, number of parallel factorizations the occured during the experiment for a given program unit, and total message marshalling time.

5.3.1 Throughput

Computational throughput is measured by the number of 'state-paths' explored per unit time. A state path is simply a trace through the computation tree of the analyzed unit from its root to some leaf. As some of the results will show, the amount of computation time per state-path is not fixed for a given unit. (i.e. some state-paths will take longer to explore than others.) This means that the throughput for a given analysis can't necessarily be used to predict how long a complete analysis will take, however, higher throughput even in a small window of time does represent a real world increase in capability because analysis data can be collated in real time. (More code coverage will be obtained over that window. More test cases will also be generated in that same amount of time.)

5.3.2 Worker idle time

Total worker idle time is the sum of the time each worker is spent in a blocked or non-running state. (The worker is not engaged in any sort of computation, analysis or message serialization.) Both hardware and software limitations will increase the worker idle time. The largest contributor of worker idle time in this system tends to be when worker threads a blocked while accessing shared data-structures, or waiting for those data-structures to

become populated with workunits.

5.3.3 Factorizations

A factorization occurs when analysis parallelism is detected by the system and one or more workunits are generated. More factorizations result in more communications overhead (message serialization/deserialization, shared data structure accesses, network protocol overhead) which may end up dominating useful computation.

5.3.4 Serialization time

The serialization time is the sum total of time across all worker threads spent either serializing a coordination message into a wire ready format or deserializating a wire data stream into a coordination message. Each coordination message contains some basic accounting information (what unit is being analyzed, etc) and a snapshot of the symbolic state including the associated path condition PC. Larger and more complicated messages will require the worker thread to devote more computational time to serializing or deserializing the message.

5.4 Experimental Setup

Each of the 6 Java program units were tested with various analysis options as specified. The large scale (large k- or array bound) experiments, were run for at least 20 minutes each, varying the number of workers for each 20 minute run. During each 20 minute span, the state paths explored during each 1 minute interval was tabulated, resulting in state path per minute throughput numbers for each minute of the 20 minute span. The sum of these are used to calculate the 20 minute totals for each experiment.

The number of workers used for each unit are 1,2,4,6, and 8. The coordinator server was run on the same machine as the worker processes. Each worker is contained in its own JVM. Sun's JRE 1.6.0 update 7 was used in each experiment (JIT was enabled, and each JVM allocated 768 megabytes for heap, and 128 megabytes for MaxPermGen). Yices

was used as the theorem prover. The computer system was a dual processor 2.33 GHz Intel Xeon Harpertown 5140 (8 cores total, 24 MB of L2 cache total). The computer was equipped with 8GB of RAM. The operating system used was Ubuntu 8.04 LTS (Kernel version 2.6.24-22-generic SMP.)

5.5 Errata

Some experiments are missing data-points. At the time the experiments were run, particular experimental configurations would cause one or more of the worker JVMs to crash with bus errors or segmentation faults. The cause of these errors were never discovered, but it is likely that certain expression being pushed to the theorem prover in certain orders were causing memory access problems in the native theorem prover bindings Kiasan provides.

5.6 Results

5.6.1 ArrayPartition.partition(), k=7, ab=100

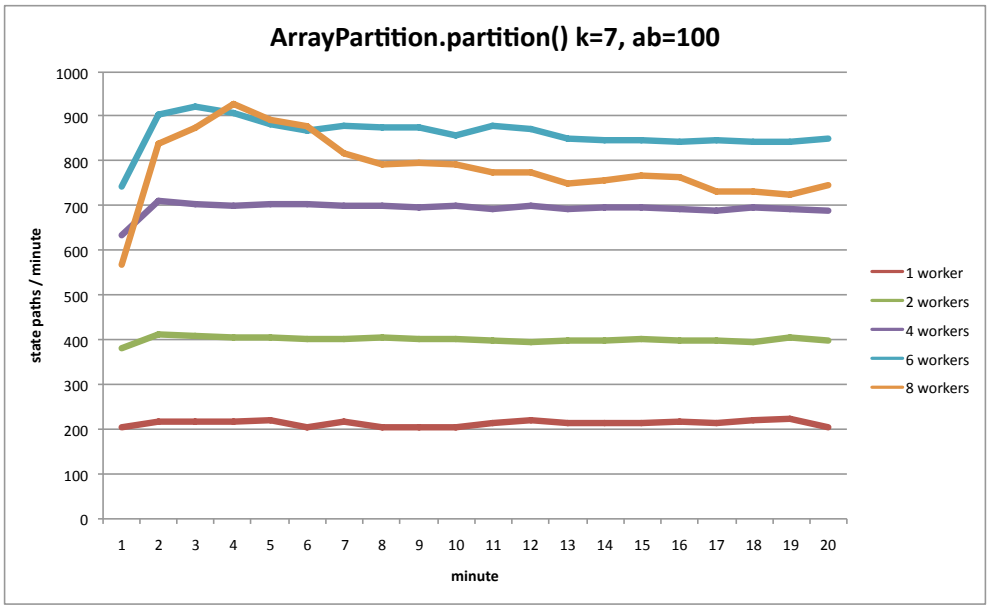


Figure 5.6: *ArrayPartition.partition() state-paths explored per minute*

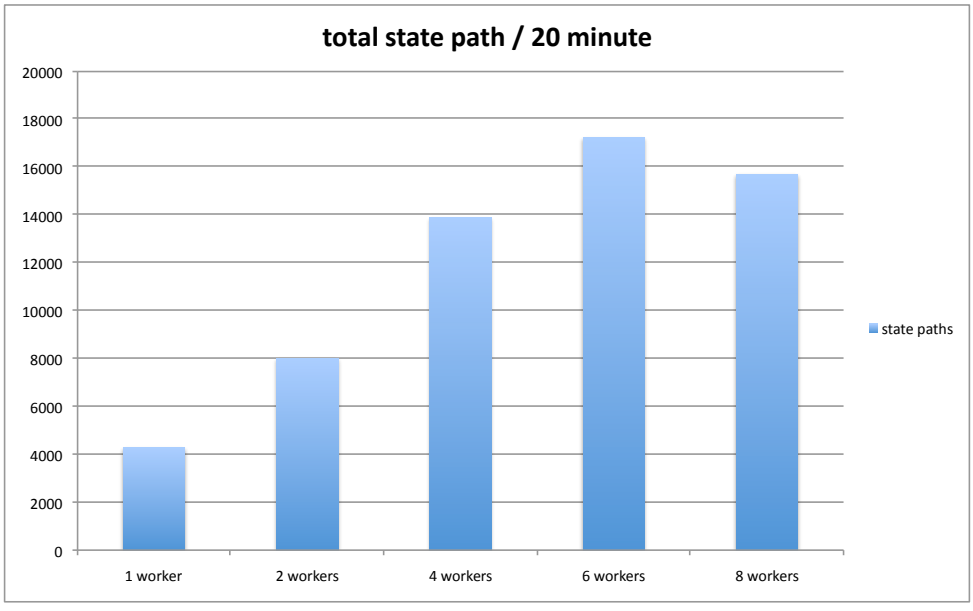


Figure 5.7: *ArrayPartition.partition() total state paths explored over a 20 minute span*

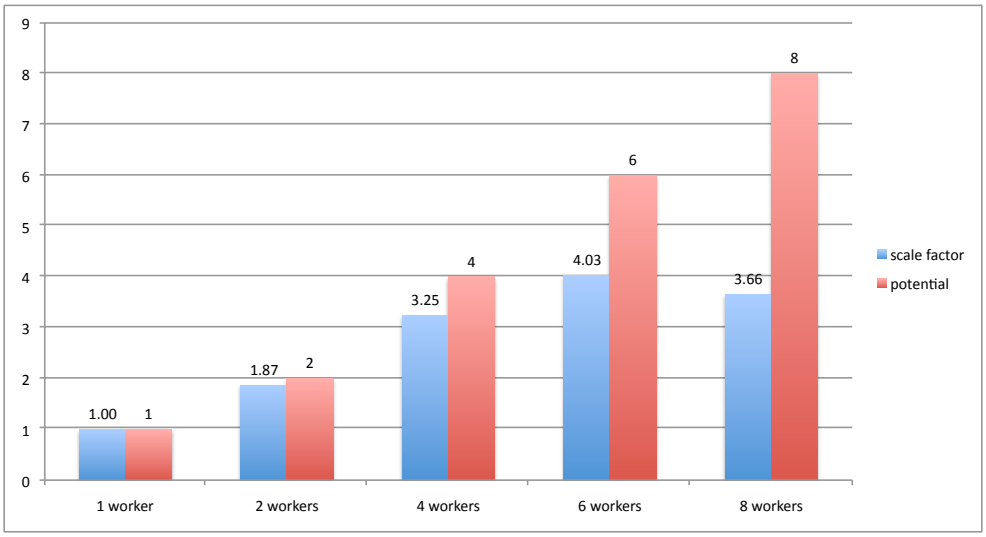


Figure 5.8: *ArrayPartition.partition() scale factor over a 20 minute window*

The ArrayPartition tests show good scaling as the number of workers are increased. Figures 5.9, 5.10, and 5.11 show why the scaling is so good. All parallelizations occured within the first minute of the analysis regardless of the number of workers. This means that at least 8 of the work-units assigned during the first minute of the analysis were computationally intensive enough that the workers could not complete them in under 20 minutes.

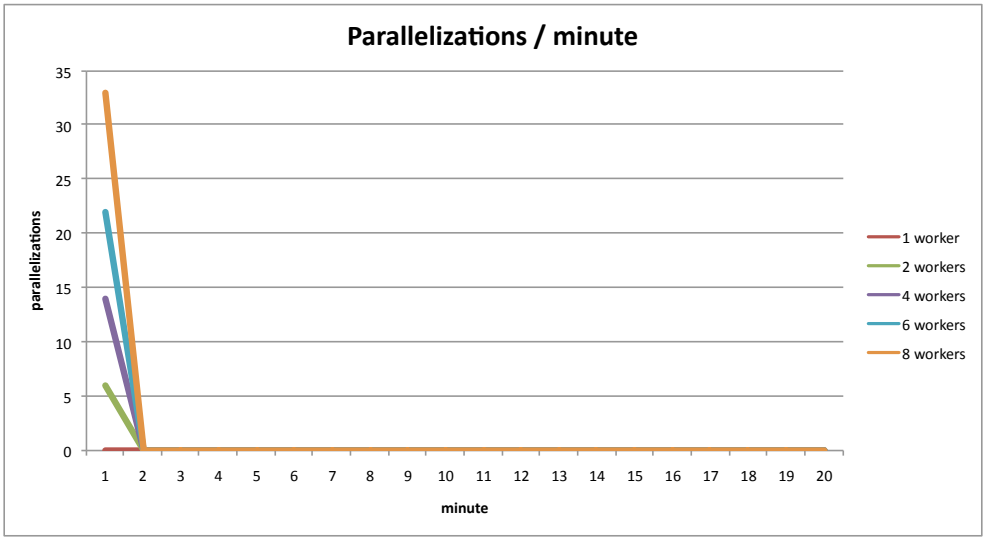


Figure 5.9: *ArrayPartition.partition() parallelizations per minute over a 20 minute span*

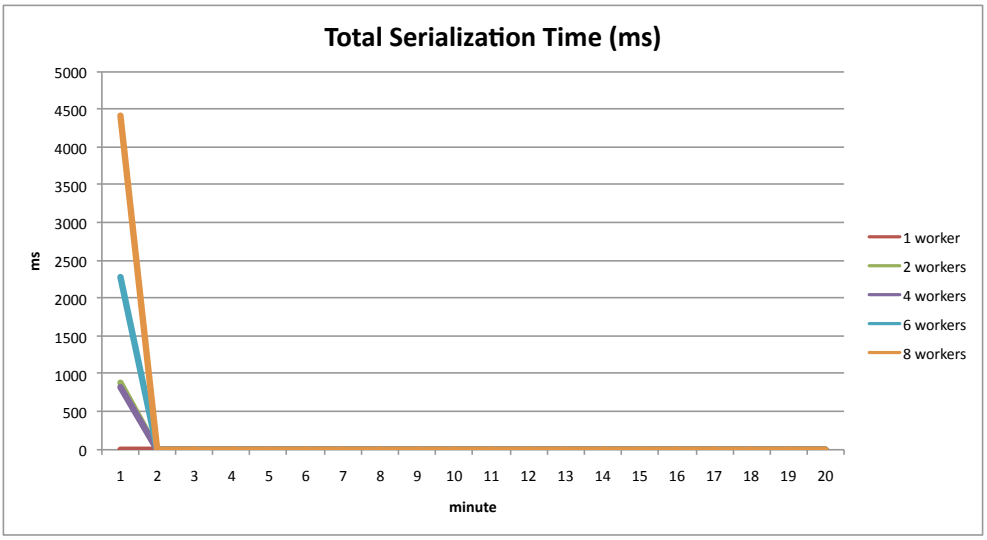


Figure 5.10: *ArrayPartition.partition() time spent serializating/deserializing workunits per minute over a 20 minute span*

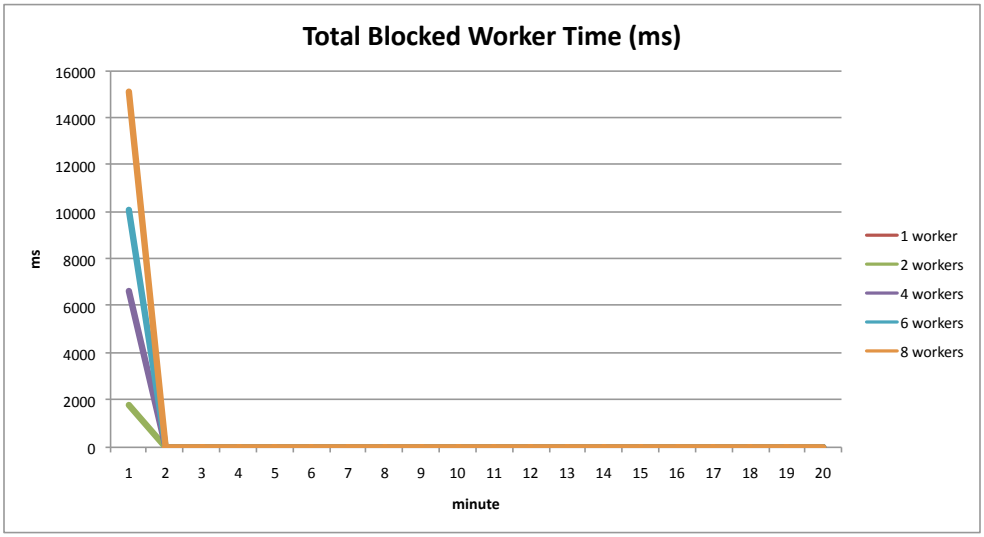


Figure 5.11: *ArrayPartition.partition() total time workers spent idle per minute over a 20 minute period*

5.6.2 BinaryHeap.findMin(), k=7, ab=100

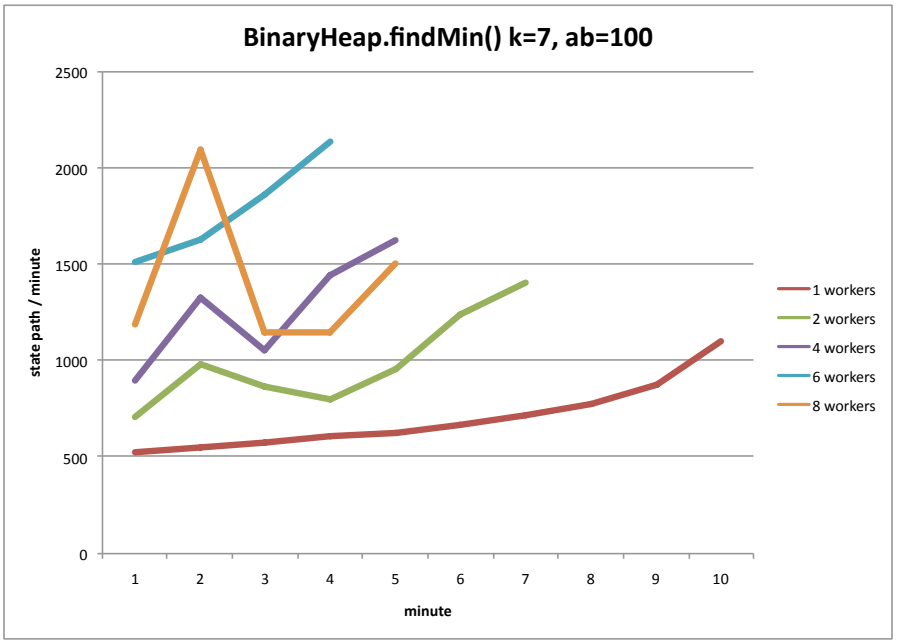


Figure 5.12: *BinaryHeap.findMin() state-paths explored per minute*

The BinaryHeap experiments are of interest because they show a unit which is both parallizable and can be analyzed in under 20 minutes even with aggressive bound settings. In figure 5.12 we see that increasing the number of workers will decrease the analysis time. Like with the ArrayPartition experiment 6 workers gave the best benefit on this computer.

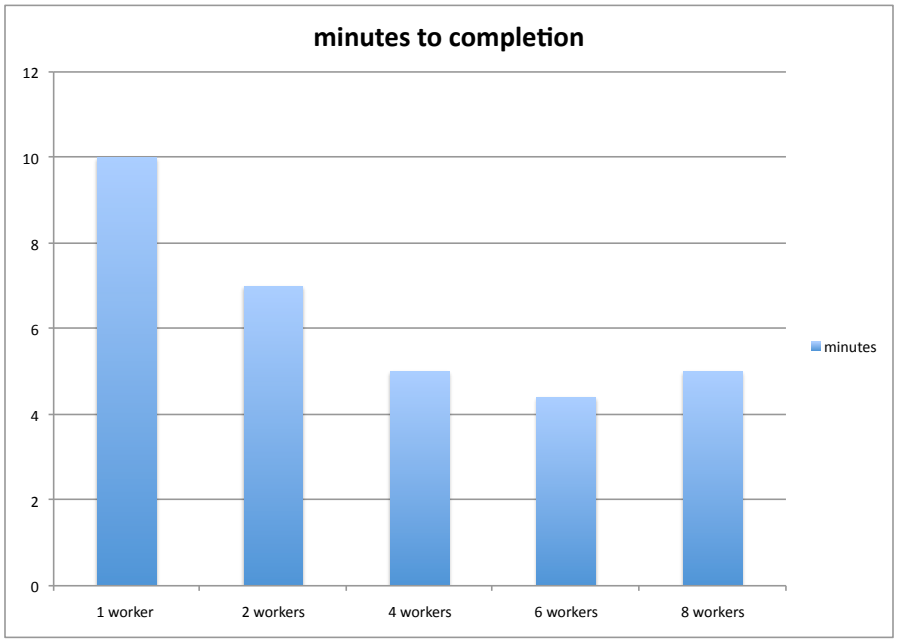


Figure 5.13: *BinaryHeap.findMin() time to finish complete analysis in minutes*

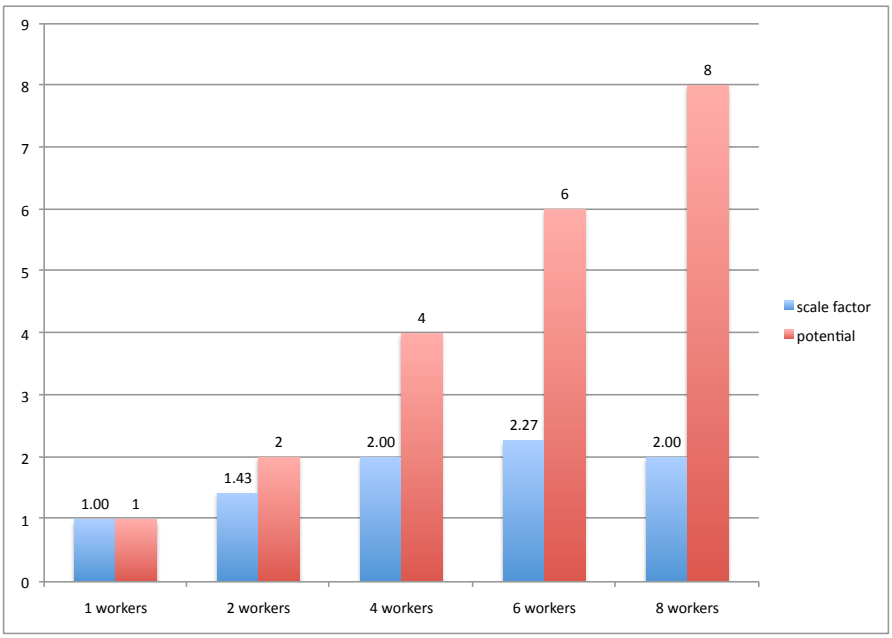


Figure 5.14: *BinaryHeap.findMin() scale factor derived from time to complete analysis*

5.6.3 BinarySearchTree.insert(), k=7

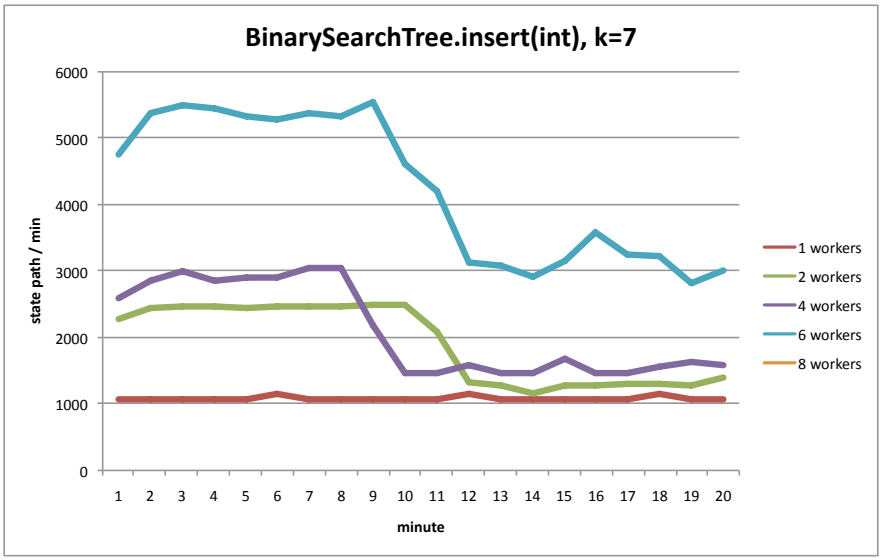


Figure 5.15: *BinarySearchTree.partition() state-paths explored per minute*

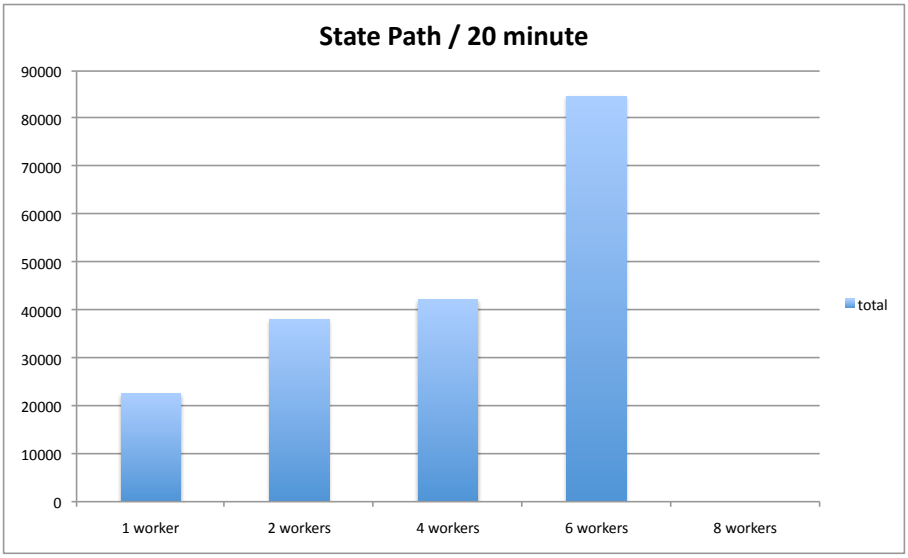


Figure 5.16: *BinarySearchTree.insert() total state paths explored over a 20 minute span*

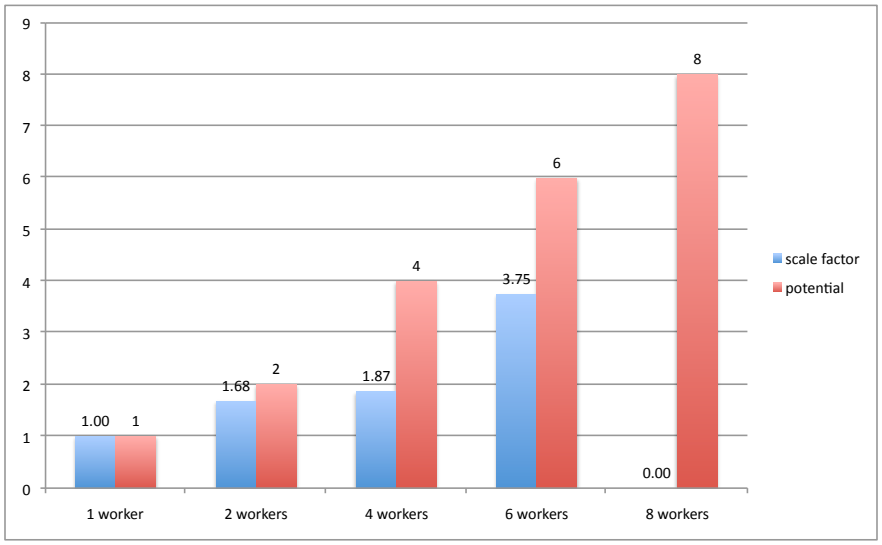


Figure 5.17: *BinarySearchTree.insert() scale factor over a 20 minute window*

5.6.4 DisjSet.union() k=7, ab=100

Figure 5.18: *DisjSet.union() state-paths explored per minute*

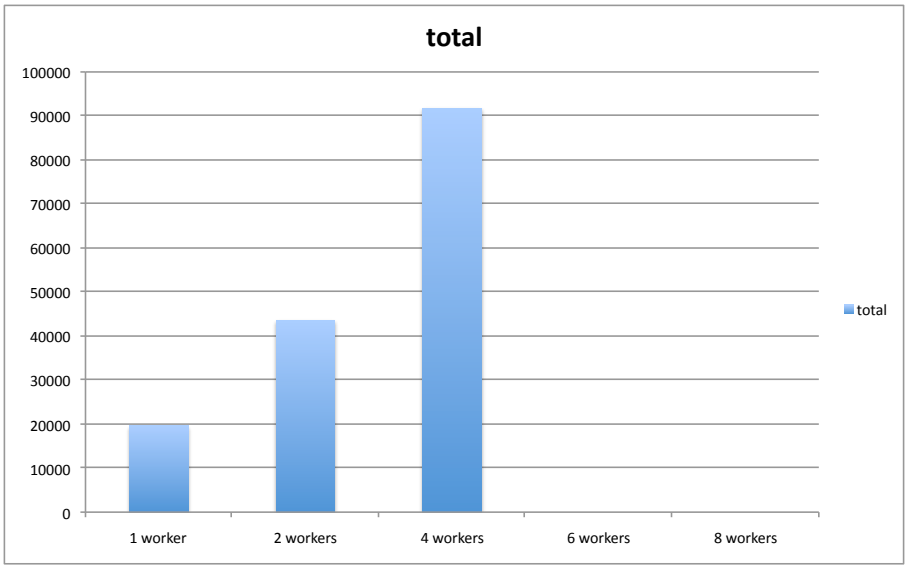


Figure 5.19: *DisjSet.union() total state paths explored over a 20 minute span*

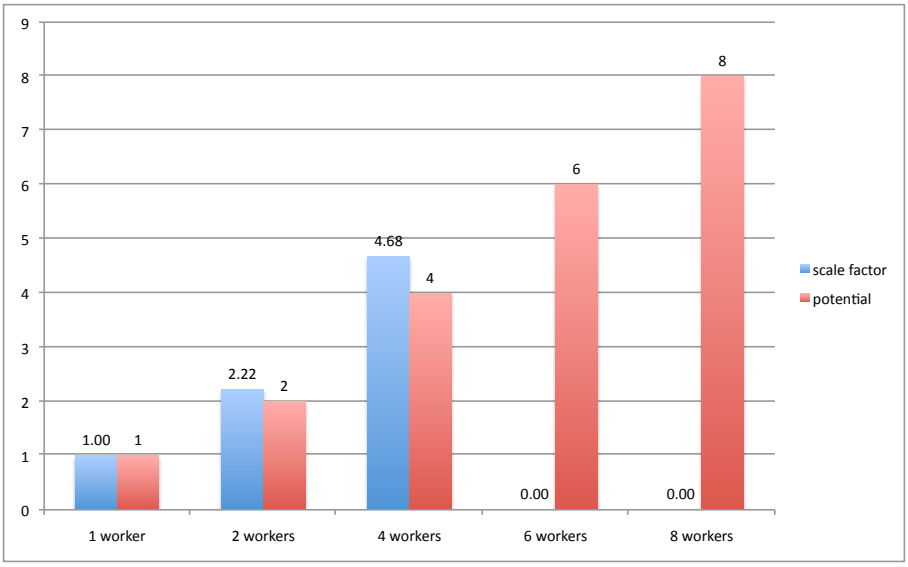


Figure 5.20: *DisjSet.union() scale factor over a 20 minute window*

In the DisjointSet.union experiments the addition of workers somehow resulted in more throughput than the performance target! How could this happen? The answer is that all execution sub-trees are not equal. Some sub-trees are just shorter than others. Some sub-

trees produce theorem prover queries which are easier than others. In DisjointSet.union() it is likely that sub-trees farmed out to the additional workers could be processed (i.e. reach the end of the paths) faster than the first choice, yielding more throughput than might be naively expected.

5.6.5 GC.mark(), k=4

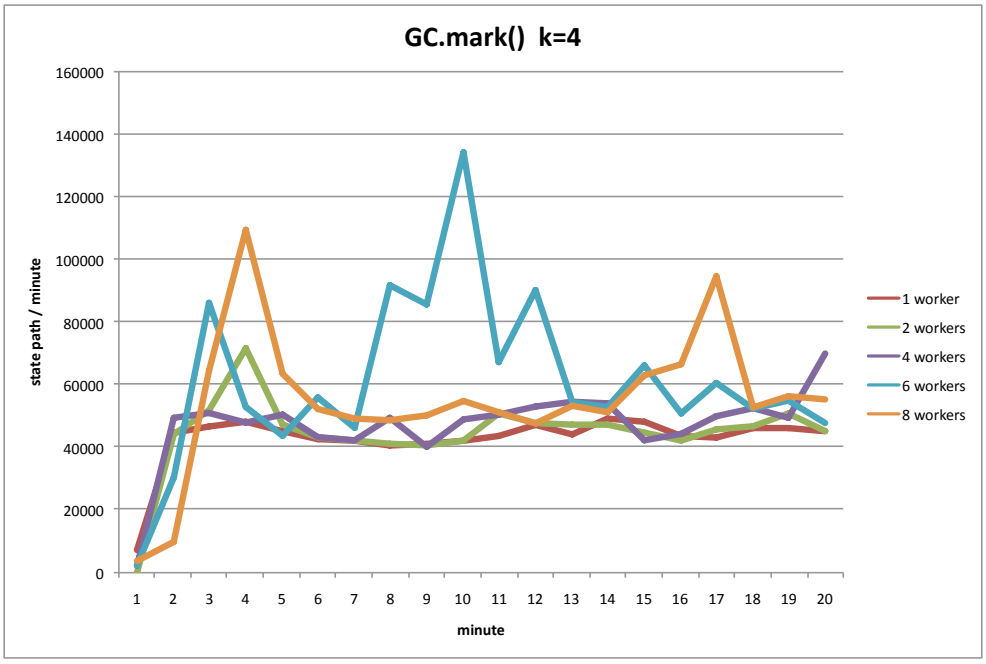


Figure 5.21: *GC.mark() state-paths explored per minute*

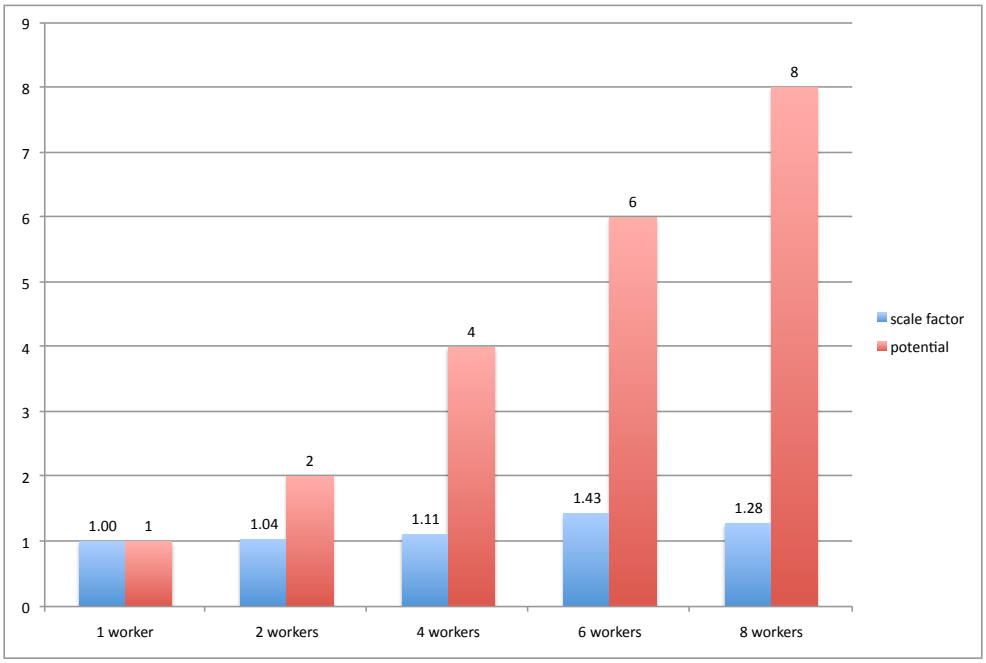


Figure 5.22: *GC.mark() scale factor over a 20 minute window*

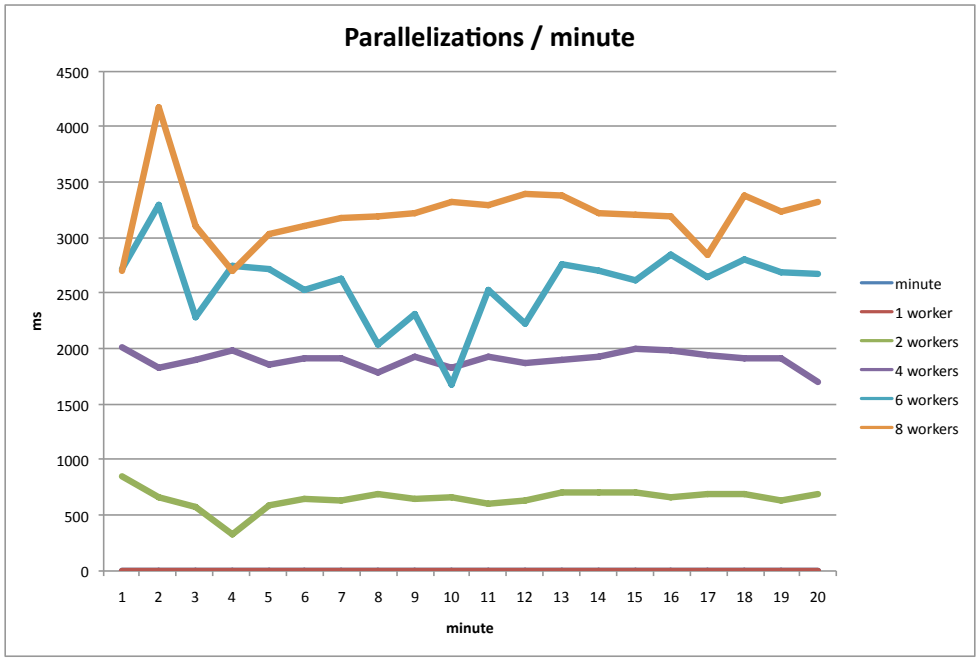


Figure 5.23: *GC.mark() parallelizations per minute over a 20 minute span*

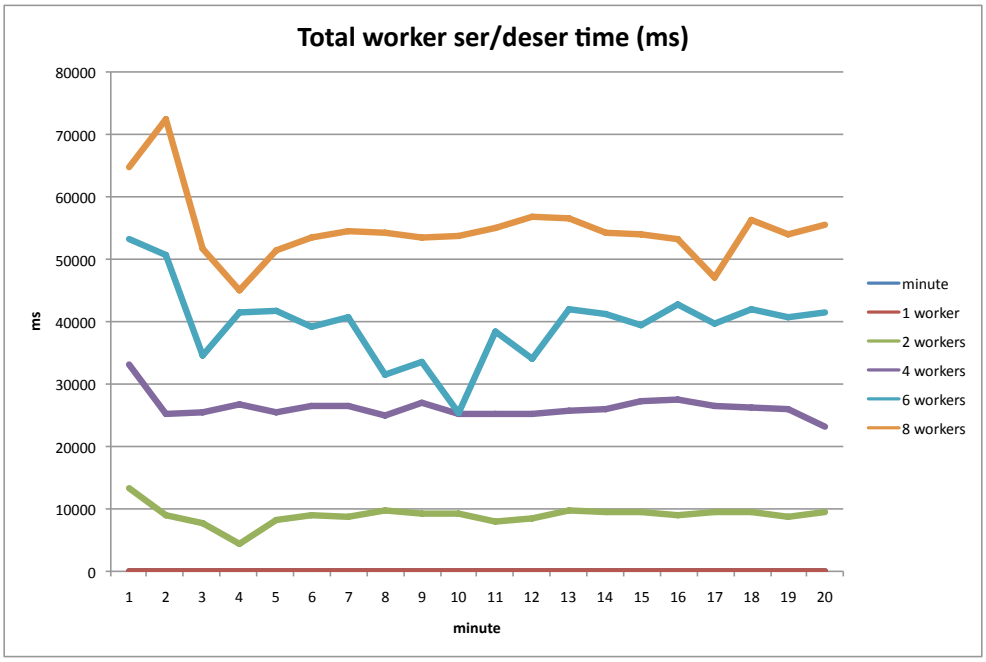


Figure 5.24: *GC.mark() time spend serializating/deserializing workunits per minute over a 20 minute span*

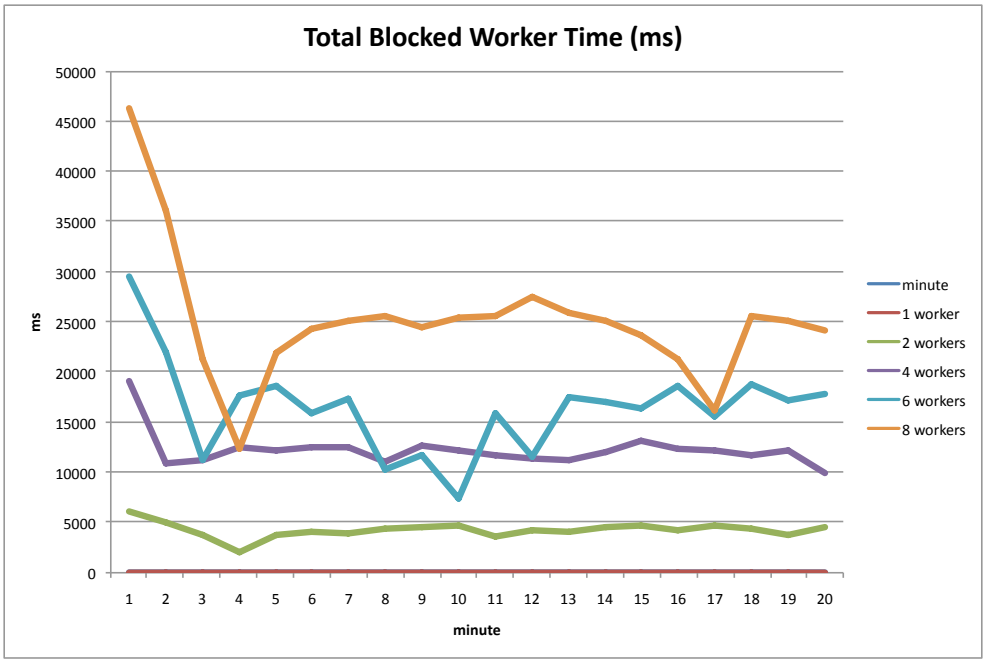


Figure 5.25: *GC.mark() total time workers spent idle per minute over a 20 minute period*

5.6.6 AvlTree.insert() k=7

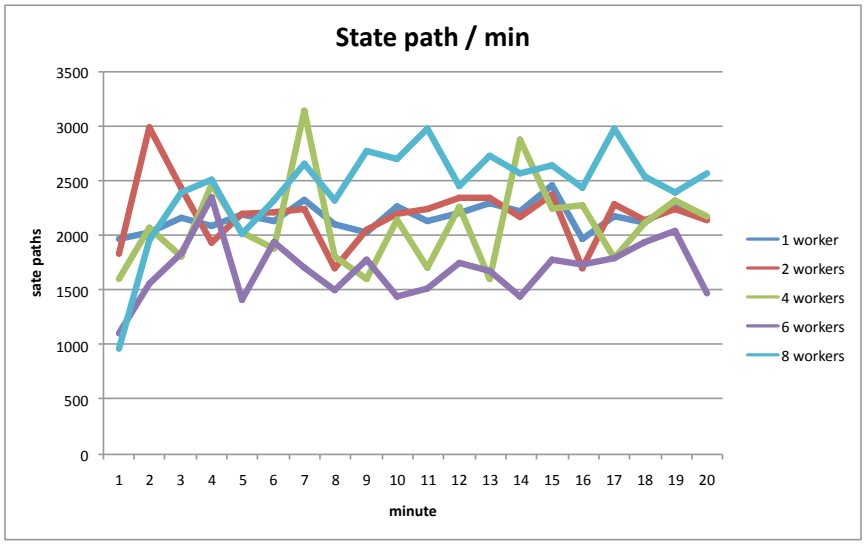


Figure 5.26: *AvlTree.insert() state-paths explored per minute*

Both the AvlTree.insert (figure 5.26) and GC.mark experiments demonstrate very poor scaling as the number of workers are increased. For both experiments there was a large amount of coordination activity taking place over the whole 20 minute period. While the GC.mark experimet did show some miniscule improvement with more than one worker operating in parallel, there was no discernible benefit from parallelization. Certainly, the execution tree for a particular analysis defines the amount of potential parallelism present. However, even highly parallel execution trees may have many (even a majority) of choices for which the parallelism is very poor. If this is the case, as lazy parallelization generates work units along the fringe of the execution tree, the workers assigned to those work units will finish quickly, become idle, and request more work units. If this process continues, then a sort of thrashing is happening and an inordinate amount of time will be spent on worker coordination activities. Section 8.0.1 I explain this problem in more detail and offer a possible solution.

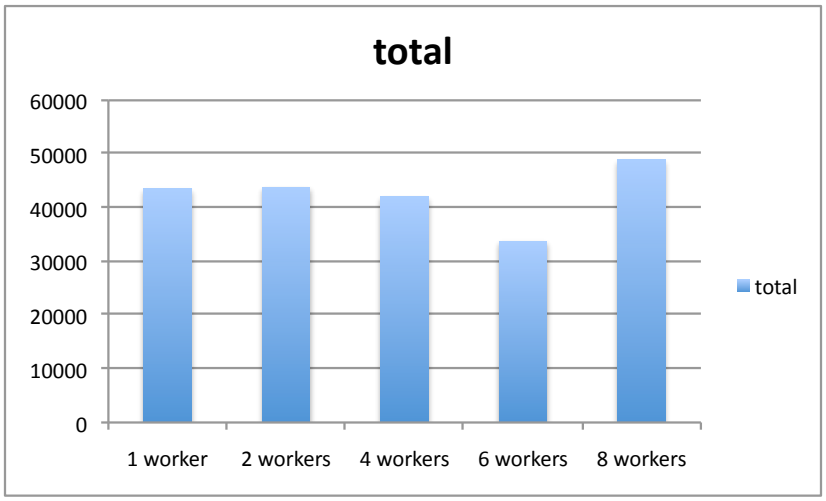


Figure 5.27: *AvlTree.insert() total state paths explored over a 20 minute span*

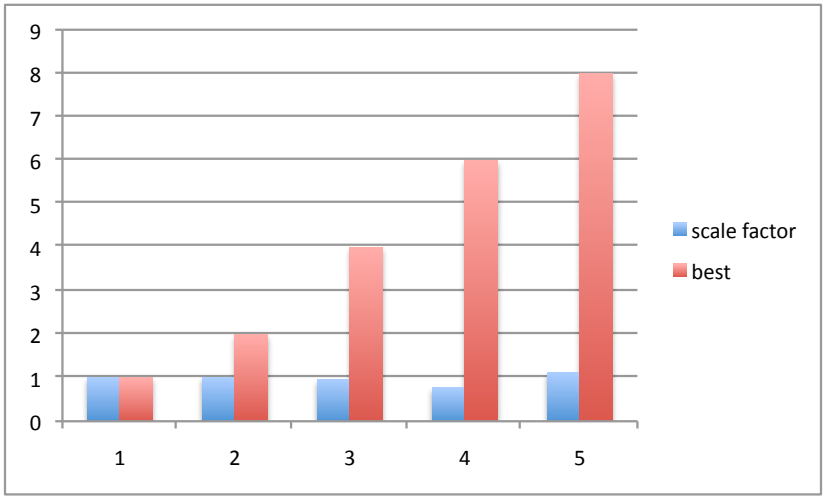


Figure 5.28: *AvlTree.insert() scale factor over a 20 minute window*

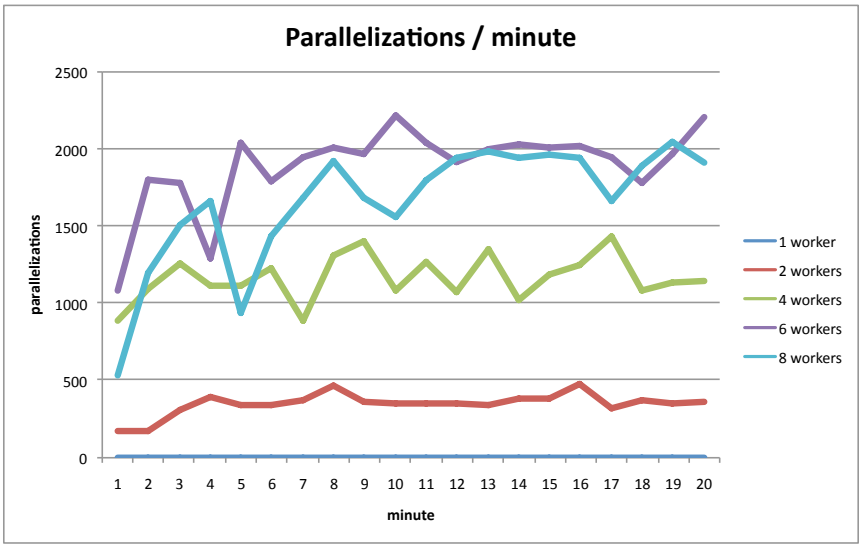


Figure 5.29: *AvlTree.insert() parallelizations per minute over a 20 minute span*

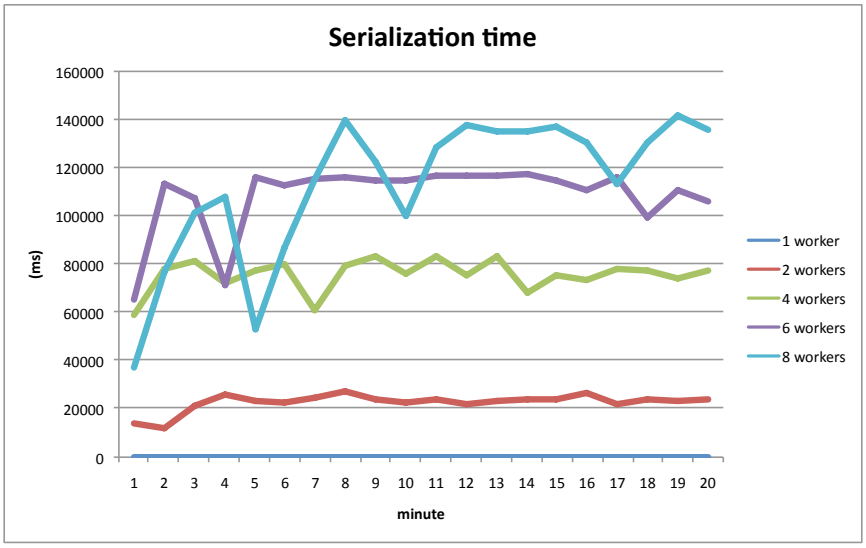


Figure 5.30: *AvlTree.insert() time spent serializating/deserializing workunits per minute over a 20 minute span*

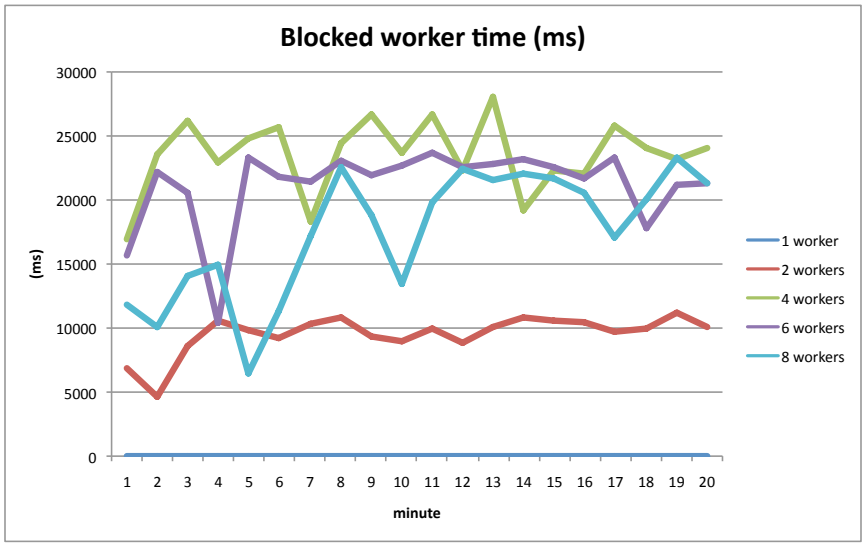


Figure 5.31: *AvlTree.insert() total time workers spent idle per minute over a 20 minute period*

Chapter 6

Related Work

6.1 ESC/Java

```
1: class Bag{
2:       int size;
3:       int[] elements;
4:
5:       Bag(int[] input){
6:               size = input.length;
7:        elements = new int[size];
8:               System.arraycopy(input, 0, elements, 0, size);
9:       }
10:
11:      int extractMin(){
13:              int min = Integer.MAX_VALUE;
14:              int minIndex = 0;
15:              for(int i = 0; i < size; i++){
16:                      if(elements[i] < min){
17:                              min  = elements[i];
18:                              minIndex = i;
19:                      }
20:              }
21:              size--;
22:              elements[minIndex] = elements[size];
23:              return min;
24:      }
25:}
```

Listing 6.1: *Non-annotated Bag implementation in Java*

ESC/Java ('Extended Static Checker for Java') is a static analysis tool originally developed at the Compaq Systems Research Center (SRC). ESC/Java is designed to be both lightweight and usable. ESC/Java is lightweight in that it does not endeavor to apply some of the more expensive analysis algorithms described later, but instead opts to uncover a certain class of less subtle programming errors with a minimum of computing resource investment. In this way ESC/Java does not provide either sound or complete(2.2) analysis (false errors may be reported, and the tool cannot detect all types of programming errors). ESC/Java's designers have engineered the tool towards usability by flexibly supporting lightweight program annotations (users are not required to code difficult to reason about predicate logic style annotations).

ESC/Java operates as a pipeline similar to how a compiler tool-chain works, but instead produces error warnings and error traces as opposed to object code. This pipeline takes annotated Java source code as its input, compiles the source code into a guarded command language *GC*s (based off of Dijkstra's guarded commands). Refer to listing 6.2.

```
S :: E //an expression
      S1;S2 //statement S2 executes immediately following S1
      S1 || S2 //statement S1 or S2 executes
      assume(E) //blocks until boolean E is true
```

Listing 6.2: *Simple guarded command language grammar*

the *GC*s are then used to generate verification conditions (known as *VC*s) via weakest precondition analysis , and finally a theorem prover (in this case Simplify) is used to determine if any of the verification conditions could violated. Because ESC/Java maintains a relationship between source code locations and particular *VC*s, if a *VC* can be violated then the user will be presented with a message describing what part of the program is problematic.

Weakest precondition calculation is a technique for lifting logical formulas directly from the semantics of a computer program. For a given program, its *weakest precondition* (*wp*)

is the weakest predicate P over the program's inputs (both state and parameters) that guarantees some predicate Q is *true* after P has finished. Following are basic weakest precondition rules for a simple language[15], where S and T represent statements like those made in program 2.1:

1. $wp\,(x = E) \Rightarrow Q[x = E]$
2. $wp\,(S;T) \Rightarrow wp\,(S, wp\,(T, Q))$, where S and T are both program statements and T is executed immediately after S is finished.
3. $wp\,(S||T) \Rightarrow wp\,(S) \wedge wp\,(T)$, where S and T are both statements, and either S or T could execute.
4. $wp\,(assumeE, Q) \Rightarrow (E \Rightarrow Q)$

The above rules can now be used to compute the weakest precondition for the program in listing 2.1. What would the weakest precondition be for $z! = 0$?

1. Unfold the expression $wp\,(p1, z! = 0)$ according to the above rules into $wp\,(z := x - y, z! = 0) \wedge wp\,(z := x + y, z! = 0)$
2. Compute the weakest precondition of the program statement from the *true* branch: $wp\,(z := x + y, z! = 0) = x + y! = 0$
3. Compute the weakest precondition of the program statement from the *false* branch: $wp\,(z := x - y, z! = 0) = x - y! = 0$
4. Since there is a guard on the branch, the last rule above needs to be used on each branch.
5. Combine the predicates for the weakest precondition of p1: $x \leq y \Rightarrow x - y! = 0 \wedge x > y \Rightarrow x + y! = 0$

Therefore any assignment of x and y that satisfies the predicate $x - y! = 0 \wedge x + y! = 0$ will result in a program execution of p1 that satisfies $Q = z! = 0$.

For an example of the types of errors ESC/Java can uncover, consider the Java Bag class listing 6.1. ESC/Java will detect that there are several places were a null pointer dereference may occur (lines 6,16, 17, 22) and array index out of bounds exceptions (lines 16, 17, 22). Clearly a null pointer dereference could occur at line 6 (a programmer could pass a null pointer to the constructor) but it looks like an array out of bounds exception would not. If the programmer annotates the source code (refer to listing 6.3),

```
1: class Bag{
1a: //@invarient 0 <= size && size <= elements.length
2:       int size;
3:       int[] elements;
4:       //@requires input != null
5:       Bag(int[] input){
6:               size = input.length;
7:        elements = new int[size];
8:               System.arraycopy(input, 0, elements, 0, size);
9:       }
10:
11:      int extractMin(){
13:              int min = Integer.MAX_VALUE;
14:              int minIndex = 0;
15:              for(int i = 0; i < size; i++){
16:                      if(elements[i] < min){
17:                              min   = elements[i];
18:                              minIndex = i;
19:                      }
20:              }
21:              size--;
22:              elements[minIndex] = elements[size];
23:              return min;
24:      }
25:}
```

Listing 6.3: *Bag example with JML annotations*

(note lines 1a and 4), Then some of the warnings that ESC/Java produced before (the

possible null pointer dereference error and the case when size is negative) are no longer produced when ESC/Java analyzes the annotated code.

As stated earlier, as analysis speed was of prime import during the design of ESC/Java, ESC/Java is intentionally un-sound and incomplete. In addition to not handling strong properties with respect to the heap (e.g. heap graph acyclicy), ESC/Java also makes some concessions in other areas. ESC/Java does not model integer overflow (in order to avoid spurious warnings) and complete semantics for loops. Precise semantics for loops (weakest fixpoints) are uncomputable in general and uncomputable in many cases that would occur during the analysis of common software. Thus, there is not a known easy way to convert a loop into a VC or set of VC. Instead ESC/Java unrolls the loop some user specified number of times and then performs analysis on the unrolled version. Any errors that may have occurred in a loop iteration past the number of unrollings will be missed.

6.2 Parallel ESC

The latest version of ESC, ESC4 supports parallelization of ESC's analysis[2]. When analyzing a program unit, the majority of ESC's computation is spent discharging the VCs. Parallelization of ESC's analysis involves the discharging of independent VCs concurrently. The designers of ESC4 target 3 different levels of parallelism:

1. Program Unit Parallelism - VCs from different program units are guranteed independent from one another and thus can be discharged concurrently.

2. Method Parallelism - VCs from different Java methods are guranteed independent from one another and thus can be discharged concurrently.

3. Sub-VC Parallelism - Sometimes a VC can be factored into a set of smaller sub-VCs. Each sub-VC represents a single path from a method's precondition to some assertion. For example, given some VC: $A \wedge B$ the sub-VCs would be A and B. A sub-VC is somewhat analogous to the path conditions from symbolic execution.

ESC4 is implemented as an Eclipse plugin that leverages the JDT compiler framework. Parallelism types 1 & 2 are handled by simply using the JDT framework to apply a different thread to each unit or method depending on available resources. For type 3, ESC4 offers two deployment possibilities: Factor the macro VCs locally into sub-VCs and then attempt to discharge those in parallel or offload the macro VC to some service which will then factor the VC and attempt to discharge the sub-VCs in parallel. Figure 6.1 contains the results the authors of ESC4 have presented. The time for the analysis reduces as the number cores are increased. Like some units analyzed under parallel Kiasan, the analysis doesn't exhibit perfect parallelism as the time to complete analysis isn't $1/numcores$.

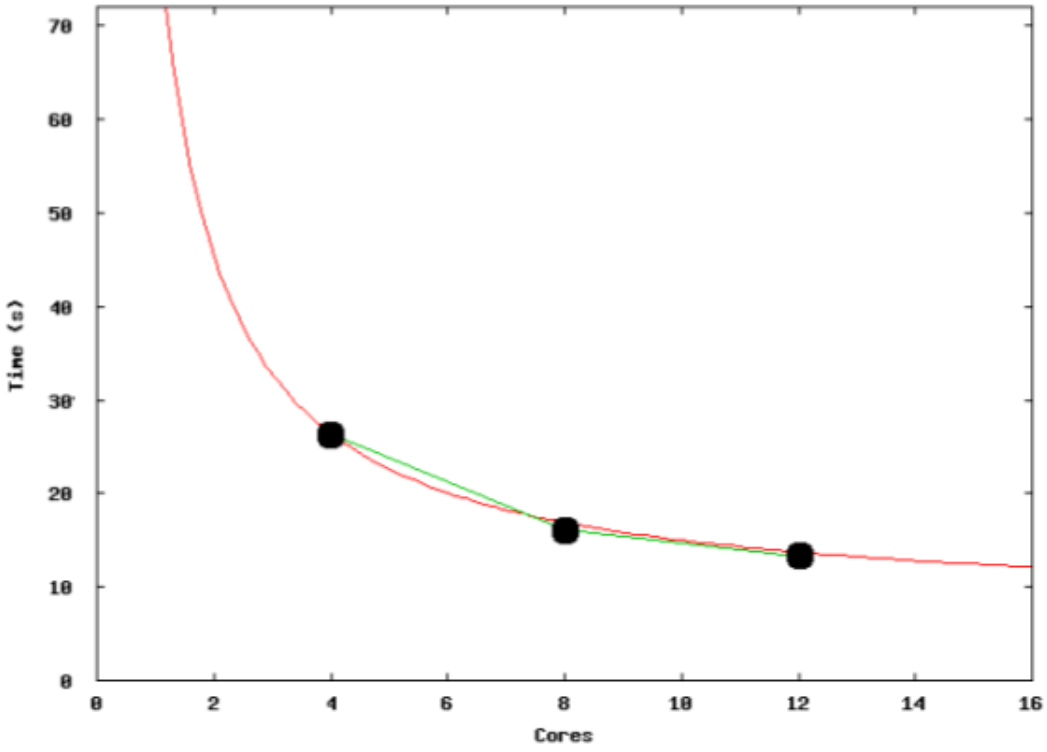


Figure 6.1: *Parallel ESC Time(seconds) vs. Number of Cores*[2]

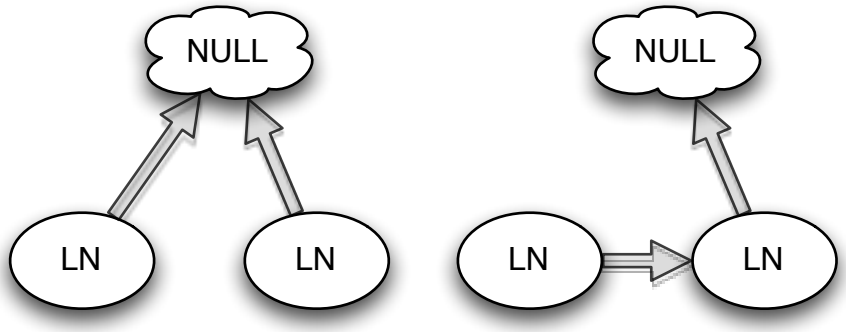
6.3 JPF

Java Path Finder (JPF) is a comprehensive model checker built at NASA Ames. JPF is used to search the concrete state space of a Java program for errors or specified property violations. JPF includes what is now considered the defacto standard set of state space search optimizations: thread symmetric reductions, collapse compression, partial order reduction, etc. JPF has been leveraged by Khurshid et al[9] to perform symbolic execution by way of instrumented code. Khurshid et al[9] has built a system comprised of the code instrumentor and JPF. Because the instrumented code contains the symbolic semantics of that program, a symbolic execution analysis of the program is performed when the instrumented code is run through an explicit state model checker like JPF. As of the writing of this thesis, symbolic execution on JPF only supports lazy initialization.

6.4 Structure Analysis for Testing

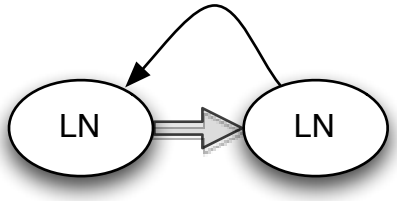
Structure analysis for testing is a technique that aids in the creation of unit test cases. Given some complex type definition (e.g., a Java type), structure analysis will automatically generate object graphs according to that type specification up to a certain bound. Sometimes a *representation predicate* is used to prune test cases from the test corpus that are not relevant to the unit's functional behavior (i.e. the test case in question does not conform to the units pre-condition).

Consider the *LinkedList* from listing 2.2. Based on *LinkedNode*'s type specification, there are 3 possible non-isomorphic *LinkedList* structures with 2 *LinkedNode*s (assuming the generation does not allow self-cycles) Refer to figure 6.2. If the representation predicate returns *false* when there is a cycle 6.2(c) would be pruned from the test corpus.



(a) next for both nodes is NULL

(b) One node points to the other, the last points to NULL



(c) Nodes are in a cycle

Figure 6.2: *Three structures generated by a structure analysis on LinkedNode*

6.5 Korat

Korat is another software system designed to automate the testing of Java programs[16]. Korat is primarily geared towards generating test input of the complex (heap-object) parameters of Java units. Korat uses a form of structure analysis to automatically generate all non-isomorphic relevant heap-graphs up to a certain user provided size when a Java *representation predicate* (Java method that returns *true* when the heap graph is structurally sound, false otherwise) and JML annotations are available. The predicate is used to prune the space of possible heap graph permutations so only heap graphs which satisfy a units preconditions are considered during unit testing. Once the heap configurations that satisfy the unit precondition are generated, the unit is tested concretely with those configurations as inputs. Standard testing harnesses then capture any errors that are uncovered by testing. To my knowledge, Korat only generates structural test inputs (i.e. it does not *automatically* give any regard to primitive typed object fields.)

6.5.1 Korat - Walkthrough

Listing 6.4: *BinarySearchTree and its representation predicate repOK*

```
class BinarySearchTree{
        private Node root;
        private int size;
        static class Node{
                private Node left;
                private Node right
                public int val;
        }
        .. ..
        public void insert(int i){.. ..}
        .. ..
        public boolean repOk(){
                if (root == null) return size == 0;
                Set visited = new HashSet();
                visited.add(root);
                LinkedList workList = new LinkedList();
                workList.add(root);
                while (!workList.isEmpty()) {
                Node current = (Node)workList.removeFirst();
                if (current.left != null) {
                        // checks that tree has no cycle
                        if (!visited.add(current.left))
                        return false;
                workList.add(current.left);
        }
        if (current.right != null) {
                // checks that tree has no cycle
                if (!visited.add(current.right))
                 return false;
                workList.add(current.right);
        }
        }
        // checks that size is consistent
        if (visited.size() != size) return false;
        return true;
        }
}
```

A simple data-structure to consider is the binary search tree whose Java source code is listed above (adapted from the BinaryTree example in Korat: Automated Testing Based on Java Predicates[16]). The above example contains the Java type specification of *BinarySearchTree* and *Node* as Java classes. Also included is the *repOk()* method, which is the Java predicate used in the precondition expressions.

Korat provides what its authors call a 'finitization' framework which is responsible for generating all possible heap configurations of objects that are of the appropriate type (in this case objects of type BinarySearchTree and Node.) From:[16]

Listing 6.5: *Finitization for BinarySearchTree*

```
public static Finitization finBinarySearchTree(int NUM_Node, int
    MIN_size, int MAX_size) {
        Finitization f = new Finitization(BinarySearchTree.class);
        ObjSet nodes = f.createObjects("Node", NUM_Node);
        nodes.add(null);
        f.set("root", nodes);
        f.set("Node.left", nodes);
        f.set("Node.right", nodes);
        return f;
}
```

Korat's finitization framework reflects on the class definition for the unit in question and emits a Java method called a finitization skeleton. Above is the finitization skeleton for the BinarySearchTree. This emitted finitization can either be specialized (such as adding bounds for the primitive type fields to range over.)

Korat will now perform a backtracking search over all possible configurations defined by the finitization, using the predicate to prune all parameter input candidates from the set that will be used in testing. The backtracking search works in the following way: A given finitization defines an ordering on class domains and field domains. A class domain is simply the set of all objects in the finitization belonging to a certain class. If, for instance, the finitization listed above was created with `MAX_size = 3`, then the class domain for *Node*

would contain N_0, N_1 and N_2. A field domain is simply the *ordered union* of all the class domains.

Candidate inputs are encoded as vectors, the length of which is equal to the size of the field domain, and whose elements are indexes into their respective class domains. For instance, the following heap state:

```
BinarySearchTree this;
Node N0, N1, N2;
//
root == N0;
N0.left == N1;
N0.right == N2;
N1.left == null;
N1.right == null;
N2.left == null;
N2.right == null;
}
```

would be encoded as *1,2,3,0,0,0,0* because index 1 in the *Node* class domain is the *Node* instance N_0. (null is the 0^{th} index for all class domains.) The next six elements of the vector index what members of the *Node* class domain are to be used for the N_0, N_1, and N_2 *left* and *right* fields.

Korat explores the space of input vectors by systematically incrementing (up to the size of the relevant class domain) the indexes in each element of the input vector and testing the heap configuration denoted by that vector against the Java predicate. If the predicate returns *true* then that heap configuration is stored for use as a test configuration. If the predicate returns *false* then Korat backtracks to the last element incremented and increments it again. If no increment is possible, Korat backtracks yet again until it finds the most recently incremented element that can still be incremented. In this way, Korat is able used the Java predicate as a guide for pruning irrelevant test inputs.

6.5.2 Distributed Korat

The authors of Korat have been able to extend Korat to take advantage of Google's MapReduce framework and distribute both the test case generation process and the testing phase across a large cluster of machines.[17] The authors describe 4 approaches they used to parallelize and distribute some aspect of the Korat pipeline[17]. These approaches are named *SEQ-OFF*, *SEQ-ON*, *PAR-OFF*, and *PAR-ON*.

In *SEQ-OFF* Korat is used in the manner described in the previous section to sequentially generate test inputs for the unit. These test inputs are stored to disk, and then distributed evenly by the MapReduce implementation to nodes in the cluster.

In *SEQ-ON* the first time Korat is every instructed to generate test inputs it does so sequentially, and testing with the inputs happens as it does in *SEQ-OFF*. If at some point in the future a user wishes to generate test inputs again, *SEQ-ON* will use information that was stored in the initial sequential test generation to aid in load balancing the current parallel test generation across nodes in the cluster.

In this example load balancing means that each node in the cluster should be assigned some start candidate vector that is 'equidistant' from candidate vectors assigned to other nodes, which means that each node explores approximately the same number of candidates. *SEQ-ON* records candidates that are equidistant in the first run and uses those in subsequent runs to 'seed' explorations of the candidate space.

In *PAR-OFF* parallelizes all test case generation runs. Because it is not know a priori the number of test cases to generate (the Java predicate is used to prune test cases from the set of test candidates, and the test cases it will prune is not cheaply determinable) *PAR-OFF* attempts to find equidistant candidate vectors by randomly generating them.

PAR-ON is a combination of *SEQ-ON* and *PAR-OFF*; The initial run of test inputs is generated in parallel in the manner of *PAR-OFF*, except that true equidistant candidates are recorded. Then test cases are tested as in *PAR-OFF* and *SEQ-ON*. If there are any subsequent test case generation runs, then the equidistant vectors from the first run are

used as node seeds.

Misailovic, et al. shows how a speed-up ratio of approximately 544 was achieved when test generation was performed using 1024 workers *PAR-OFF* for a generic graph structure whose Java predicate required that the graph be acyclic[17].

6.6 Concolic Testing

Concolic testing is another modification of symbolic execution for use with heap graphs developed by Koushek Sen[18]. Concolic testing hopes to avoid some of the computational tractability issues that can arise when symbolic execution is generating constraints over a heap graph. For instance, when testing some real world software it can be the case that some constraints generated by standard symbolic execution may be too complex for current theorem provers to efficiently handle. In order to avoid using a theorem prover to primarily drive the analysis concolic testing instead merges aspects of concrete testing and symbolic execution.

In concolic testing either the user supplies some input parameters for a unit or some input is randomly generated. These inputs are stored in some logical input map I. The program is then executed concretely with the input parameters stored in I. As the program executes, the concolic execution engine detects what program statements were involved in the concrete execution. The symbolic semantics are then applied to that particular program slice and a path condition PC is generated. The concolic algorithm then backtracks, and negates the PC. A constraint solver is then used to uncover a satisfying assignment of input parameters for some contradicition of PC. I is replaced with I', the logical input map containing the parameters that are a satisfying assignment to the contradiction of PC. This process of concretely executing the program with the parameters from I and then replacing I is repeated until some stop condition occurs.

For a brief example consider the program from listing 2.1. This program has 3 input parameters x, y, z. Randomly assign $x = 1014$, $y = 977$, and $z = -247$. Under concrete

execution with these parameters the program flows down the true branch and executes z := x + y. At this point, the PC for this execution will be generated, namely that $\alpha > \beta \wedge x = \alpha \wedge y = \beta \wedge z = \gamma \wedge \gamma = \beta + \alpha$. Concolic testing backtracks, negates $\alpha > \beta$ into $\alpha \leq \beta$ and uses a constraint solver to find a satisfying assignment. That satisfying assignment will be used in the subsequent concrete execution.

6.7 jCUTE

jCUTE is Sen et al's[19] implementation of concolic testing for Java programs. jCUTE operates in much the same way as JPF, in that it instruments the Java program with library calls. This library then manages the symbolic state of the symbolic execution. jCUTE combines concolic testing of sequential programs with explicit state model checking creating what Sen et al's[19] calls *explicit path* model checking. The combination of symbolic heap state and concrete thread schedules allows jCUTE to detect concurrency errors in addition to the more standard functional errors. jCUTE has been used to discover concurrency errors in supposedly thread safe classes that make up parts of the Java 1.4 runtime language.

6.8 Theoremprovers / SMT Solvers

The static analysis techniques discussed in this chapter all require some facility to determine if some logical assertion (such as int $x > 0$) is either *true* or *possible*. This facility is realized in a class of programs known as theoremprovers. Theoremprovers allow a client (a user or program) to make logical assertions. Then, the client can query the theoremprover with some logical statement. The theoremprover will then attempt to decide if the query is *true* or *possible* given the previous assertions. Since theoremprovers attempt at some level to decide an instance of SAT, in general theorem provers are in the complexity class NP-complete. Different theorem provers provide different optimizations for different types of logical assertions. Here I will give a very brief overview of two theorem provers used in Kiasan.

6.8.1 CVC3

CVC3 is an automatic theoremprover for the *Satisfiability Modulo Theories* (SMT) problem[20]. CVC3 is a descendant of the CVC and CVCLite theorem provers, which in turn descended from the SVC (Stanford Validity Checker). CVC3 provides 3 different interfaces to its checker; an input scripting language, an interpreter prompt for that language, and an API for programs written in C or C++. All 3 interfaces allow a client to assert various logical assumptions and then query the prover for the validity of logical formulas in the context of those assumptions.

CVC3 includes theory solvers for the following theories:

1. Abstract Data Types - CVC3 can reason about arbitrary recursive and mutually recursive data types. (Similar in expressive power to data types in a functional language like LISP)

2. Bitvectors - CVC3 has a decision procedure for reasoning over an arbitrary string of bits.

3. Quantifiers - CVC3 has a separate decision procedure optimized for dealing with quantified formulas.

6.8.2 Yices

Yices is a SMT solver developed at SRI[21]. Yices employees the Davis-Putname-Logemann-Loveland (DPLL) algorithm to determine the satisfiability of a first order logic formula. If the formula contains more complex expressions, then Yices will offload the expression to the appropriate theory solver module. The standard Yices distribution includes solvers for:

1. Arithmetic - Yices currently includes a solver for linear arithmetic. Non-linear expression will cause Yices to give up.

2. Bit Vectors

3. Arrays

4. Datatypes - Yices supports abstract datatypes, through currently this feature is not exposed in the programming API, only the interactive command shell and file input interface.

5. First Order Quantification

In this author's experience, yices is much faster deducing the satisfiability of the formulas symbolic execution generates than CVC3. Yices is closed source, so projects like Kiasan offer the open source CVC3 as an option.

6.9 JUnit

JUnit is a unit testing framework for the Java programming language. JUnit allows programmers to create test cases for Java program units (Java methods or classes.) The framework can then invoke those units with programmer specified mock inputs and then the framework will automatically compare the output of the unit with expected output. This makes JUnit particularly useful for regression testing.

Listing 6.6: *A Java unit that computes the average of numbers in an array*

```
public class Average{
        public static double computeAverage(double[] numArray){
                double accum = 0.0;
                for(int i = 0; i < numArray.length; i++){
                        accum += numArray[i];
                }
                return accum / numArray.length;
        }
}
```

Listing 6.7: *A test case that uses the JUnit framework to perform the test*

```
import org.junit.*;

public class AverageTest{
        @Test
        public void testAverage(){
                Assert.assertEquals('Average', 2.0, Average.
                   computeAverage({1.0, 2.0, 3.0}))
        }
}
```

Chapter 7

Conclusion

This thesis described both why symbolic execution can be parallelized and how that could be done. Over the course of this research two major versions of parallel symbolic execution were implemented; Once on Kiasan/Bogor, and currently on Kiasan/Sireum. These implementations allowed the exploration of the engineering issues concerning parallel symbolic execution and performance evaulations of parallel symbolic execution when it is applied to examples of real-world software units.

The experiments exposed scenarios where the current implementation of symbolic execution performed very well, scaling linearly as the number of workers were increased (e.g. BinarySearchTree.insert, section 5.2.1 and ArrayPartition.partition, section 5.6.1). The experiments also demonstrated that some units did not have very parallel execution trees, or the shape of those execution trees did not lend themselves to parallelization with the currently applied parallelization hueristics (e.g. ListStack.put, section 5.2.2 and AvlTree.insert, section 5.6.6). Overall the experiments indicate that in most cases parallel symbolic execution will provide a substantial speed-up on multi-core systems, which means that this approach could have immediate utility for software developers.

Chapter 8

Future Work

8.0.1 Root-closest Parallelization

As identified in chapter 7, a possible hindrance to distributed Kiasan's ability to scale is Kiasan's eager parallelization. As workers become idle, active workers will begin to generate work units at the point the active worker is at in the computation tree. This gives rise to the unfortunate possibility that newly idle workers will get assigned work units that only encompass a small fraction of the overall computation tree, and which therefore take a significantly smaller amount of time to compute.

Optimal parallelization occurs when each worker thread is assigned work units that take a long (relatively) time to compute. Optimal parallelization would require knowlege of the computation tree apriori so distributed Kiasan could choose the best locations to generate work units. Unfortunately the computation tree cannot be known *a priori* (as this is what the analysis seeks to uncover), however it may still be possible to change the distribution logic in such a way that would increase the *probability* that work units are generated from subtrees that are large (computationally.) See appendix A for a detailed description of how this approach could be implemented.

8.0.2 Parallel Report Generation

In non-parallel Kiasan, reports are generated from statistics the tool accumulated during analysis of a particular program unit. These statistics include generated test cases, byte-code

instruction coverage information, branch coverage information, and any potential errors the analysis detected. More formally, the reports contain:

1. T_c, a set of test cases.
2. I_c, the set of indices of instructions covered.
3. B_{rc}, the set of branches covered.
4. E, the set of errors detected.

Since parallel Kiasan could explore different sections of an execution tree independently, the reports generated from each independent exploration may not have correct information with respect to the total analysis. In addition, parallel Kiasan may be applied to very expensive (i.e. long running) analyses, so it would be helpful if Kiasan was able to accumulate report information using some soft of any-time algorithm that could also be easily parallelized.

What follows is a high level description of a simple any-time report statistics merging algorithm. Because the algorithm is any time, parallelization is trivial, it simply involves performing a union over all the sets that comprise the statistics:

Algorithm 17 Any time report merge algorithm

input($< T_c^1, I_c^1, B_{rc}^1, E^1 >, < T_c^2, I_c^2, B_{rc}^2, E^2 >$)
$T_c^{ret} \leftarrow T_c^1 \bigcup T_c^2$
$I_c^{ret} \leftarrow I_c^1 \bigcup I_c^2$
$B_c^{ret} \leftarrow B_c^1 \bigcup B_c^2$
$E_c^{ret} \leftarrow E_c^1 \bigcup E_c^2$
return $< T_c^{ret}, I_c^{ret}, B_{rc}^{ret}, E^{ret} >$

With the any time algorithm above, it is simple to imagine a 'Bag of Tasks' algorithm to merge a collection of reports in parallel, where the collection of initial tasks is configured at start up:

Algorithm 18 Parallel report merge worker process

```
while true do
  report1 ← getReportWU()
  report2 ← getReportWU()
  mergedReport ← merge(report1,report2)
  putReportWU(mergedReport)
end while
```

8.0.3 Kiasan on Map Reduce

Map Reduce is a parallel programming framework developed at Google to enable programmers to take advantage of massively parallel computer systems or clusters (see section 2.9.) Other static analysis techniques such as Korat (see section 6.5) have been adapted to the Map Reduce model and have subsequently taken advantage of the concurrent processing capability provided by thousands of processors[17] It would seem worthwhile to attempt to adapt Kiasan to the Map Reduce programming model in order to take advantage of the properties inherent to that platform.

However, it is not clear if the state-space exploration of Kiasan could be easily shoehorned into *Map* and *Reduce* phases in a way that would take advantage of the massive parallelism a large cluster could provide. Map Reduce shines for problems where the input data set is very large a known *a priori* (e.g. sequencing a genome or sorting Google's web page index). With symbolic execution the scope and breadth of the analysis is unknown before the analysis.

One way to take advantage of Map Reduce could be to iteratively apply a set of specialized *Map* and *Reduce* functions to grow the set of input data. In this manner, Map Reduce would provide a parallel breadth-first search (BFS) of the execution tree.

Algorithms 19, 20 and 21 together illustrate this pattern. During the first few phases of iteration, the MapReduce cluster would be poorly saturated because the width of the execution tree near the root would be relatively small. However the number of paths through a large analysis can be quite large (over 170 million for RedBlackTree.insert(), k=7) so there is a large workload available for parallelization.

Algorithm 19 Map function for emitting the successor states of a state

```
input (parentstate, state)
currentState ← state
while multipleChoices == 0 do
  currentState ← step(currentState)
  multipleChoices ← countChoices(currentState)
end while
for each state s in currentState do
  EmitIntermediate(currentState,s)
end for
```

Algorithm 20 Reduce function Map Reduce kiasan. Simply passes through input.

```
input (parentstate, list(states))
for each state in list(states) do
  Emit(state)
end for
```

Algorithm 21 Iterate MapReduce to explore a symbolic execution tree breadth-first

```
while |inputKeyPairSet| > 0 do
  invoke Map phase over inputKeyPairSet
  invoke Reduce phase
  inputKeyPairSet ← generateInputKeyPairs(reduceResults)
end while
```

Bibliography

[1] X. Deng, J. Lee, and Robby, Bogor/kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems, in *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 157–166, Washington, DC, USA, 2006, IEEE Computer Society.

[2] P. R. James, P. Chalin, L. Giannas, and G. Karabotsos, Distributed multi-threaded verification of java programs, in *Seventh International Workshop on Specification and Verification of Component-Based Systems*, 2008.

[3] S. Khurzid, C. Pasareanu, and W. Visser, Tools and Algorithms for Construction and Analysis of Systems , 553 (2003).

[4] J. P. Desmond, SOFTWARE Mag (2008).

[5] NIST, Software errors cost u.s. economy $59.5 billion annually - nist 2002-10, 2002.

[6] B. Gain, WIRED Magazine (2006).

[7] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*, The MIT Press, 2000.

[8] J. C. King, Commun. ACM **19**, 385 (1976).

[9] S. Khurshid, C. S. Psreanu, and W. Visser, Generalized symbolic execution for model checking and testing, in *In Proceedings of the Ninth International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 553–568, Springer, 2003.

[10] M. Sagiv, T. Reps, and R. Wilhelm, Solving shape-analysis problems in languages with destructive updating, in *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT*

symposium on Principles of programming languages, pages 16–31, New York, NY, USA, 1996, ACM.

[11] X. Deng, Robby, and J. Hatcliff, Towards a case-optimal symbolic execution algorithm for analyzing strong properties of object-oriented programs, in *SEFM '07: Proceedings of the Fifth IEEE International Conference on Software Engineering and Formal Methods*, pages 273–282, Washington, DC, USA, 2007, IEEE Computer Society.

[12] G. Andrews, *Foundations of Multithreaded, Parallel, and Distributed Programming*, Addison-Wesley, 1999.

[13] J. Dean and S. Ghemawat, Commun. ACM **51**, 107 (2008).

[14] R. Lämmel, Sci. Comput. Program. **68**, 208 (2007).

[15] K. R. M. Leino, Inf. Process. Lett. **93**, 281 (2005).

[16] R. Boyapati, S. Khurshid, and D. Marinov, Korat: Automated testing based on java predicates, in *In Proc. International Symposium on Software Testing and Analysis (ISSTA*, pages 123–133, ACM Press, 2002.

[17] S. Misailovic, A. Milicevic, N. Petrovic, S. Khurshid, and D. Marinov, Parallel test generation and execution with korat, in *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 135–144, New York, NY, USA, 2007, ACM.

[18] K. Sen, Concolic testing, in *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 571–572, New York, NY, USA, 2007, ACM.

[19] K. Sen and G. Agha, Automated systematic testing of open distributed programs, in

In Fundamental Approaches to Software Engineering (FASE06), volume 3922 of LNCS, pages 339–356, Springer, 2006.

[20] C. Barrett and C. Tinelli, Cvc3, in *CAV*, pages 298–302, 2007.

[21] B. Dutertre and L. D. Moura, The yices smt solver, Technical report, 2006.

Appendix A

Root Closest Parallelization

This appendix describes in detail the problem that root-closest parallelization addresses and how root-closest parallelization could be implemented.

```
public static int intTest(int x){
        if(x > 10){
                if(x > 20){
                        return 1;
                }
                else{
                        return -1;
                }
        }
        else{
                if(x > 5){
                        return 0;
                }
                else{
                        return 0;
                }
        }
}
```

Listing A.1: *Java method intTest(int x)*

To illustrate the sub-optimal behavior of eager parallelization refer to the simple method in listing A.1. When Kiasan starts analyzing this method a single worker starts to expand the execution tree. That worker does not have the full information of the fully expanded

execution tree in figure A.1. Instead, the execution tree is gradually expanded as the symbolic state space is explored. Figure A.2 shows the execution tree partially expanded, and $w1$'s current symbolic state is the deepest expanded node. If at this point another worker becomes idle, $w1$ may parallelize on its current state, assigning the other worker one of the two choices available. Figure A.3 illustrates the immediate result of this parallelization.

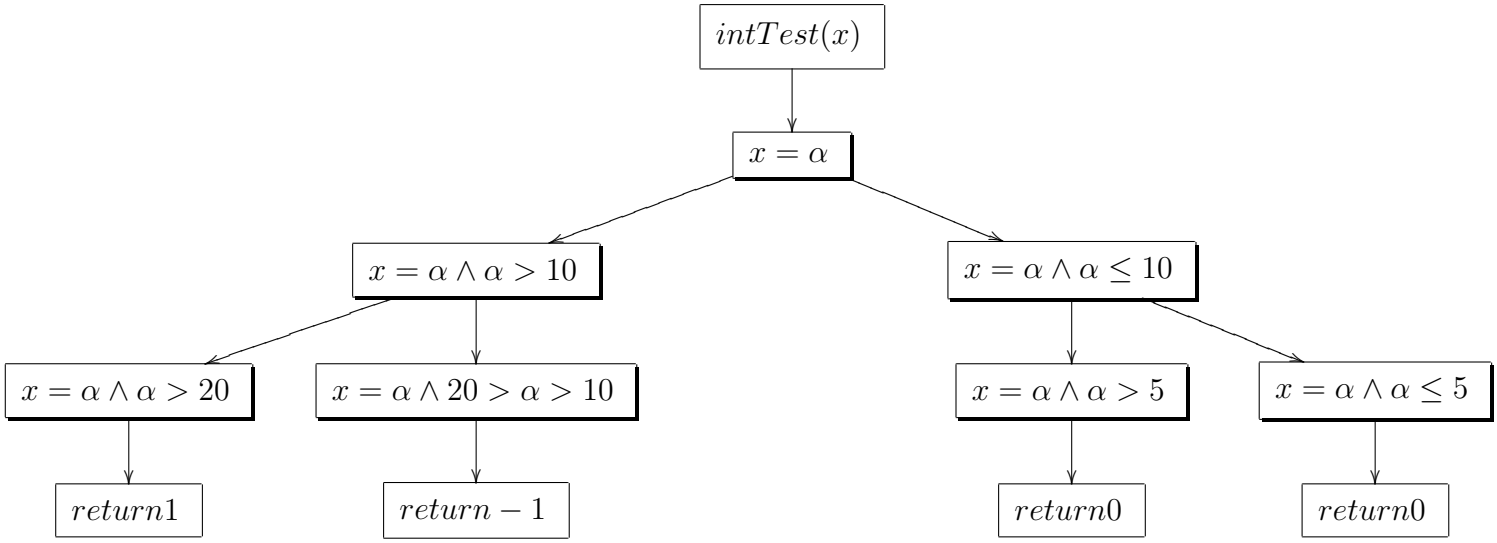


Figure A.1: *The fully expanded execution tree for intTest(x), The nodes with drop shadows represent the state if the symbolic execution after a certain choice has been followed. The arcs represent a choice. A trace from the root of the tree to a leaf represents a path through the method intTest(x), as determined by symbolic execution.*

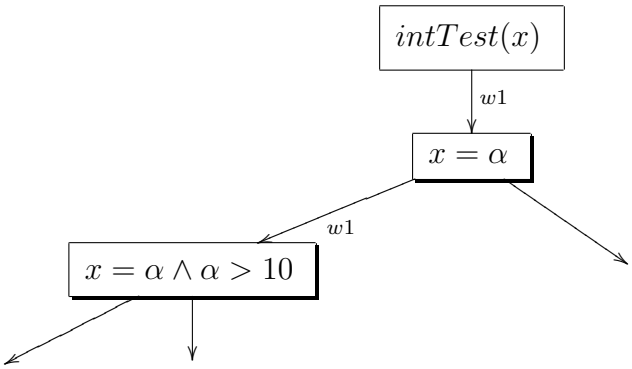


Figure A.2: *The partially expanded execution tree for intTest(x) at time $t(x)$. The covered choices are annotated by what worker explored them (in this case worker $w1$). There are three choices that may be distributed to another worker at $t(x)$.*

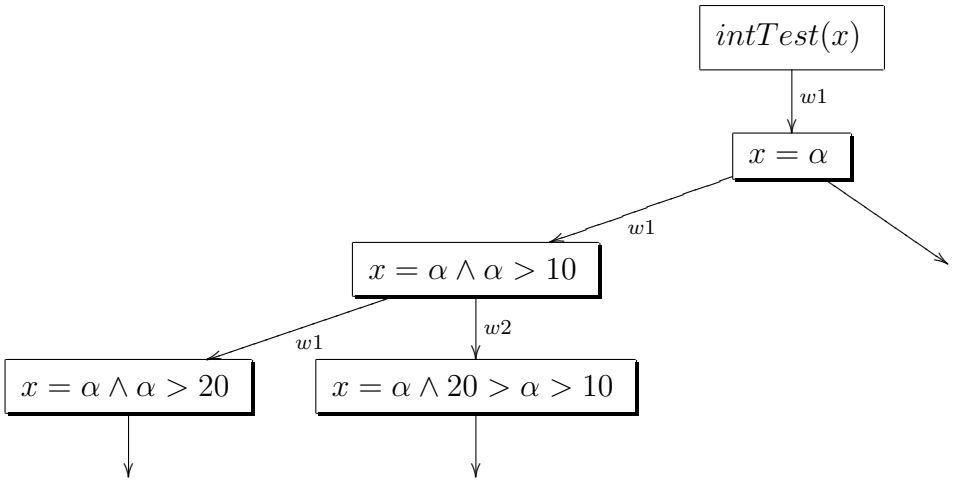


Figure A.3: *The partially expanded execution tree for intTest(x) at time $t(x + c)$. The covered choices are annotated by what worker explored them (in this case worker $w1$ and $w2$). Here, eager parallelization has been applied to parallelize the immediate choice from the previous state of w1.*

Figure A.4 shows the likely full expansion of the execution tree using two workers and eager parallelization. Here, there were at least 2 instances of parallelization, and each of those parallelizations parallelized a subtree of height 2.

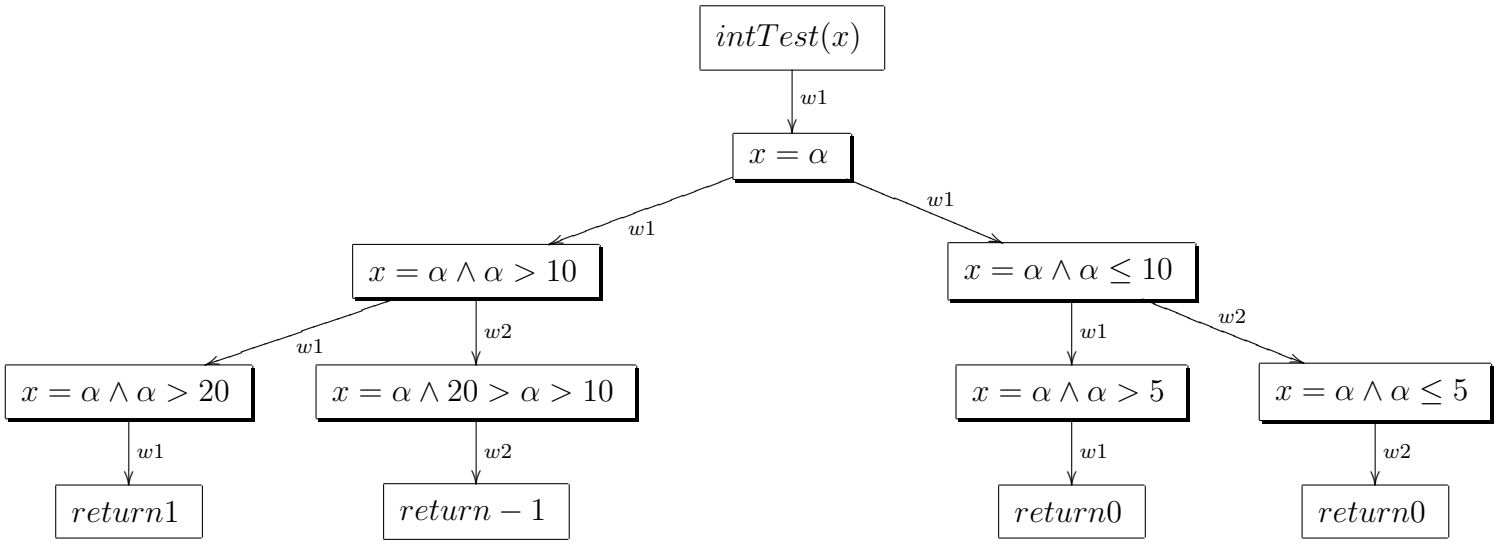


Figure A.4: *A possible fully expanded execution tree for intTest(x), which has been explored by two workers with eager parallelization. In this scenario, w1 has completed the initial path it explored and backtracked to the first choice in the tree while w2 was busy finishing. When w2 becomes idle again, w1 generates a workunit from w1's current state.*

If Kiasan would parallelize choices close to the root of the execution tree by default, then the probability that a subtree assigned to a worker is substantial computationally increases. How would the previous example play out if Kiasan used a 'root-first' parallelization? Refer to figure A.5. At time $t(x + c)$ instead of parallelizing the immediate choice, $w1$ looked through the parent states it explored for uncovered choices. Since the parent of $w1$'s current state is the state closest to the root with an uncovered choice, $w1$ parallelized that choice.

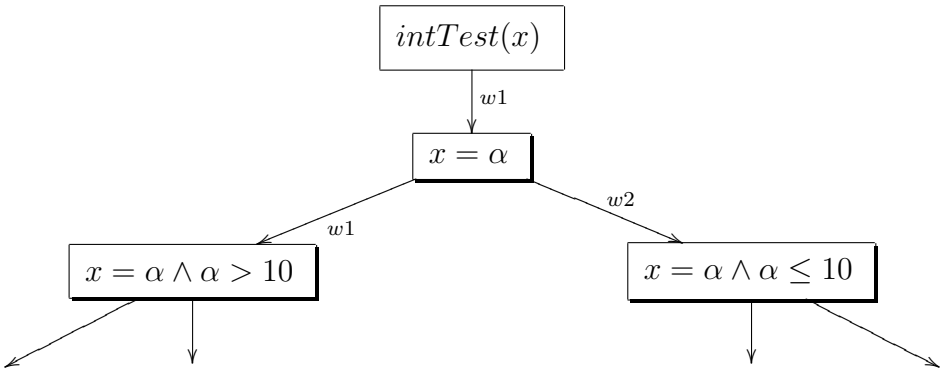


Figure A.5: *The partially expanded execution tree for intTest(x) at time $t(x + c)$. Here, root-closest parallelization was used to parallelize; $w1$ looked in its history to discover a choice that was closer to the root of the execution tree and then generated a work unit from that choice.*

Now, as analysis proceeds, worker $w2$ has a much more substantial subtree to explore, resulting in more effective parallelism (figure A.6) because both workers remained actively analyzing for a longer amount of time.

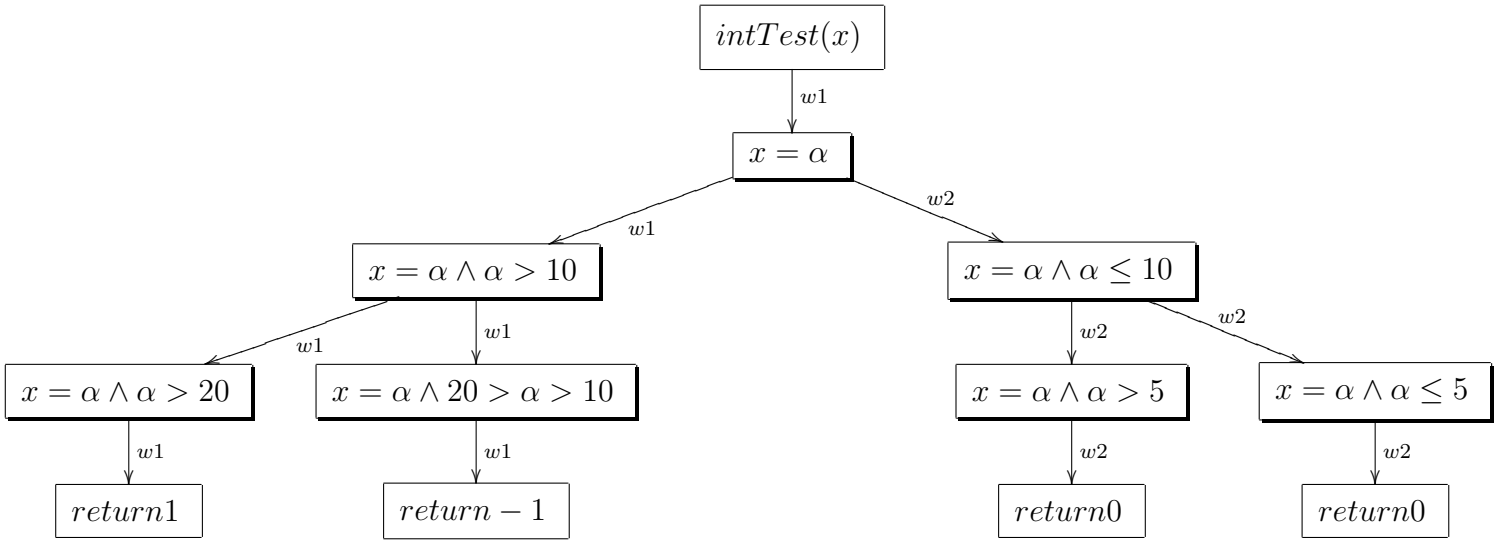


Figure A.6: *A possible fully expanded execution tree for intTest(x), which has been explored by two workers with root-first parallelization. Here, the load was much more evenly balanced between the two workers vs. eager parallelization. Also note that less inter-worker communication had to take place, which would have a positive impact on parallelization efficiency.*

How could this be done? One way to do this would be to have distributed Kiasan workers remember each time they encounter a non-deterministic choice and store the choices tree-location in some sort of queue. When one of the system's workers becomes idle, the active workers can now generate a work unit from the information stored in that queue as opposed to the current non-deterministic choice. This would ensure that each work-unit generated would describe a sub-tree as close to the root as possible given the current system state.

However, engineering this solution is not straightforward. An efficient way of recording what nodes in a worker's current subtree are uncovered needs to be used, otherwise any performance gained by choosing work units in this fashion will be lost. For example, It would not be efficient to store the analysis state at each uncovered node, because that will require processor time and a significant amount of system memory. A good choice instead would be to use the execution schedule Kiasan already stores for each prior non-deterministic choice. Kiasan maintains this schedule in its stack of backtracking information. This information is used by Kiasan to rewind the analysis state when Kiasan reaches the end of a path condition. This means that the backtracking information already contains the history of non-deterministic choices for a particular worker.

Now, the condition for parallelization is not just if the current state has uncovered choices, but if any previous state has uncovered choices. Refer to algorithms 23 and 24. Each time Kiasan has an uncovered choice, the index to that choice is archived in the *sendQ*, FIFO, before Kiasan proceeds. Next, when a worker becomes idle, Kiasan only needs to check the contents of the *sendQ* to determine what previous choice is still uncovered.

Sending a work unit also becomes slightly more complicated. In eager parallelism, Kiasan immediately backtracks after sending a work unit. This serves to automatically cover the choice that was just dispatched to another worker in the local worker. In root-closest parallelism, a choice from this history of the current state may be parallelized. This means that the backtracking mechanism is no longer appropriate to cover the parallelized choice. In root-closest parallelism, Kiasan should check if the parallelized choice is on the fringe or in

the history of the state space. If the choice is on the fringe the normal backtracking approach is used. If the choice is in the history, that choice is covered using the *coverSentChoice* method. (Refer to algorithm 25).

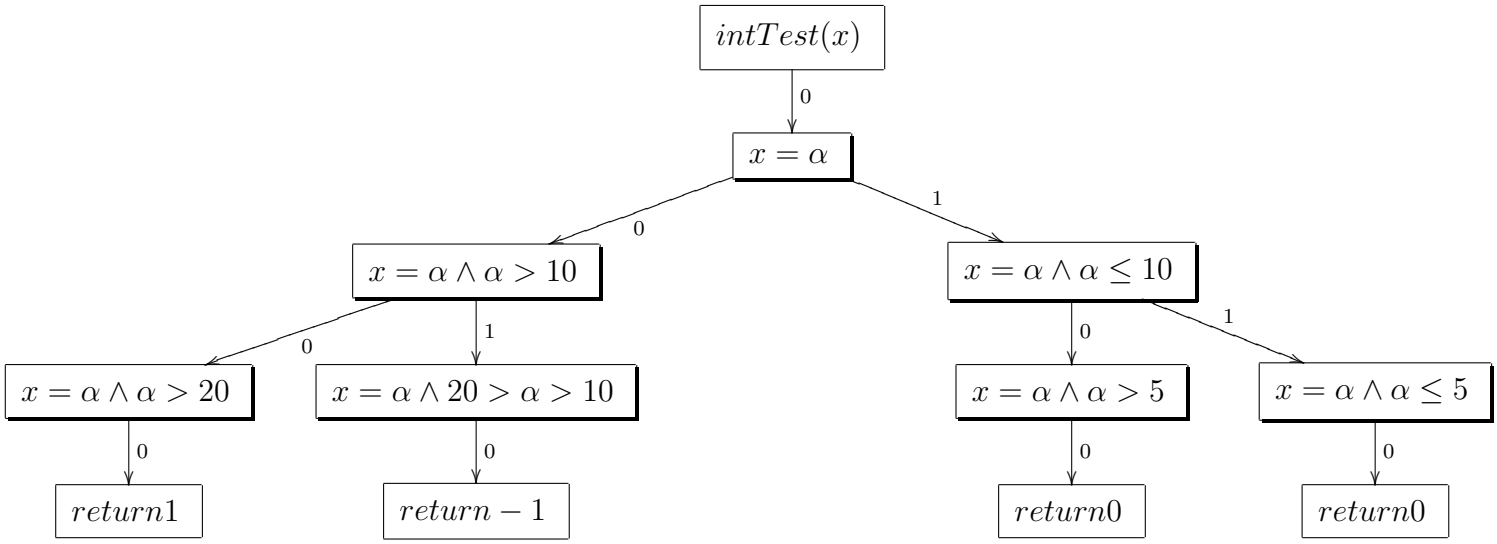


Figure A.7: *The fully expanded execution tree for intTest(x) each choice is annotated with its numerical index.*

Algorithm 22 explore() method for root-closest parallelization

```
while !shouldTerminate do
  unCovered ← false
  if !step() then
    if !backtrack() then
      break
    end if
  end if
  while shouldDistribute() do
    sendScheduleWU()
    if sentBacktracking = peek(backtrackingInfos) then
      backtrack()
      unCovered ← false
    end if
    step()
  end while
end while
```

Algorithm 23 shouldDistribute() for root-closest parallelization

```
checkUncovered()
if shouldBacktrack∧size(sendQ) = 0 then
  ret ← false
else
  if pingServer() then
    ret ← true
  else
    ret ← false
  end if
end if
return ret
```

Algorithm 24 checkUncovered() for root-closest parallelization

```
checkUncovered()
if isUncovered then
  add(sendQ, currentChoiceIndex)
  uncoveredCounter ← uncoveredCounter + 1
end if
```

Algorithm 25 sendScheduleWU() for root-closest parallelization

```
shortestUncoveredList ← buildShortestUncoveredList()
parallelizableBi ← getLastElement(shortestUncoveredList)
lastSendBi ← parallelizableBi
scheduleList ← generateSchedule(shortestUncoveredList)
if lastSendBi ! = peek(biStack) then
  archiveSchedule(lastSendBi)
  coverSentChoice(lastSendBi)
end if
intList ← makeIntListFromSched()
sendIntList ← combineLists(currentPrefix, intList)
sendWU(sendIntList)
```