Parallel Test Generation and Execution with Korat

Sasa Misailovic (Univ. of Belgrade)
Aleksandar Milicevic (Univ. of Belgrade & Google)
Nemanja Petrovic (Google)
Sarfraz Khurshid (Univ. of Texas)
Darko Marinov (Univ. of Illinois)

FSE 2007
September 06, 2007
Motivation

- Testing a program developed at Google
  - Input: based on acyclic directed graphs (DAGs)
  - Output: sets of nodes with specific link properties
- Manual generation of test inputs hard
  - Many “corner cases” for DAGs: empty DAG, list, tree, sharing (aliasing), multiple roots, disconnected components...
Automated generation with Korat

- Korat is a tool for automated generation of structurally complex test inputs
  - Well suited for DAGs

- User manually provides
  - Properties of inputs (graph is a DAG)
  - Bound for input size (number of nodes)

- Tool automatically generates **all** inputs within given bound (all DAGs of size S)
  - Bounded-exhaustive testing
Problem: Large testing time

- Korat can generate a lot of inputs
  - Example: DAGs with 7 nodes: 1,468,397
- How to reduce testing time?
  - Generation: Speed up test generation itself
  - Execution: Generate fewer inputs

Solutions

- **Parallel Korat**: Parallelized generation and execution of structurally complex test inputs
- **Reduction methodology**: Developed to reduce the number of equivalent inputs
Outline

- Overview
- Background: Korat
- Parallel Korat
- Reduction Methodology
- Conclusions
Korat: input

- User writes:
  - Representation for test inputs

```java
public class DAG {
  DAGNode[] nodes;
  int size;
}
```

- Imperative predicate method to identify valid test inputs
- Finitization defines search bounds
Imperative predicate: repOK

Methods that check validity of test inputs

```java
public class DAG {
    public boolean repOK() {
        Set<DAGNode> visited = new HashSet<DAGNode>();
        Stack<DAGNode> path = new Stack<DAGNode>();
        for (DAGNode node : nodes) {
            if (visited.add(node))
                if (!node.repOK(path, visited))
                    return false;
        }
        return size == visited.size();
    }
}

public class DAGNode {
    public boolean repOK() {
        ... } // 11 lines
    }
}```
Finitization

- Bounds search space
- Example
  - Number of objects
    - 1 DAG object \((D_0)\)
    - \(S\) DAGNode objects \((N_0, N_1, \ldots N_{S-1})\)
  - Values for fields
    - \(S\) exactly for size (could be 0..\(S\))
    - 0..\(S\)-1 children for each node
    - Each child is one of \(S\) nodes
Korat: output

- Generates structurally complex data
  - Example: DAG
    - Set of nodes and set of directed edges
    - No cycles along those directed edges
Korat: input space

- Korat exhaustively explores a bounded input space
- Finitization describes all possible inputs
  - Example for $S=3$

<table>
<thead>
<tr>
<th>$D_0$ size</th>
<th>len</th>
<th>$N_0$</th>
<th>$c_0$</th>
<th>$c_1$</th>
<th>len</th>
<th>$N_1$</th>
<th>$c_0$</th>
<th>$c_1$</th>
<th>len</th>
<th>$N_2$</th>
<th>$c_0$</th>
<th>$c_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
<td>$N_0$</td>
<td>$N_0$</td>
<td>0</td>
<td>$N_0$</td>
<td>$N_0$</td>
<td>0</td>
<td>$N_0$</td>
<td>0</td>
<td>$N_0$</td>
<td>$N_0$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>$N_1$</td>
<td>$N_1$</td>
<td>1</td>
<td>$N_1$</td>
<td>$N_1$</td>
<td>1</td>
<td>$N_1$</td>
<td>1</td>
<td>$N_1$</td>
<td>$N_1$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>$N_2$</td>
<td>$N_2$</td>
<td>2</td>
<td>$N_2$</td>
<td>$N_2$</td>
<td>2</td>
<td>$N_2$</td>
<td>2</td>
<td>$N_2$</td>
<td>$N_2$</td>
<td></td>
</tr>
</tbody>
</table>
Candidate vector

- Sequence of indexes into possible values
- Encodes 1 object graph, valid or invalid
- Example (invalid DAG)

```
size len c
D_0 0 0 - -
N_0   -
      c_0 c_1
N_1 1 1 -
      c_0 c_1
N_2 0 - -
      c_0 c_1
```

DAG
size: 3

```
N_0
N_1
N_2
```
Korat: search

- Starts from candidate vector with all 0’s
- Generates candidate vectors in a loop until the entire space is explored
  - For each vector, executes repOK to find
    (1) whether the candidate is valid or not
    (2) what next candidate vector to try out
  - Field-access stack
    - Korat monitors field accesses during execution of repOK
    - Backtracks on last accessed field on stack, pruning large portions of the search space
Korat: next candidate vector

- Backtracking on $N_1.c_0$
  
  | $D_0$ | $N_0$ | $N_1$ | $N_2$ |
  | size | len | $c_0$ | $c_1$ | $c_0$ | $c_1$ | $c_0$ | $c_1$ |
  | 0 | 0 | - | - | 1 | 1 | - | 0 | - | - |

- Produces next candidate (valid DAG)
  
  | $D_0$ | $N_0$ | $N_1$ | $N_2$ |
  | size | len | $c_0$ | $c_1$ | $c_0$ | $c_1$ | $c_0$ | $c_1$ |
  | 0 | 0 | - | - | 1 | 2 | - | 0 | - | - |

DAG size: 3

- $N_0$  
- $N_1$  
- $N_2$  

$c_0$
Two key Korat concepts

- repOK
  - User provides predicates that check properties of valid inputs

- Candidate vector
  - Used in Korat search
  - Next vector computed from previous by executing repOK
Outline

- Overview
- Background: Korat
- Parallel Korat
- Reduction Methodology
- Conclusions
Parallel Korat: design goals

- Target clusters of commodity machines
  - Google infrastructure
- Minimize inter-machine communication
  - Improves overall performances by removing any expensive message passing
  - Makes code easily portable
- Challenge for **load balancing**: partition search space among various machines statically (before starting parallel search)
  - No overlap of work among machines
Korat: easy for parallelization

- Candidate vector **compactly** encodes the entire search state, both
  - Part that has been explored
  - Part that is yet to be explored
- Easy to parallelize search by using candidate vectors as the bounds for the ranges that split state space
Korat: hard for parallelization

- Korat pruning
  - Makes search more efficient 😊
  - Makes search mostly sequential 😞
    - Next candidate vector depends on the execution of repOK on current candidate vector

- Implication: given an arbitrary candidate vector, cannot statically know if the search would explore that vector or not
- Cannot purely randomly choose candidate vectors for partitioning
Parallel Korat: four algorithms

- **Test generation** can be
  - **SEQquential**: use one machine
  - **PARallel**: use multiple machines

- **Test execution** always parallel, can be
  - **OFF-line**: generation and execution decoupled (all inputs stored on disk)
  - **ON-line**: execution follows generation (inputs not stored on disk)

- **Four algorithms**
  - **SEQ-OFF**, **SEQ-ON**, **PAR-OFF**, **PAR-ON**
SEQ-OFF algorithm

- Runs test generation sequentially (SEQ) and stores to disk all test inputs
- Distributes test inputs evenly across several worker machines to execute code under test in parallel (OFF)
- Use case
  - Generation requires a lot of search and produces only few inputs (so it is preferred to store them for future execution)
SEQ-ON algorithm

- Use case: do **not** store inputs on disk
- Goal: Run sequentially once (SEQ) but prepares to make future runs parallel
- Sequential test generation stores to disk $m$ equidistant candidate vectors: $\nu_1...\nu_m$
  - Union of ranges $[\nu_i, \nu_{i+1})$ covers entire space
  - Each range explores same # of candidates
- All future generations/executions done in parallel on $w \leq m$ worker machines (ON)
Equidistancing algorithm

- Challenge: Choose $m$ equidistant vectors not knowing total number before search
  - If we knew total $T$, we would store $T/m$-th

- Solution uses an array of size $2m$ to remember specific candidate vectors
  - Example for $m=3$
    - Fill out the array: 1,2,3,4,5,6
    - Halve the array: 2,4,6
    - Double distance: 2,4,6,8,10,12
    - Repeat these 3 steps: 4,8,12... 16,18,20...
Evaluation: SEQ-ON, DAGs of size 8

- Experiments on Google infrastructure
  - Up to 1024 machines, Google File System
  - Testing time: from 35.9 hours (1 machine) to 4 mins (1024 machines)

![Graph showing speed-up vs number of machines](image-url)
Evaluation: SEQ-ON, DAGs of size 7

- Experiments on Google infrastructure
  - Peek on 128 machines
    - Testing time: from 10 mins to 1/2 min
  - A lot of time goes on file distribution

![Graph showing speed-up vs. number of machines](image-url)
PAR-OFF algorithm

- Parallelizes the initial run (PAR)
  - Challenges:
    - How to partition input space into several ranges without generating all inputs as in SEQ-ON
    - Hard to estimate the number of vectors explored between two given vectors (Korat’s dynamic pruning)
  - Solution: use randomization
    - Randomly fast-forward search on one machine to generate vectors that cover the entire search space

- Parallelize search for generated vectors and write all generated test inputs to disk
- Performs test execution separately (OFF)
**Fast-forwarding algorithm**

- Randomly chooses $m$ candidate vectors
  - Starts from candidate with all 0’s (as Korat)
  - Repeatedly
    - Chooses randomly a number of usual Korat steps to apply
    - Chooses randomly a “jump” in search (discarding some fields from access stack)
    - Stores current candidate
  - If search space explored before storing $m$ candidates, repeat the process from 0’s
  - Sort the candidates by their indexes
Results for PAR-OFF

- Ran PAR-OFF to select $m$ candidates $v_1 \ldots v_m$
  - Divided # of candidates over largest range $[v_i, v_{i+1})$
- Repeated for 50 random seeds, averages:

![Graph showing speed-up vs. number of machines]

- Number of machines
- Speed-up
  - 1
  - 2
  - 4
  - 8
  - 16
  - 32
  - 64
  - 128
  - 256
  - 512
  - 1024

- Averages:
  - 7.93
  - 7.94
  - 8.08
Outline

- Overview
- Background: Korat
- Parallel Korat
- Reduction Methodology
- Conclusions
Reduction methodology

- Independent of parallel algorithms
- Goal to generate fewer equivalent inputs
  - Equivalent: either all or none show bugs
  - Korat prunes out some equivalent inputs
  - User may want to prune out even more
- Methodology: Manually change repOK
  - Add more checks to repOK to prune some valid (but equivalent) inputs
  - User encodes an ordering on candidates such that “larger” can be pruned
Equivalence of DAGs

- Three versions of repOK
  - Basic: no ordering
  - Children: number of immediate children
  - Descendants: total number of descendants

- DAGs of size 6: non-equivalent 5,984

<table>
<thead>
<tr>
<th>repOK size</th>
<th>Inputs</th>
<th>Time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td>22</td>
<td>1,336,729</td>
</tr>
<tr>
<td>Children</td>
<td>26</td>
<td>185,569</td>
</tr>
<tr>
<td>Descendants</td>
<td>34</td>
<td>21,430</td>
</tr>
</tbody>
</table>

Speedup: 60x exec.  7x gen.
Conclusions

- Developed parallel Korat
  - Example speedups evaluated at Google
    - Over 500x on 1024 machines for DAGs of size 8
    - Slowdown after 128 machines for DAGs of size 7

- Developed reduction methodology
  - Example improvements for DAGs of size 6
    - Over 7x reduction in generation time
    - Over 60x fewer test inputs (execution time)
http://korat.sourceforge.net

Thanks!
Isomorphic inputs

- Korat generates all valid non-isomorphic test inputs within given bounds
- Isomorphic object graphs have:
  - Same shape and primitive values
  - Potentially different node identities

Example

- DAG size: 3
- $N_0 \rightarrow N_1 \rightarrow N_2$
- $N_0 \rightarrow N_2 \rightarrow N_1$
Equivalent inputs

- Isomorphism $\neq$ equivalence
  - Example: Two DAGs are equivalent if they are isomorphic as graphs not object graphs

- Problem: Korat can generate object graphs non-isomorphic at concrete level but equivalent at abstract level, e.g.:

```
N_0 \rightarrow N_1 \rightarrow N_2
```

```
N_0 \rightarrow C_0 \rightarrow N_1 \rightarrow N_2
```