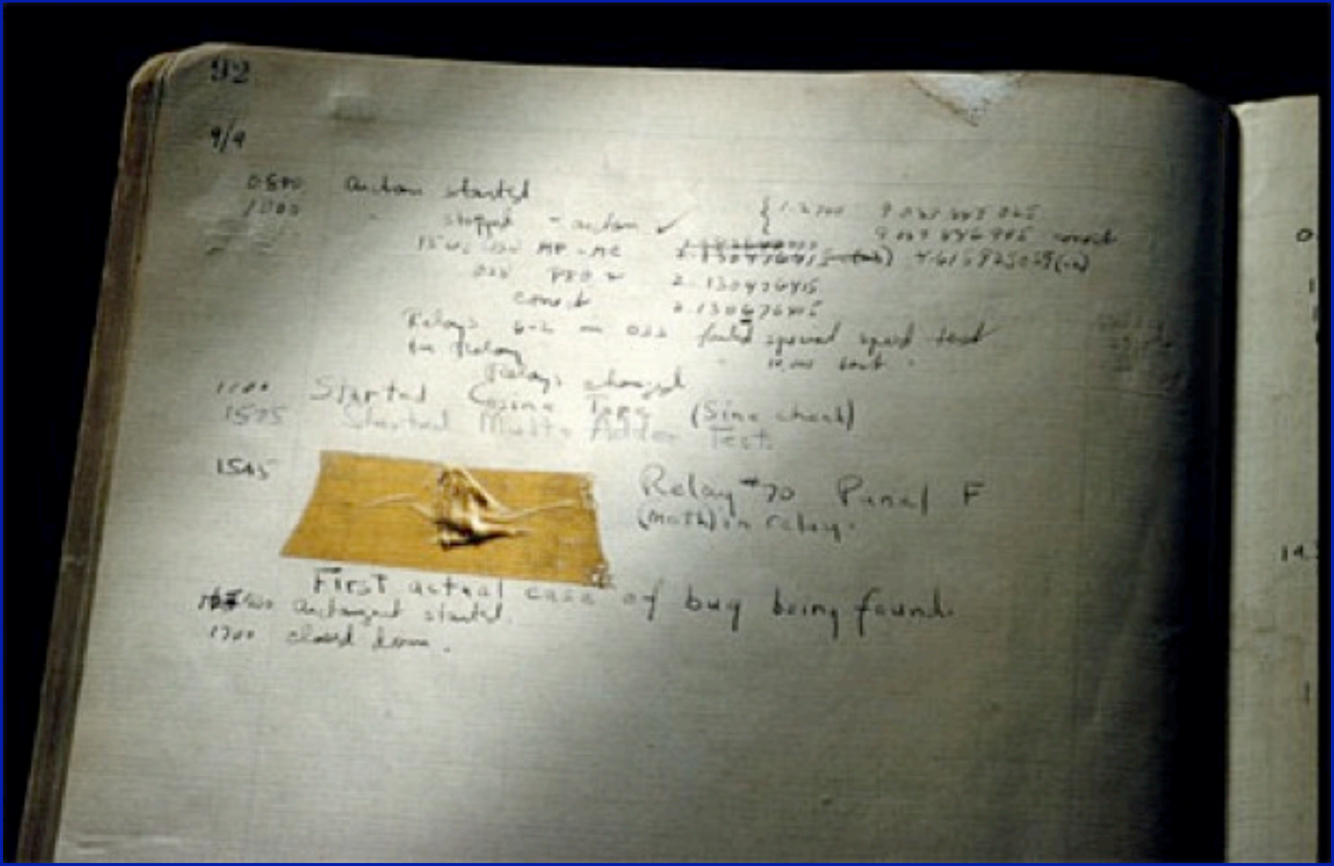
Optimizing Generation of Object Graphs in Java PathFinder

Milos Gligoric, Tihomir Gvero,
Steven Lauterburg, Darko Marinov,
Sarfraz Khurshid

JPF Workshop
1.5.8

Bugs–Six Decades Ago

1947: Harvard Mark II



92

Started Cosine Tape (Sine check)

Started Mult + Adder Test.

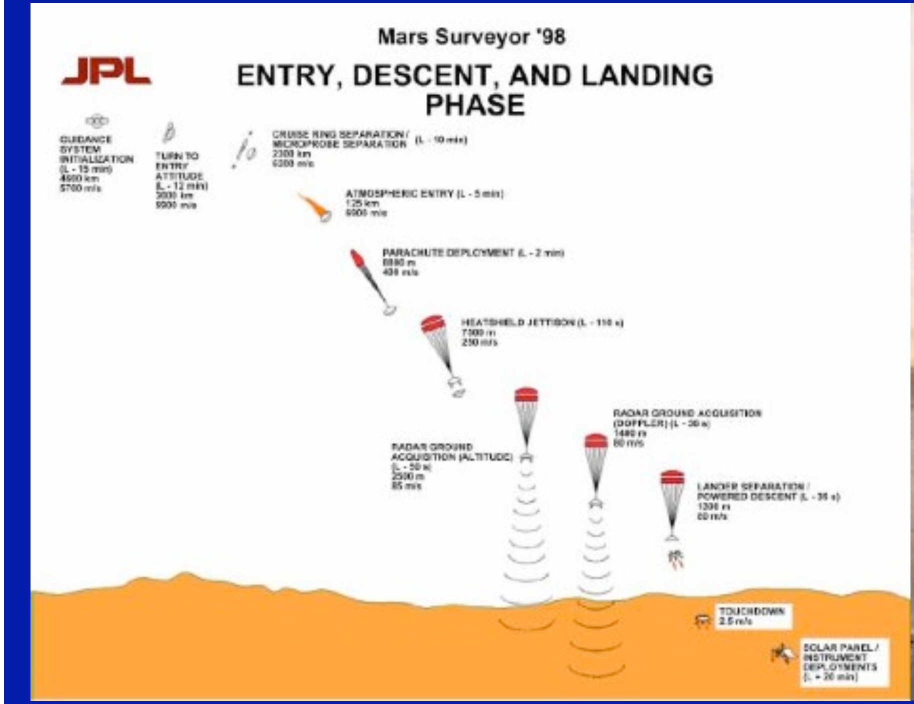
1545 Relay #70 Panel F (moth) in relay.

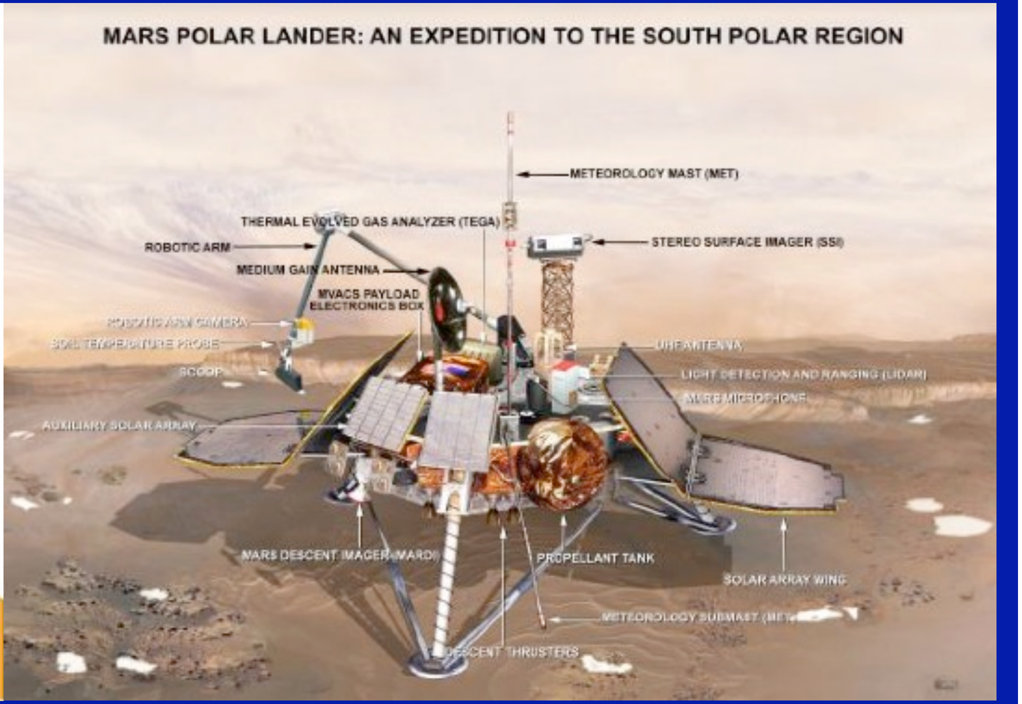
First actual case of bug being found.

Photo: *National Museum of American History*

Mars Polar Lander, 1999

Crashed–premature shut down at 40 meters altitude





Photos: *JPL/NASA*

USS Yorktown, 1997

“Dead in the water” for 3 hours

Photo: *navsource.org*

Java PathFinder

Java PathFinder (JPF) is a popular **explicit-state** model checker

- Directly checks properties of a wide range of Java programs

Implements a **customized** Java Virtual Machine

- Runs on the top of the host JVM

Two key operations for explicit state-space search:

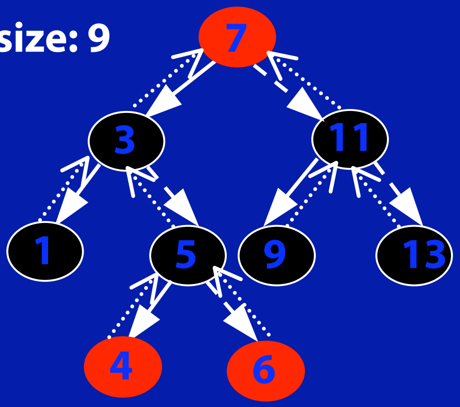
- **State backtracking**: stores/restores state to backtrack the execution during the state-space exploration
- Control over non-deterministic choices (including thread scheduling)

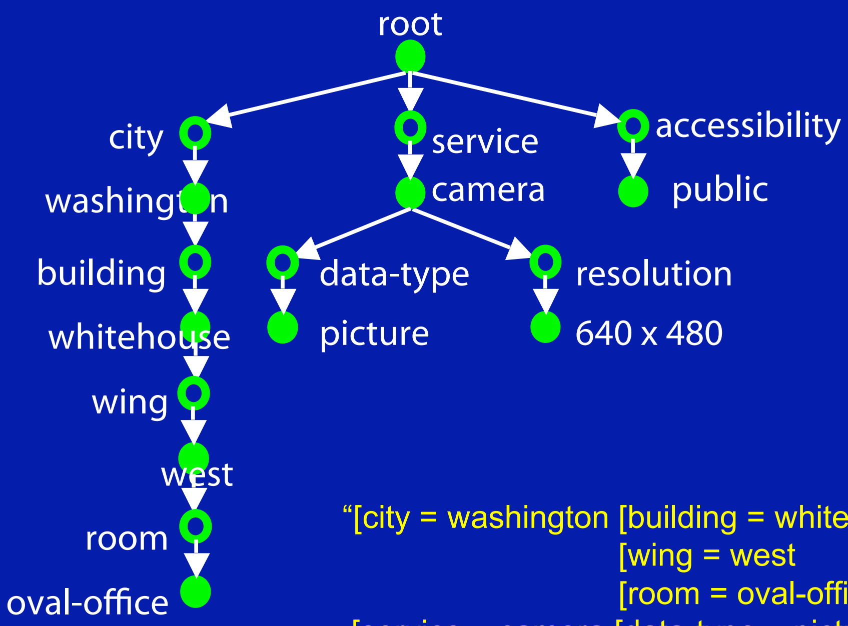
Traditional focus: check properties of control, such as deadlock

- Recent work: also check properties of **data**, such as acyclicity

Structurally Complex Data

size: 9





“[city = washington [building = whitehouse
[wing = west
[room = oval-office]]]]
[service = camera [data-type = picture
[format = jpg]]
[resolution = 640x480]]
[accessibility = public]”

Korat Framework

Enables systematic, constraint-based testing

- E.g., testing on all “small” inputs

constraint

solve

instances

concretize

tests

Implements a constraint solver for **imperative predicates**

Takes two inputs:

- A Java predicate that encodes constraints that represent properties of desired object graphs
- A **bound** on the size of the graph

Generates all **valid** graphs (within the given bound)

Performs a **backtracking search**

- Systematically explores the bounded space of object graphs

Korat in JPF

Korat solver was originally developed from scratch

- However, it can leverage tools, such as model checkers, which support backtracking as a fundamental operation

JPF has been used to generate object graphs

We use JPF as an implementation engine for Korat

- Start with a simple instrumentation of the Java predicate
- Observe that a direct implementation makes generation of object graphs unnecessarily slow in JPF
- Explore changes (optimizations) that could speed-up Korat in JPF

JPF State Operations

JPF uses a special representation for the state of program it checks

It performs three basic kinds of operations on the state representation:

- **Bytecode execution**: manipulates the state to execute the program bytecodes
- **State backtracking**: stores/restores state to backtrack the execution during the state-space exploration
- **State comparison**: detects cycles in the state space

Our changes target each of these operations

Outline

Overview

Example

Korat search in JPF

Optimizing Korat search in JPF

Evaluation

Conclusions

Example Korat Input

A set implemented as a red-black tree

```
public class RedBlackTree {
 Node root;
 int size;

 static class Node {
  int element;
  boolean color;
  Node left, right, parent;
  ...
 }

 boolean repOk() {
  if (!isTree()) return false;
   if (!hasProperColors())
    return false;
   if (!areElementsOrdered())
    return false;
   return true;
 }
```

```
boolean isTree() {
  if (root == null) return size == 0; // is size correct
  if (root.parent != null) return false; // incorrect parent
  Set<Node> visited = Factory.<Node>createSet();
  visited.add(root);
  Queue<Node> workList = Factory.<Node>createQueue();
  workList.add(root);
  while (!workList.isEmpty()) {
    Node current = workList.remove();
    if (current.left != null) {
      if (!visited.add(current.left))
        return false; // not a tree
      if (current.left.parent != current)
        return false; // incorrect parent
      workList.add(current.left);
    }
... // analogous for current.right
}
return visited.size() == size; // is size correct
}
...
}
```

Korat Input and Output

Input:

- **Method repOk**—checks whether an object graph indeed represents a valid red-black tree
- **Finitization**—provides:
  - A bound for the number of objects of each class in one graph, e.g., number N of Node objects
  - A set of values for each field of those objects
    - E.g., root, left, right, and parent are either null or point to one of the N Node objects

Output:

- Enumeration of all valid non-isomorphic object graphs, within the bounds given in the finitization
  - E.g., all valid red-black trees within the bounds

Korat Search

Systematically explores the bounded input space

- Orders values for each field to build its **field domain**
- Starts the search using candidate with all fields set to the first value in their domain
- Executes repOk on the candidate
  - Monitors repOk’s execution dynamically
  - Records the field access order according to first access
- Generates the next candidate based on the execution
  - Backtracks on the last field in field access order
    - Prunes the search
    - Avoids equivalent candidates

Uses bytecode instrumentation to insert monitors

Korat Search in JPF

Simple implementation of Korat using code instrumentation

- Use Verify.getInt, which returns a non-deterministic value, to index into a field domain
- Use shadow boolean fields to monitor field accesses
- Track assigned objects of a field domain to ensure exploration of non-isomorphic structures

Example: Code instrumentation in JPF

```
public class RedBlackTree {
 // data for finitization and search
 static Node[] nodes;
 static int assigned_nodes;
 static int min_size;
 static int max_size;

 // instrumentation for "root" field
 Node root;
 boolean init_root = false;
 Node get_root() {
  if (!init_root) {
   init_root = true;
   int i = Verify.getInt(0,
          min(assigned_nodes + 1,
          nodes.length - 1));
   if (i == assigned_nodes + 1)
    assigned_nodes++;
  root = nodes[i];
  }
  return root;
 }
```

```
// instrumentation for "size" field
int size;
boolean init_size = false;
int get_size() {
  if (!init_size) {
    init_size = true;
    size = Verify.getInt(min_size, max_size);
  }
  return size;
}
static class Node {
... // analogous instrumentation for each field
}
boolean repOk() {... /* same as before*/}
boolean isTree() {
  if (get_root() == null) return get_size() == 0;
  if (get_root().get_parent() != null) return false;
   ... // replace read of each field "f"
       // with a call to "get_f" method
}
```

Example: Finitization in JPF

```
// "N" is the bound for finitization
static void main(int N) {
  nodes = new Node[N + 1]; // nodes[0] = null;
  for (int i = 1; i < nodes.length; i++) nodes[i] = new Node();
  min_size = max_size = N;
  RedBlackTree rbt = new RedBlackTree();
  // this one call to "repOk" will backtrack a number of times,
  // setting the fields of objects reachable from "rbt"
  if (rbt.repOk()) print(rbt);
}
... // end of class RedBlackTree
}
```

A bound for the number of objects of each class

Optimizing Search in JPF

Groups of changes made in JPF:

- Modifying interpreter
- Reducing state comparison costs
- Reducing backtracking costs for heap
- Reducing bytecode execution costs
- Reducing costs for stacks and threads
- Reducing other overhead costs

These changes reduce the search time by over an order of magnitude

Modifying Interpreter

Idea: instead of code instrumentation, change the interpreter itself

Modified the class that implements the bytecode that reads a field from an object

- To check whether a field is initialized
- If not to create a non-deterministic choice point

No need to instrument the code

Makes it much easier to use Korat on JPF

Reducing State Comparison Costs

JPF is a stateful model checker:

- Stores (hash value of) the states that it visits
- Compares (hash value of) newly encountered states with the visited states

Idea: disable state comparison

- Korat search does not need any state comparison
- The search always produces different states

Reducing State Comparison Costs (2)

Idea: reduce garbage collection (GC)

- GC helps the state comparison
- Unnecessary to perform GC that often
- Perform GC occasionally, only to reduce the amount of memory

Reducing Backtracking Costs for Heap

State backtracking is expensive operation

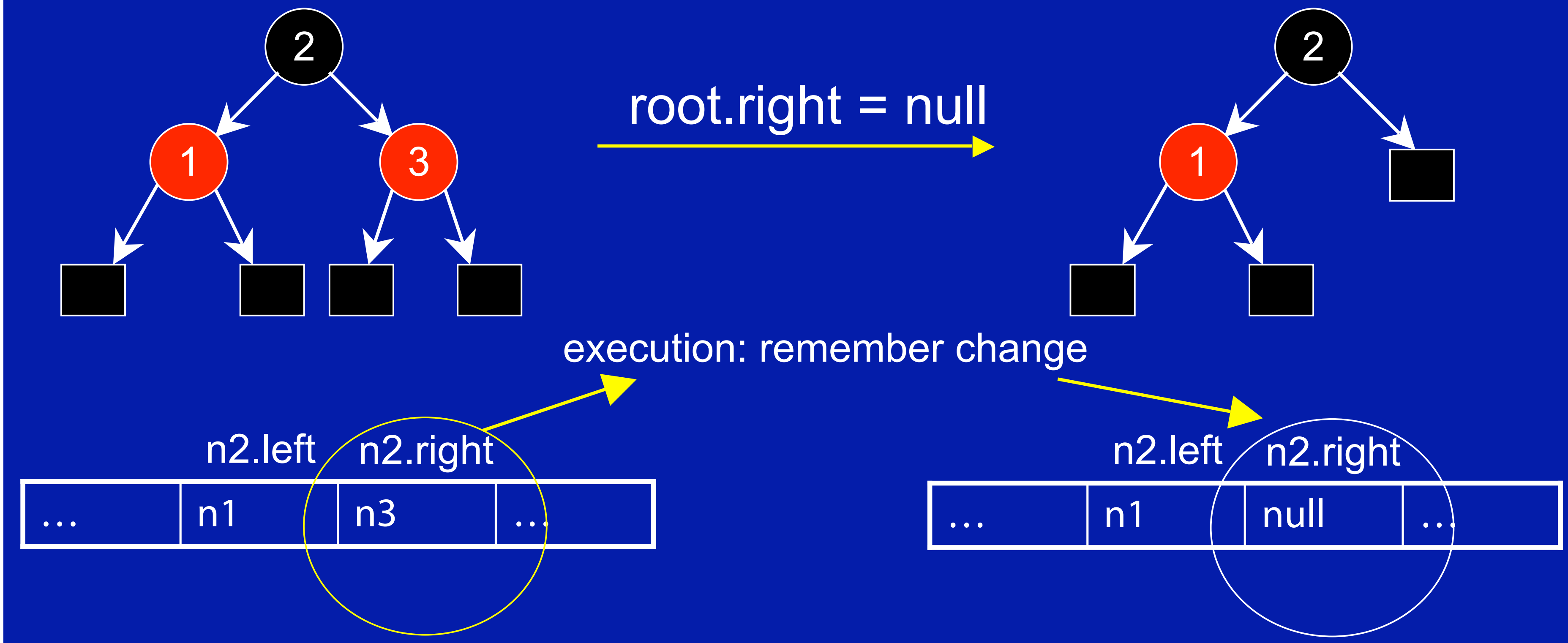
- Storing and restoring the entire JVM states at the choice points

Idea: undo backtracking

- Incrementally stores and restores states
- Only keeps track of the state changes that happen between choice points
- Later restores the state by undoing these state changes

Undo Backtracking

An execution of a field assignment



root.right = null

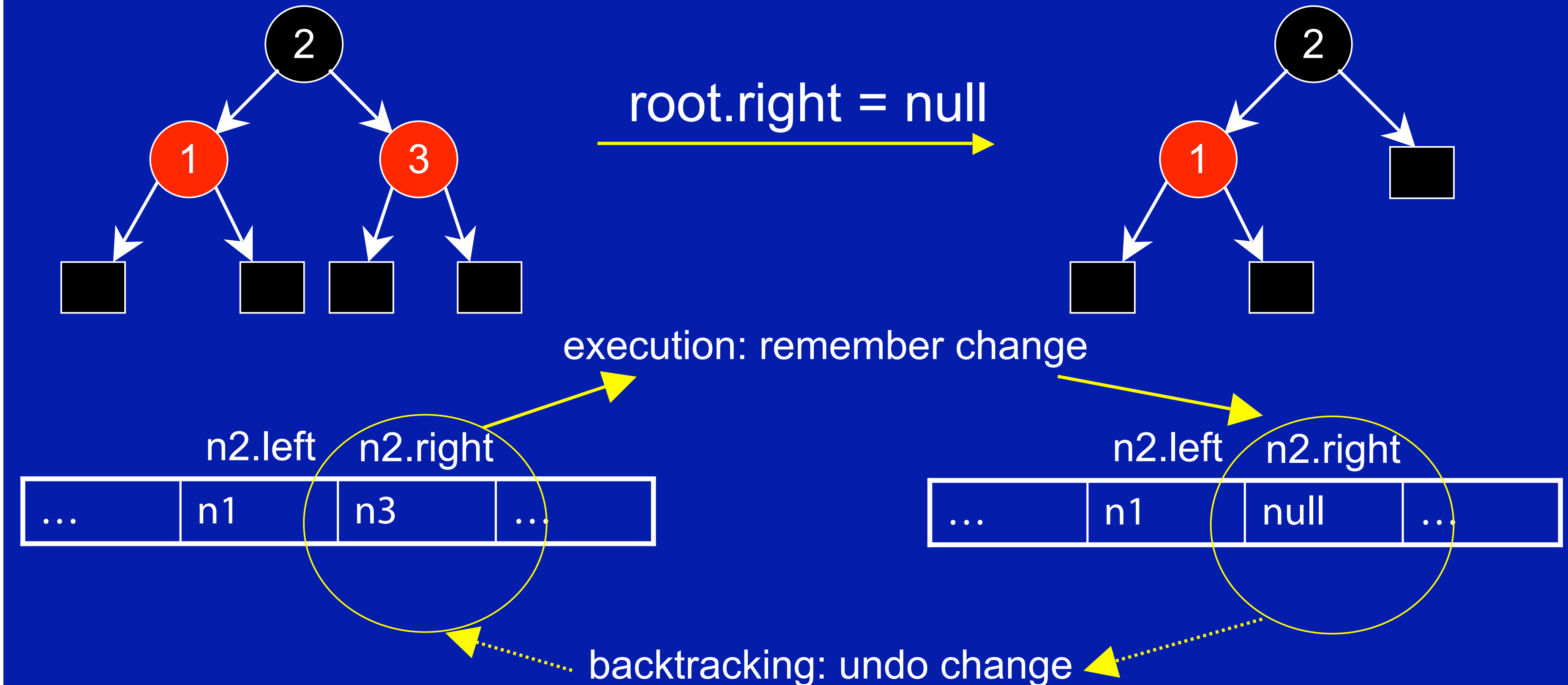
execution: remember change

	n2.left	n2.right	
…	n1	n3	…

	n2.left	n2.right	
…	n1	null	…

Undo Backtracking (2)

Reduces the execution time as it does not require JPF to clone the integer array that encodes all fields of an object whose one field is being written to



root.right = null

execution: remember change

backtracking: undo change

Reducing Backtracking Costs for Heap (2)

Idea: special handling of singleton non-deterministic choice

- Verify.getInt(int lo, int hi) does not need to create a backtracking point if lo == hi

Both Undo backtracking and Singleton non-deterministic choice changes have been added to JPF and are publicly available

Reducing Bytecode Execution Costs

JPF is an interpreter for Java bytecodes

For most bytecodes, JPF simply performs as a regular JVM

Slow in executing even regular bytecodes

Idea: use simple collections

- repOk methods build fairly small collections
- A simple library of collections that execute faster on JPF, when collections are small

Idea: execute on JVM

- Moving our library structures execution from the JPF level to the underlying JVM level

Reducing Costs for Stacks and Threads

A major portion of JPF time goes on manipulation of stacks and threads

Idea: optimize stack handling

- Simplify stack frames
- Shallow cloning rather than deep cloning of some stack frames
- Avoids "copy on write" when a stack frame is not shared

Idea: optimize thread handling

- Korat operates on single-threaded code
- Unnecessary to pay the overhead for multi-threaded code
  - Recall DynamicAreaIndex triples that Peter mentioned

Reducing Other Overhead Costs

JPF is built as an extensible tool

JPF uses:

- Listeners
- Logging

Listeners and logging provide overhead in execution even when they are not needed

Configuration flag to turn off listeners and logging

Evaluation: Time Speedup

Experiment		Time (sec)							Time speedup
Subject	N	Base	Mode 1	Mode 2	Mode 3	Mode 4	Mode 5	Mode 6	
BinaryTree	11	1,403.75	1,398.66	658.92	223.12	72.15	52.12	44.86	31.29x
BinHeap	8	885.94	866.78	551.16	278.70	90.11	81.90	66.89	13.25x
DisjSet	5	159.25	161.91	66.14	16.12	16.02	14.87	12.13	13.12x
DoublyLL	11	1,632.86	1,576.30	649.86	158.17	123.14	101.04	85.91	19.01x
FaultTree	6	5,348.47	5,102.74	3,456.12	2,405.72	631.90	634.98	505.57	10.58x
HeapArray	8	1,504.12	1,659.79	678.03	136.15	136.28	103.51	86.43	17.40x
RedBlackTree	9	886.79	890.70	365.47	128.05	112.23	101.05	77.44	11.45x
SearchTree	8	1,068.02	1,056.18	457.72	114.16	110.15	87.07	71.46	14.95x
SinglyLL	11	4,993.96	4,898.96	2,253.04	766.14	316.87	239.23	205.31	24.32x
SortedList	9	172.11	168.21	65.32	11.21	11.09	8.26	7.50	22.96x

Exploration time for various modications to basic Korat-JPF

- *Mode 1:* Modifying interpreter
- *Mode M:* includes all of the optimizations used in *Mode M-1*
- *Mode 2:* Reducing state comparison costs
- *Mode 3:* Reducing backtracking costs for heap
- *Mode 4:* Reducing bytecode execution costs
- *Mode 5:* Reducing costs for stacks and threads
- *Mode 6:* Reducing other overhead costs

- *Modifications of JPF reduce the search time by over an order of magnitude*

Evaluation: Memory Savings

Experiment		Peak Memory (MB)			Memory savings
Subject	N	Base	Mode 2	Mode 6	
BinaryTree	11	88.19	6.76	5.53	15.93x
BinHeap	8	35.32	6.68	5.67	6.23x
DisjSet	5	15.15	5.68	5.62	2.69x
DoublyLL	11	83.70	6.53	5.78	14.47x
FaultTree	6	258.91	7.07	6.20	41.75x
HeapArray	8	21.17	5.68	5.58	3.79x
RedBlackTree	9	63.52	7.00	5.95	10.66x
SearchTree	8	84.85	6.76	5.79	14.64x
SinglyLL	11	262.48	6.50	5.76	45.53x
SortedList	9	14.82	6.08	5.21	2.84x

Peak memory for selected modifications to basic Korat-JPF

- Used Sun's jstat monitoring tool to measure the peak usage of garbage-collected heap

Summary

Implemented the Korat search algorithm in JPF

A basic implementation results in an slow search

Modified several core operations of JPF to speed up the search

Our modifications reduce the search time in JPF by over an order of magnitude

Two modifications are already included in the publicly available JPF

- http://mir.cs.uiuc.edu/jpf

Future work

- JPF as a solver for constraints in other languages
- Parallelize JPF constraint solver