Optimizing Parallel Korat Using Invalid Ranges

Nima Dini  
The University of Texas at Austin  
Austin, Texas 78712  
nima.dini@utexas.edu

Cagdas Yelen  
The University of Texas at Austin  
Austin, Texas 78712  
cagdas@utexas.edu

Sarfraz Khurshid  
The University of Texas at Austin  
Austin, Texas 78712  
khurshid@ece.utexas.edu

ABSTRACT

An effective form of systematic testing is constraint-based testing where the user writes logical constraints to describe properties of desired inputs, and constraint solvers enumerate all tests within a bound on the input size. The key challenge in systematic constraint-based testing is efficiently exploring very large spaces of all possible inputs to enumerate the desired valid inputs. Previous work introduced the Korat technique to address this challenge. Korat uses desired input properties written as imperative predicates and performs a backtracking search that prunes large parts of the input space to enumerate all non-equivalent (i.e., non-isomorphic) inputs within a given bound on input size. While Korat’s pruning improves its performance, systematically generating large numbers of inputs and testing against them can be costly in practice.

This paper introduces a novel approach to reduce the cost of the Korat search in certain application scenarios. Our key insight is that sometimes the Korat search over the same state space is repeated across separate runs of Korat, and an earlier run of the search can be summarized to more efficiently perform a later run. Specifically, we introduce the idea of invalid ranges which succinctly encode parts of the exploration space that do not contain any valid inputs but have to be explicitly explored by the Korat search since it is unable to prune them. Our approach directly prunes these parts in a future run of Korat over the same input space. We develop our approach for two settings: a sequential setting where the Korat search is run using one worker (i.e., processing unit), and a parallel setting where the Korat search is distributed to several workers. In the parallel setting, we build on a previous technique for parallel Korat, namely SEQ-ON, and integrate invalid ranges with it. Our prototype tool MKorat embodies our approach. Experimental evaluation using 6 subjects from the standard Korat distribution show that MKorat achieves: in the sequential setting, a speedup of up to 2.82X over sequential Korat (in comparison, SEQ-ON does not provide any speedup in the sequential setting); and in the distributed setting, using up to 32 workers, a speedup of up to 38.84X over sequential Korat (using 1 worker), and up to 3.04X over SEQ-ON in terms of total execution time across the workers.

KEYWORDS

Constraint-based testing, Test input generation, Korat, Parallel search

1 INTRODUCTION

Systematic software testing [1, 3, 5, 13, 14, 16, 21, 25], which has its roots in the core idea of systematic exploration of bounded state spaces in model checking [6, 14, 37], has been used in a number of applications for finding subtle bugs in software systems. A particularly effective approach for systematic testing is constraint-based testing where logical constraints characterize desired inputs and expected program behaviors as preconditions and postconditions respectively [3, 25]. A number of different techniques embody this approach and support constraints written in different languages, including declarative languages [25] and imperative languages [3].

Our work focuses on the constraints written as imperative predicates, termed repOK [24], which are executable checks that characterize desired properties using an imperative language, e.g., Java, and likely pose minimal learning burden on users because of the wide use of such languages. The foundation of our work is the Korat technique for test input generation using imperative constraints [3, 26]. Given a repOK predicate, which characterizes desired inputs, and a finitization, i.e., a bound on the input size, Korat enumerates each non-isomorphic input within the bound such that executing repOK on the input returns true. Thus, the inputs generated by Korat form bounded exhaustive tests and include every valid input with respect to the given repOK and finitization bound. The space of all candidates to consider as inputs to repOK is usually very large, e.g., > 2^{72}, even for small bounds on input size, e.g., 10 nodes in a binary search tree [3]. The Korat algorithm performs pruning and isomorph-breaking to exhaustively explore such large input spaces. However, the application of Korat in practice is limited by two key factors: the size of the underlying state spaces and the number of valid inputs created.

We introduce a novel approach to reduce the cost of the Korat search in certain application scenarios. Our key insight is that the Korat search is sometimes repeated over the same state space across separate runs of Korat, and an earlier run of the search can be summarized to more efficiently perform a later run. Such a scenario arises, for example, when Korat search is used for testing multiple methods where some of the methods share a common input constraint but storing (all) the inputs is not feasible [26]: the testing approach chooses one method to test, and iteratively creates an input, runs the method, and checks its behavior until testing this chosen method is complete, and then selects the next method, and continues until all methods have been tested. Another example scenario arises when Korat is used as an external constraint solver [11]. To illustrate, consider systematic checking [20, 21] of two methods m() and n() of a class with class invariant inv(), which represents a precondition for both m() and n(). Checking m() requires checking the program p: “if (inv()) m();” and checking n() requires checking the program q: “if (inv()) n();”. Thus, checking m() and n() requires solving the same constraint inv() –
all execution paths that reach $m(t)$ in $p$ (likewise $nt$ in $q$) require $inv()$ to evaluate to true. Indeed, the same optimization opportunity arises when $n(t)$ is actually just an updated version of $m(t)$, which needs to be re-checked say after on a bug fix or feature addition in the context of evolution.

We introduce the idea of invalid ranges which succinctly encode parts of the exploration space that do not contain any valid input but must be explicitly explored by the Korat search since it is unable to prune them [8]. Our approach prunes these parts in a future run of Korat over the same input space. We develop our approach for two settings: a sequential setting where the Korat search is run using one worker (i.e., processing unit), and a parallel setting where the Korat search is distributed to several workers. In the sequential setting, we directly use invalid ranges to prune the search. In the parallel setting, we build on a previous technique, namely SEQ-ON [26], and integrate invalid ranges with it.

SEQ-ON was defined by the parallel Korat approach [26], which originally introduced the idea of parallel test generation and execution using Korat to mitigate the two key limiting factors. Conceptually, parallel execution of tests generated using Korat is relatively straightforward: distribute the tests evenly among the parallel workers. However, parallel generation of tests using Korat is a non-trivial problem because Korat’s pruning is inherently sequential: what to prune depends on what was explored and cannot simply be determined a priori. Specifically, Korat considers one candidate input at a time, checks the validity of the current candidate by executing $repOK$ against it, and uses the execution as a basis of creating the next candidate, and by doing so prunes many candidates from the search. Thus, evenly distributing the test generation workload among parallel Korat workers is challenging.

SEQ-ON was specifically designed to address the scenario where Korat is used to create inputs for testing a number of different methods under test but the inputs are not stored: for each method under test, inputs are created and the method executed against each input as it is created. A key contribution of the SEQ-ON algorithm is that it uses the first execution of Korat for input generation to create a fixed number of equidistant candidates based on the number of workers, which allows all subsequent executions of Korat on the same constraint solving problem to be performed in parallel such that each parallel worker only explores the range defined by two consecutive equidistant candidates, and the workload is evenly distributed among the parallel workers.

We define invalid ranges as sequences of consecutive invalid candidates that are explored by the standard Korat search. We summarize such a sequence succinctly using just two candidates as end-points during the first execution of Korat for input generation, and re-use it for more efficient exploration in the subsequent executions of Korat for input generation – the subsequent executions are able to prune invalid candidates that the initial Korat search was unable to prune and had to explicitly check using $repOK$. We apply invalid ranges in tandem with equidistant candidates to define a more effective technique for parallel test generation using Korat.

Our prototype tool MKorat embodies our approach. We show the effectiveness of our approach using a suite of controlled experiments. Specifically, we evaluate how MKorat compares with traditional Korat in a sequential setting and how the use of invalid ranges improves over just equi-distancing. Moreover, we evaluate how the performance of MKorat varies as the number of invalid ranges is increased.

This paper makes the following contributions:

- **Idea.** We introduce the idea to summarize and re-use parts of the state-space that do not contain any valid input to enhance solving of imperative constraints for systematic input generation.
- **Invalid ranges.** We define invalid ranges, which succinctly represent consecutive invalid candidates that the standard Korat search is unable to prune and must explicitly check by invoking $repOK$.
- **Test generation technique.** We introduce a new technique to optimize input generation using imperative constraints when the Korat search is re-run for the same exploration space. We develop our technique for a sequential setting and a parallel setting by building on the SEQ-ON algorithm from previous work [26] and integrating invalid ranges with it.
- **Evaluation.** We use a suite of 6 subjects from the standard Korat distribution to evaluate our approach. Experimental results show that MKorat achieves: in the sequential setting, a speedup of up to 2.82X over sequential Korat (in comparison, SEQ-ON does not provide any speedup in the sequential setting); and in the distributed setting, using up to 32 workers, a speedup of up to 38.84X over sequential Korat (using 1 worker), and up to 3.04X over SEQ-ON in terms of total execution time across the workers.

We believe invalid ranges provide the foundation for an exciting method for increasing the efficacy of systematic testing and analysis using imperative constraints. While our focus in this paper is on re-execution of the Korat search over the same state space as the previous execution, we believe invalid ranges can be generalized to enable re-use in more general settings where state spaces among different executions differ, e.g., due to a change in the constraint being solved or the bound being used. We plan to address such settings in future work.

## 2 ILLUSTRATIVE EXAMPLE

We illustrate the concept of invalid ranges and the basis of our approach using an example from the Korat project’s source code. Figure 1 shows the Java declaration of the red-black tree data structure [7], which implements a balanced binary search tree, the $repOK$ predicate that implements a check for the structural integrity constraints of a red-black tree (acyclicity, correct coloring of nodes etc.), and the finitization description (the $finRedBlackTree$ method) which sets a bound on Korat search. Each tree has a root node and caches the number of nodes in the size field. Each node contains an integer key and value, and has a left child, a right child, and a parent pointer, as well as a color, which is RED or BLACK. To test a method that operates on an input red-black tree, such as instance method “add(int x)”, we must generate a valid tree $t$, i.e., $t().repOK()$ returns true, as an input (the receiver object) as well as provide an integer input $x$. To create valid red-black trees, Korat uses the given $repOK$ method and finitization to create the space

1https://korat.svn.sourceforge.net/svnroot/korat/trunk, revision 12
Optimizing Parallel Korat Using Invalid Ranges

```java
public class RedBlackTree {
    private Node root = null;
    private static int size = 0;
    private static int RED = 0;
    private static int BLACK = 1;

    public static class Node {
        int key;
        int value;
        Node left = null;
        Node right = null;
        Node parent;
        int color = BLACK;
    }

    public boolean repOK() { ... }

    // Bound Korat exploration to trees with 'num' Nodes.
    public static IfinInitialization finRedBlackTree(int num) {
        IniHNitation f = IniHNitationFactory.create(
            RedBlackTree.class);
        IClassDomain entryDomain = f.createClassDomain(
            Node.class, num);
        IObjSet entries = f.createObjSet(Node.class, true);
        entries.addClassDomain(entryDomain);
        IIntSet sizes = f.createIntSet(num, num);
        IIntSet keys = f.createIntSet(-1, num - 1);
        IIntSet values = f.createIntSet(0);
        IIntSet colors = f.createIntSet(0, 1);
        f.set("root", entries);
        f.set("size", sizes);
        f.set("Node.left", entries);
        f.set("Node.right", entries);
        f.set("Node.parent", entries);
        f.set("Node.color", colors);
        f.set("Node.key", keys);
        f.set("Node.value", values);
        return f;
    }
}
```

![Figure 1: RedBlackTree subject](image)

**Figure 1:** RedBlackTree subject.

![Figure 2: Korat generates 961 candidates for finRedBlackTree(4). A valid candidate index is marked by a cross (x).](image)

**Figure 2:** Korat generates 961 candidates for `finRedBlackTree(4)`.

![Figure 3: Valid red black trees with 4 nodes generated by Korat. Node keys are uniquely assigned from set S = {1, 2, 3, 4}.](image)

**Figure 3:** Valid red black trees with 4 nodes generated by Korat. Node keys are uniquely assigned from set S = {1, 2, 3, 4}.

The core approach performs an initial run of Korat to build invalid ranges (Figure 4), which are used for additional pruning in subsequent runs of Korat when exploring the same input space. In this example, the last range has 961-768=193 elements.

3 TECHNIQUE

In this section, we first describe our technique to build invalid ranges using an initial run of Korat. Next, we recall the original SEQ-ON algorithm [26] (Section 3.1) and present our parallel technique MKorat, which builds on SEQ-ON (Section 3.2). Next, we discuss some implementation details of MKorat (Section 3.3) and several key properties of our prototype (Section 3.4).

Our core approach performs an initial run of Korat to build invalid ranges (Figure 4), which are used for additional pruning in subsequent runs of Korat when exploring the same input space. In this example, the last range has 961-768=193 elements.
3.1 Background: Equi-distancing for SEQ-ON

The key novelty of the original parallel test generation and execution algorithm using Korat [26] is to not store all inputs for creating equidistant candidates. The design goal behind this algorithm is to store sufficient information during the first sequential run, so that all future runs can be parallelized and load-balanced. Specifically, this algorithm obtains a sequence of equidistant candidate vectors \( (C_1, C_2, ..., C_n) \), i.e., Korat explores (almost) the same number of candidates in any range \([C_i, C_{i+1})\) for \( 0 \leq i < n - 1 \) and \([C_{n-1}, C_n)\), and the union of all such ranges, becomes the entire explored space \( \bigcup_{i=0}^{n-2}[C_i, C_{i+1}) \cup [C_{n-1}, C_n] = [C_0, C_n) \), where \( C_0 \) is the initial and \( C_n \) is last candidate vectors explored.

Figure 5 shows the pseudo-code of the SEQ-ON algorithm. The `equiDistantCandidates` function keeps an array of candidates, with size twice as large as the number of maximum workers. As the number of explored candidates in Korat search is not known beforehand, this technique records each candidate being explored in the first round. When the array capacity is full, it moves the candidates at even indexes in the array to left half, and continues recording every second candidate in the right half. In the next round, it records every fourth candidate being explored. This process continues, and at the end, the function returns the candidates to keep for the future parallel executions.

3.2 MKorat

Our parallel technique, MKorat, builds on SEQ-ON and stores \( m \geq 1 \) largest invalid ranges during the first sequential run, and excludes those ranges prior to distribution for future parallel runs. The remaining ranges \( (r_1, r_2, ..., r_m) \), are distributed with respect to the \( k \geq 1 \) equidistant candidate vectors maintained in SEQ-ON. Moreover, if any equidistant candidate vector \( C_q \) falls into a range \( r_j = [C_{q_1}, C_{q_2}) \), the algorithm breaks the range to \([C_{q_1}, C_{q_2}) \), \([C_{q_2}, C_{q_3}) \). The splitting phase continues until no range in the final collection of ranges \( Q = \{q_1, q_2, ..., q_t\} \) contains an equidistant candidate unless it is the starting endpoint of a range. Finally, each worker takes a subset of ranges from \( Q \) which belong to its equi-distant range.

Note that if an equi-distant range does not contain any valid candidate, it will be discarded and no worker will be assigned to that range, saving computational resources. Further, MKorat borrows the single-pass spirit of SEQ-ON and does not impose extra time and space complexity on top of this algorithm.

MKorat\( _{exc} \): Korat provides a command-line option --cvWrite, which writes all explored candidates to a serialized file \( f \). Further, two additional options --cvStart <num1> and --cvEnd <num2> are supported to limit the search exploration range to the \( num1-st \) and \( num2-nd \) candidates from file \( f \). For two main reasons this existing option was not sufficient:

1. Writing all generated tests of a sequential execution to a file can be prohibitively expensive, e.g., for \( fnRedBlackTree(12) \) the size of the candidates.dat file on disk is about 7.63GB. However, to re-explore a range (for both SEQ-ON and MKorat), only \( start \) and \( end \) candidates are required. Hence, we overrode the --cvWrite command-line option to only writes the endpoints of desired ranges into a file.

2. Due to existence of invalid ranges, the distribution phase of MKorat may assign a worker several subranges to run.
number of equidistant candidates & \# of equidistant candidates \\
--invalid & \# of invalid ranges \\
--subranges & set of \langle start, end \rangle candidate vector pairs to run \\

<table>
<thead>
<tr>
<th>Option</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>--version</td>
<td>0: Korat, 1: SEQ-ON, 2: MKorat, 3: MKorat_exc</td>
</tr>
<tr>
<td>--equi</td>
<td># of equidistant candidates</td>
</tr>
<tr>
<td>--invalid</td>
<td># of invalid ranges</td>
</tr>
<tr>
<td>--subranges</td>
<td>set of \langle start, end \rangle candidate vector pairs to run</td>
</tr>
</tbody>
</table>

**Table 1: Korat extended command-line options**

For example, worker three in Figure 2 takes two subranges. We implemented an extension of Korat, namely MKorat\_exc which takes arbitrary number of subranges (\langle start, end \rangle candidate vector pairs), and explores only candidates within those given subranges.

For each Korat exploration, there are two special ranges which only contain invalid candidates, namely head and tail invalid ranges, described below. MKorat safely removes these two ranges in addition to the \( m \geq 1 \) parameter provided by the user:

- **head invalid range** is the range \([C_0, C_r]\) where \( C_0 \) is the initial candidate vector and \( C_r \) is the first valid candidate generated by the Korat search. For example, the range \([0, 366]\) is the head invalid range in Figure 2. Note in case that \( C_0 \) is equal to \( C_r \), the range \([C_0, C_r]\) contains no element, and head invalid range does not exist.

- **tail invalid range** is the range \([C_w, C_n]\) where \( C_n \) is the last candidate vector and \( C_w \) is the last valid candidate generated by the Korat search. For instance, \([896, 961]\) is the tail invalid range in Figure 2. In case \( C_w \) equals to \( C_n \), the range \([C_w, C_n]\) is empty and tail invalid range does not exist.

### 3.3 Implementation

Table 1 shows the new run-time options we introduced in our MKorat framework. Option --version chooses between the 4 techniques implemented within Korat framework, namely: the original Korat, SEQ-ON, MKorat, and MKorat\_exc. Next, option --equi determines the number of equidistant candidates for SEQ-ON and MKorat. Note that this number cannot be greater than the total number of explored candidates. Therefore, our implementations will use the minimum of the two numbers. Option --invalid is the number of invalid ranges MKorat considers. Similar to the --equi option, if this option exceeds the total number of invalid ranges, the minimum of the two values will be selected. Finally, option --subranges specifies subranges \( \langle \text{start, end} \rangle \) candidate pairs to run. Table to the right shows which options are required for each Korat extension we used in our study.

### 3.4 Properties

Given a Korat search problem (repOK and finitization), we define the Reduction achieved by MKorat as follows:

\[
\text{Reduction} = \frac{\text{\# of invalid candidates MKorat prunes}}{\text{\# of candidates Korat explores}}
\]

The denominator of the equation above is the total number of candidates Korat explores in a sequential run, which is the same number workers re-explore in SEQ-ON algorithm. The numerator is the number of invalid candidates MKorat prunes for future runs. Given a large enough \( m \geq 1 \) parameter, MKorat can achieve the Reduction\_max by pruning all existing invalid ranges from future executions. MKorat has the following properties:

1. Maximum candidates re-explored by a single worker, is at most equal to the number explored by a worker in SEQ-ON algorithm, because MKorat assigns each worker a subset of its original equi-distant range.
2. By definition of Reduction, the larger the number of explored invalid candidates are (compared to the valid explored candidates), the better MKorat is expected to perform. As an extreme case, for the constant returning repOK, i.e, return true or false, Reduction\_max will be 0% and 100% respectively.
3. The number of valid instances Korat finds for a given subject is an upper bound on the number of invalid ranges that subject can have.

### 4 EVALUATION

We evaluate the effectiveness of MKorat, on a suite of standard subjects shipped with Korat and used in prior studies. This section describes the experiment procedure we designed to answer the following questions:

Q1. Can MKorat achieve Reduction\_max?
Q2. How does the number and distribution of valid candidate vectors affect MKorat reduction?
Q3. What are the practical benefits of MKorat in terms of execution time and required computational resources for sequential and parallel settings?

#### 4.1 Study

Table to the right shows the 6 subjects used in our study, which are taken from Korat’s open-source repository. Prior studies used similar subjects in their evaluation [3, 26, 29]. Due to the bounded exhaustive nature of Korat search, running these subjects does not scale for large finitization values. For instance, given finRedBlackTree(12) for the red-black tree example discussed in Section 2, Korat explores 205,512,574 candidates in 4 minutes and finds 1,296 valid structures. We evaluated the effectiveness of MKorat for each subject, and compared it with the original SEQ-ON algorithm discussed in Section 3.1, with respect to the reduction definition in Section 3.4.

#### 4.2 Results

Table 2 shows basic information obtained by Korat execution for our 6 subjects. This table includes the number of candidates explored, valid instances found, and number of invalid ranges for 5 different finitizations. Recall from section 3.2 that MKorat safely removes the head and tail invalid ranges from the explored range; hence, the number of invalid ranges in table 2 excludes these two ranges (All 6 subjects across different finitizations had head and tail invalid ranges). As shown in Table 2, the number of candidates...
Table 2: Number of candidates explored, valid instances found, and invalid ranges explored by Korat.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Invalid ranges</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT</td>
<td>256 512 768 1024</td>
</tr>
<tr>
<td>BH</td>
<td>12.77 34.21 45.76 45.84 46.10 46.79</td>
</tr>
<tr>
<td>DLL</td>
<td>0.92 0.92 0.92 0.92 0.92 0.92</td>
</tr>
<tr>
<td>RBT</td>
<td>48.34 58.45 69.22 99.98 99.98 99.98</td>
</tr>
<tr>
<td>ST</td>
<td>0.17 0.43 1.40 5.01 18.81 72.54</td>
</tr>
<tr>
<td>SLL</td>
<td>0.22 0.50 1.45 4.54 13.84 38.79</td>
</tr>
</tbody>
</table>

Table 3: MKorat Reduction [%] for Fin = 8.

<table>
<thead>
<tr>
<th>Finitization</th>
<th># of</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explored candidates</td>
<td>16</td>
<td>245</td>
<td>3653</td>
<td>54418</td>
<td>815100</td>
<td></td>
</tr>
<tr>
<td>Valid candidates</td>
<td>2</td>
<td>18</td>
<td>132</td>
<td>1430</td>
<td>16796</td>
<td></td>
</tr>
<tr>
<td>Invalid ranges</td>
<td>1</td>
<td>13</td>
<td>131</td>
<td>1429</td>
<td>16795</td>
<td></td>
</tr>
<tr>
<td>Explored candidates</td>
<td>58</td>
<td>1666</td>
<td>42815</td>
<td>1323194</td>
<td>150727471</td>
<td></td>
</tr>
<tr>
<td>Valid candidates</td>
<td>6</td>
<td>120</td>
<td>7602</td>
<td>603744</td>
<td>117157172</td>
<td></td>
</tr>
<tr>
<td>Invalid ranges</td>
<td>3</td>
<td>25</td>
<td>941</td>
<td>33555</td>
<td>6628009</td>
<td></td>
</tr>
<tr>
<td>Explored candidates</td>
<td>27</td>
<td>94</td>
<td>776</td>
<td>17166</td>
<td>562823</td>
<td></td>
</tr>
<tr>
<td>Valid candidates</td>
<td>3</td>
<td>37</td>
<td>674</td>
<td>17007</td>
<td>562595</td>
<td></td>
</tr>
<tr>
<td>Invalid ranges</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Explored candidates</td>
<td>40</td>
<td>961</td>
<td>16487</td>
<td>322806</td>
<td>7530712</td>
<td></td>
</tr>
<tr>
<td>Valid candidates</td>
<td>2</td>
<td>8</td>
<td>20</td>
<td>64</td>
<td>260</td>
<td></td>
</tr>
<tr>
<td>Invalid ranges</td>
<td>1</td>
<td>7</td>
<td>19</td>
<td>63</td>
<td>259</td>
<td></td>
</tr>
<tr>
<td>Explored candidates</td>
<td>22</td>
<td>875</td>
<td>45233</td>
<td>2606968</td>
<td>155455872</td>
<td></td>
</tr>
<tr>
<td>Valid candidates</td>
<td>2</td>
<td>14</td>
<td>132</td>
<td>1430</td>
<td>16796</td>
<td></td>
</tr>
<tr>
<td>Invalid ranges</td>
<td>1</td>
<td>13</td>
<td>131</td>
<td>1429</td>
<td>16795</td>
<td></td>
</tr>
<tr>
<td>Explored candidates</td>
<td>17</td>
<td>139</td>
<td>2194</td>
<td>52567</td>
<td>1702171</td>
<td></td>
</tr>
<tr>
<td>Valid candidates</td>
<td>2</td>
<td>15</td>
<td>203</td>
<td>4140</td>
<td>115975</td>
<td></td>
</tr>
<tr>
<td>Invalid ranges</td>
<td>1</td>
<td>14</td>
<td>202</td>
<td>4139</td>
<td>115974</td>
<td></td>
</tr>
</tbody>
</table>

The 3D plots in Figure 6 show how the number of invalid ranges and finitization sizes (Fin) can affect reduction achieved by MKorat for all 6 subjects. Table 3 (discussed earlier) is basically an snapshot of Figure 6 for Fin = 8. By definition (Section 3.4), higher Reduction can be achieved for subjects with smaller ratio of valid to explored candidates. As shown in Table 2, RBT has the smallest ratio of valid to explored candidates for each finitization. Figure 6 shows that higher Reduction is achieved for RBT compared to the other subjects and this reduction is reached for smaller values of invalid ranges maintained by MKorat. Unlike, RBT, DLL has the highest ratio of valid to explored candidates and the reduction reached for this subject is smaller than others. This observation is noticeable in Table 3 as well for these two subjects.
Table 4: The min, max, and total execution time (in sec) for (1) MKorat used in sequential settings (1 worker) compared to Korat and (2) MKorat used in distributed settings (2, 8, and 32 workers) compared to SEQ-ON. For each subject the user provided upper bound on number of invalid ranges (IRs), \( m \) in Figure 4, is a percent of number of valid candidates for that subject(\( \text{Fin} \)).

<table>
<thead>
<tr>
<th>Workers</th>
<th>IRs [%]</th>
<th>Subject(( \text{Fin} ))</th>
<th>BT(12)</th>
<th>BH(9)</th>
<th>DLL(11)</th>
<th>RBT(10)</th>
<th>ST(9)</th>
<th>SLL(11)</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>min</td>
<td>max</td>
<td>sum</td>
<td>min</td>
<td>max</td>
<td>sum</td>
<td>min</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>10.76</td>
<td>26.56</td>
<td>32</td>
<td>17.28</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>0.43</td>
<td>0.89</td>
<td>0.65</td>
<td>6.90</td>
<td>21.60</td>
<td>28.50</td>
<td>11.80</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td>0.87</td>
<td>1.74</td>
<td>2.61</td>
<td>11.20</td>
<td>22.40</td>
<td>33.60</td>
<td>17.12</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td>1.72</td>
<td>3.48</td>
<td>5.20</td>
<td>15.12</td>
<td>30.24</td>
<td>45.36</td>
<td>27.04</td>
</tr>
<tr>
<td>32</td>
<td></td>
<td></td>
<td>3.48</td>
<td>6.96</td>
<td>10.44</td>
<td>31.24</td>
<td>62.48</td>
<td>93.72</td>
<td>49.60</td>
</tr>
</tbody>
</table>

Table 5: Total number of candidates explored by SEQ-ON and the percent of those candidates MKorat prunes.

Q2. How does the number and distribution of valid candidate vectors affect MKorat reduction?

MKorat achieves higher Reduction on subjects where the majority of explored candidates are invalid. As an extreme case, a repoK method returning a constant, can achieve 0% or 100% reduction for the constant values true and false respectively (Validated on two constant returning methods). Table 4 shows the minimum, maximum, and total execution time (in seconds) for 4 different number of workers. Specifically, 1 worker is used for the sequential setting comparing Korat with MKorat, while 2, 8, and 32 workers indicate a parallel setting using SEQ-ON and MKorat techniques). For each subject, we chose the largest finitization for which a sequential execution of Korat terminated within 30 seconds. Further, the number of invalid ranges MKorat maintains is provided based on a percent of number of valid candidates explored by Korat for a given subject and Fin. Note that the number of valid candidates found for a given subject is an upper bound on the number of invalid ranges that subject may have (Section 3.4).

Our results show that for both sequential and parallel settings, MKorat can speedup the test generation. For example, the first two rows in Table 4 show that given 1 worker, when only 4% of test input pairs are maintained, the re-generation of test inputs becomes 18% faster on average (9.41 sec compared to 11.10 sec). Further, as the percent of user-provided invalid ranges increase to 32%, the re-execution becomes 51% faster on average (7.34 sec as opposed to 11.10 sec).

Recall from Figure 6 and Table 3 that the increase in number of invalid ranges \( m \) maintained by MKorat, affect the Reduction differently for different subjects. Table 4 validates the same trend of growth we expected for each subject. For instance, DLL had the lowest growth of Reduction among other subjects (due to having only two small head and tail invalid ranges); the 3 columns for DLL in Table 4 show that the Reduction achieved for this subject is almost agnostic to the increase of \( m \) invalid ranges. RB7, on the contrary, had the highest rise in Reduction as \( m \) increased (Figure 6); this outperformance is also evident from Table 4. The effect of increasing \( m \) on speedup for the other 4 subjects, is smaller than RB7 and larger than DLL, which is in alignment with the trend observed earlier.

Besides saving in execution time, MKorat can save on the number of workers required. Specifically, if each candidate of an equidistant range \( e \) belongs to an invalid range that MKorat maintains, then no worker would be assigned to re-explore range \( e \). We observed this case for RB7 with 8 and 32 workers, when 4% (or above) test input pairs are maintained, MKorat requires 5 (37.5% fewer
of execution time) and observed that their functional behavior remained the same across two versions. Further, we placed assertions at certain points in our scripts to perform some sanity checks. For instance, the script which distributed subranges (obtained by MKorat) on workers for re-exploration (in Table 4), ensured that per each MKorat execution, the total number of valid instances found across all workers, is equal to the ones found using a sequential execution of Korat. Further, to increase the confidence in our scripts, we reviewed our code, tested it on a number of subjects manually, and inspected several results. To reduce noise and get more consistent numbers for Table 4 (which contain execution times), we measured the values several times and reported the average.

**Construct:** To find equi-distant candidates (Table 4) we used \( m = 2048 \) as maximum number of workers maintained by SEQ-ON algorithm [26]. We chose this large enough value to form evenly distributed subranges for distribution among up to 32 workers used in our study. For each subject in Table 4, we considered different number of invalid ranges (equal to a percent of number of valid test inputs found for that specific subject), to have a more meaningful and fair comparison between the execution times among different subjects. For Figure 6, we considered a wide range of finalizations and invalid ranges to observe the unique trend in Reduction increase for each subject.

5 RELATED WORK

This chapter presents related work on parallel analysis for systematic testing. Specifically, we consider two approaches for testing sequential programs, including one black-box testing technique, namely Korat [3], and one white-box testing technique, namely symbolic execution [22], and one approach, namely model checking [6], for testing multi-threaded programs. Moreover, we consider incremental analysis techniques that re-use results from previous runs.

5.1 Parallel Korat

The idea of parallel test generation and execution in the context of Korat was introduced by Misailovic et al. [26]. The idea of invalid ranges is rooted in their discussion on potential optimizations [26] where they observe the potential usefulness of creating sub-ranges that start and end at valid candidates. Our technique, MKorat, builds on this observation.

PKorat [29] introduced a different approach for parallel test generation using Korat. The key idea in PKorat is to explore Korat’s non-deterministic field assignments in parallel. Thus, PKorat does not require a previous execution of the Korat search but can still explore the space of candidate structures in parallel. However, re-running PKorat in the online test generation setting does not utilize any information about any previous execution of Korat; specifically, re-running PKorat does not utilize invalid ranges and re-explores all candidates that sequential Korat explores by default. While the original PKorat technique was defined for execution for a cluster of traditional computing platforms, recent work specialized PKorat for modern GPU’s [27]. Our approach is orthogonal to PKorat and can be integrated with PKorat. For example, PKorat can be used to explore each range that our approach creates based on the first execution of Korat search.

| Construction: | To find equi-distant candidates (Table 4) we used \( m = 2048 \) as maximum number of workers maintained by SEQ-ON algorithm [26]. We chose this large enough value to form evenly distributed subranges for distribution among up to 32 workers used in our study. For each subject in Table 4, we considered different number of invalid ranges (equal to a percent of number of valid test inputs found for that specific subject), to have a more meaningful and fair comparison between the execution times among different subjects. For Figure 6, we considered a wide range of finalizations and invalid ranges to observe the unique trend in Reduction increase for each subject.

5 RELATED WORK

This chapter presents related work on parallel analysis for systematic testing. Specifically, we consider two approaches for testing sequential programs, including one black-box testing technique, namely Korat [3], and one white-box testing technique, namely symbolic execution [22], and one approach, namely model checking [6], for testing multi-threaded programs. Moreover, we consider incremental analysis techniques that re-use results from previous runs.

5.1 Parallel Korat

The idea of parallel test generation and execution in the context of Korat was introduced by Misailovic et al. [26]. The idea of invalid ranges is rooted in their discussion on potential optimizations [26] where they observe the potential usefulness of creating sub-ranges that start and end at valid candidates. Our technique, MKorat, builds on this observation.

**PKorat** [29] introduced a different approach for parallel test generation using Korat. The key idea in PKorat is to explore Korat’s non-deterministic field assignments in parallel. Thus, PKorat does not require a previous execution of the Korat search but can still explore the space of candidate structures in parallel. However, re-running PKorat in the online test generation setting does not utilize any information about any previous execution of Korat; specifically, re-running PKorat does not utilize invalid ranges and re-explores all candidates that sequential Korat explores by default. While the original PKorat technique was defined for execution for a cluster of traditional computing platforms, recent work specialized PKorat for modern GPU’s [27]. Our approach is orthogonal to PKorat and can be integrated with PKorat. For example, PKorat can be used to explore each range that our approach creates based on the first execution of Korat search.

| Table 5 shows the number of candidates explored by SEQ-ON and the percent of those candidates MKorat explored for the same subjects, finalizations, and invalid ranges reported in Table 4. As shown, RBT achieved the highest Reduction (76.12%) across all subjects for any percent of invalid ranges used, which explains why MKorat performed better on RBT (execution times shown in Table 4). This observation was also expected based on the Reduction achieved for RBT in Figure 6. Further, BH and DLL subjects achieve their Reduction_{max} (25.74% and 0.01% respectively) for all 4 values of invalid ranges used in our study. For these two subjects, we observed that 4% of the number of valid candidates is a larger value than their total number of invalid ranges maintained by MKorat. Therefore, providing any percent of invalid ranges larger than 4% (of valid candidates) is not expected to affect the Reduction or execution time of BH and DLL. This observation justifies why given the number of workers, the min, max, and total execution time of MKorat for BH and DLL (in Table 4) stays (almost) the same.

**Q3:** What are the practical benefits of MKorat in terms of execution time and required computational resources for sequential and parallel settings?

| MKorat speeds up the minimum, maximum, and total worker execution time by up to 2.82X in the sequential setting (1 worker) and up to 446X, 1.86X, and 3.04X for the distributed setting with up to 32 workers. In the distributed setting, for subjects with small ratio of valid to explored candidates, like RBT, MKorat can provide up to 56.25% savings in number of physical workers required for re-exploration.

**Execution platform:** We obtained all data on a dedicated cluster in which each node has 16-core 2.7 GHz Intel Xeon CPU E5-2680 with 32GB of RAM, running CentOS release 6.8 (Final). We used Oracle Java: 1.8.0_25. Each MKorat was run on a single physical node (on a separate JVM) provided by -Xms2g -Xmx3g command line options.

4.3 Threats to Validity

**External:** The subjects used in our study may not be representative. To mitigate this threat, we considered 6 subjects shipped with Korat source code that vary in code size, complexity of rep0K, number of explored and valid candidates. Some of these projects have been also used in prior studies on Korat. Our results may vary for different finalizations and number of invalid ranges maintained. Exploring all the combinations was not feasible. To mitigate this threat, we considered several combinations to show the existing relation between different values of parameters. Further, the finalizations and number of equi-distant candidates considered in our study is on a par with prior work [3, 26, 29].

**Internal:** Korat, implementation of MKorat, and our automation scripts may contain bugs that can impact our conclusions. We are mostly confident in the correctness of Korat, as it is a robust tool used in several prior studies. To increase the confidence in our scripts, we developed core parts of our technique twice following (1) an efficient approach, and (2) a less efficient technique (in terms of execution time) and observed that their functional behavior remained the same across two versions. Further, we placed assertions at certain points in our scripts to perform some sanity checks. For instance, the script which distributed subranges (obtained by MKorat) on workers for re-exploration (in Table 4), ensured that per each MKorat execution, the total number of valid instances found across all workers, is equal to the ones found using a sequential execution of Korat. Further, to increase the confidence in our scripts, we reviewed our code, tested it on a number of subjects manually, and inspected several results. To reduce noise and get more consistent numbers for Table 4 (which contain execution times), we measured the values several times and reported the average.
5.2 Parallel symbolic execution

ParSym [30] applies the PKorat approach to symbolic execution — a classic program analysis based on systematic exploration of the program’s bounded execution paths. Simple Static Partitioning [33] for parallel symbolic execution first performs a shallow depth execution to build a set of preconditions based on the number of available workers who perform deeper exploration with respect to their individual preconditions. The parallel symbolic execution tool Cloud9 [4] embodies a production quality infrastructure based on load balancing.

Ranged symbolic execution [31] defines ranges for symbolic execution and uses them for distributing the symbolic exploration of bounded execution paths; each range is defined by a pair of in-order concrete inputs where the first input represents the path where symbolic execution starts and the second input represents the path where symbolic execution ends; moreover, work stealing is used for dynamic load balancing.

Most recent work by Qiu [28] introduces the idea of feasible ranges for succinctly memoizing symbolic execution results where the path conditions for all paths in a feasible range are satisfiable. Our idea of invalid ranges for Korat is inspired by Qiu’s idea of feasible ranges for symbolic execution and complements it. We could extend our work and support feasible ranges for Korat, so the cost of running it to generate valid inputs in a feasible range is reduced; for example, any candidate within a feasible range is known to be valid and therefore its validity does not need to be checked again; however, repOK may still need to be partially (and in some cases fully) executed on it to determine what the next candidate (which is also known to be feasible) is. Likewise, we could introduce the use of invalid ranges in symbolic execution.

5.3 Parallel model checking

Funes et al. [12] introduced the idea of ranging for software model checking using Java PathFinder (JPF) [37], an explicit state model checker; specifically, the exploration by the model checker is ranged by a pair of in-order paths that define the start and end of the model checking run. Previous work on parallel randomized state space search used multiple randomly generated start configurations for JPF and ran them in parallel with the expectation that one of them would find an erroneous state faster than the sequential run of the model checker [9]. One of the earliest techniques for parallel search for explicit state checking was parallel Merg, introduced by Stern and Dill [34], and shown to provide approximately linear speedups. Swarm verification [19] shows how to leverage multi-core computation platforms in the context of the SPIN model checker [18].

5.4 Incremental analysis

A number of incremental analyses re-use results from previous runs to optimize subsequent runs, e.g., for test generation [17, 35], symbolic execution [15, 36, 39], and model checking [2, 23, 32, 38]. The key difference between our approach and previous work is to reuse state-space exploration results, specifically about consecutive invalid candidates, to optimize constraint solving. Our approach shares the spirit of incremental SAT and conflict-driven clause learning [10] but works at a very different level (Java predicates versus CNF formulas).

6 CONCLUSION

This paper introduced a novel approach to reduce the cost of systematic testing using the Korat approach in certain application scenarios. Our key insight is that sometimes Korat’s backtracking search is repeated over the same state space across separate runs of Korat, and an earlier run of the search can be summarized to more efficiently perform a later run. We introduced the idea of invalid ranges which succinctly encode parts of the exploration space, which do not contain any valid inputs but have to be explicitly explored by the Korat search since it is unable to prune them. Our approach directly prunes these parts in a future run of Korat over the same input space. We developed our approach for two settings: a sequential setting where the Korat search is run using one worker (i.e., processing unit), and a parallel setting where the Korat search is distributed to several workers. In the parallel setting, we build on a previous technique for parallel Korat, namely SEQ-ON, and integrate invalid ranges within it. An experimental evaluation using a suite of standard subjects shows the efficacy of our approach.

Acknowledgments. We thank Ahmet Celik, Milos Gligoric, Rui Qiu, and Marko Vasic for their constructive comments on this work. This research was partially supported by the US National Science Foundation under Grants Nos. CCF-0845628 and CNS-1239498.

REFERENCES
