2014-07-01

# Slice—n—Dice Algorithm Implementation in JPF

Eric S. Noonan
*Brigham Young University - Provo*

# Slice–N–Dice Algorithm Implementation in JPF

Eric S. Noonan

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Eric Mercer, Chair
Jay McCarthy
Mark Clement

Department of Computer Science

Brigham Young University

July 2014

ABSTRACT

Slice–N–Dice Algorithm Implementation in JPF

Eric S. Noonan
Department of Computer Science, BYU
Master of Science

This work deals with evaluating the effectiveness of a new verification algorithm called slice–n–dice. In order to evaluate the effectiveness of slice–n–dice, a vector clock POR was implemented to compare it against. The first paper contained in this work was published in ACM SIGSOFT Software Engineering Notes and discusses the implementation of the vector clock POR. The results of this paper show the vector clock POR performing better than the POR in Java Pathfinder by at least a factor of two. The second paper discusses the implementation of slice–n–dice and compares it against other verification techniques. The results show that slice–n–dice performs better than the other verification methods in terms of states explored and runtime when there is no error in the program or little thread interaction is needed in order for the error to manifest.

Keywords: Verification, model checking, POR, Under-approximation

# ACKNOWLEDGMENTS

Thanks goes to my mother who took a chance on me and enrolled me in classes that would allow me to develop into a computer scientist later. Thanks goes to my brother Ryan for all his support in my various endeavors and helping me stay focused. Thanks goes to my younger brother Matthew who helped me get out and have fun when things were hard.

Last but not least, thanks goes to Dr. Eric Mercer, for believing that I had what it took to become a good graduate student and constantly supporting me through everything, the highs, the lows, and the occasional frustration.

## Table of Contents

## List of Figures

## List of Tables

# Chapter 1

# Introduction and Related Work

## 1.1 Introduction

There are many parallel programs being developed or that have been developed that operate on critical systems whose failure could lead to disastrous real-life consequences. Therefore it is necessary to ensure these systems have the correct behavior before using them. A program can be checked for correct behavior using a model checker. A model checker takes a program as input and checks for the reachability of error states in the program. JPF is a widely used model checker for Java programs. JPF is used by NASA to verify their critical systems. Model checkers like JPF have to deal with a problem called the state explosion problem.

The state explosion problem refers to the fact that exhaustively generating all possible states due to non-determinism in the scheduling or non-determinism in the input of the program yields an exponential number of states to explore. This number of states can be so large that it is not practical in terms of execution time or required memory to do an exhaustive exploration of all possible states in order to discover error states. There are numerous methods developed to solve the state explosion problem. One method of solving the state explosion problem is partial order reduction.

Partial order reductions (PORs) use the notion of dependence in order to determine redundant execution paths that generate the same global state with respect to properties that are being checked [10]. A transition is an operation that changes the global program state. Two transitions are dependent if they enable or disable each other or yield a different global result when executed in a different order. In contrast, two transitions are independent

if executing them in either order yields the same global program state. The basic idea behind a POR is to execute all the dependent transitions from a given state to generate a set of new states [13, 14, 20]. This exploration of all interleavings of dependent transitions ensures PORs are sound which means if there is an error, it will be found by a POR. The state space generated by a POR can be thought of as a graph where each node in the graph is a program state and edges leaving the state are transitions. If a node has more than one edge leaving it, then the two transitions from the node are dependent. This graph is explored in a depth first search fashion by POR methods. The depth of the search can be bounded by user in order to allow for the model checking of programs that do not terminate. As the POR generates states in the graph, details about the program state are available. These program states can be checked against constraints. If the model checker finds something that violates the specified constraints while performing a POR, then the program is provably wrong and needs to be corrected.

PORs reduce the number of explored states by only exploring one schedule of execution between independent operations. A POR still explores the whole state space of the system, which is still an exponential state space based on the number of dependent transitions. Exploring the entire unique state space of a program is a costly endeavor in terms of execution time and memory requirements even with a POR. This means model-checking using POR is impractical for larger parallel programs. For this reason, under-approximation methods have been developed [17].

There are many different kinds of under-approximation methods that have been published such as proof guided under-approximations, delay bounded scheduling and iterative context deepening [9, 17, 21]. Under-approximation methods leave a piece of the state space unexplored in order to allow for the model checking of larger systems than the systems that can be feasibly model checked by methods that explore the whole system's state space. The errors discovered by under-approximation methods are guaranteed to be actual errors in the system, but under-approximations are not guaranteed to find all errors. These undetectable

errors in larger systems may never be detectable if the program is too large to be model checked with a POR or other reduction method that explores the entire unique state space.

The slice and dice algorithm (slice–n–dice) was designed to address this issue of finding errors too large to be found by a POR [25]. The assumption behind the slice–n–dice algorithm is this: If an error may be represented as reachability to a target program location, then it is possible to determine if a property is violated by detecting reachability to the target location that represents the violated property.

slice–n–dice works by executing the thread that contains the target location in isolation. This thread is called the initial program slice. If the target location is not reachable by executing the initial slice, then another thread is added to the initial program slice that could modify data that affects control flow to the target location in the thread where the property violation may exist. This process of adding threads that could potentially help the program reach the error state is called refinement. After refinement, the slice is executed again multiple times, attempting all interleavings of the threads in the slice in order to detect the error. If the error is not found, then another refinement and round of execution is done. This process of slice construction and execution is repeated until the error is found or no additional refinements can be made.

The principle contribution of this research is the implemenation of slice–n–dice in JPF and an evaluation of slice–n–dice's effectiveness by comparing it against other verification techniques. This research discusses related work on solving state explosion, the implementation of a vector clock POR in JPF [22], the implementation of iterative context bounding in JPF, the first full implementation of the slice–n–dice algorithm and a comparison of slice–n–dice to other verfication methods, including POR and iterative context bounding. Slice–n–dice performed better than both types of POR it was compared to in all experiments at high thread counts by at least a factor of ten in terms of states explored. Slice–n–dice beat all verification methods by at least the same factor when there was no error present in the

system. Slice–n–dice only saw similar improvements against iterative context deepening when an error was present on one of the four benchmark types.

Section 2 of this chapter discusses related work on state explosion that is not described in other chapters. Chapter 2 contains a published paper describing the implementation of a vector clock POR used in the comparison of slice–n–dice to other verification techniques. Chapter 3 contains a paper yet to be published describing the implementation of iterative context bounding in JPF, the implementation of slice–n–dice and the results of comparing slice–n–dice to other verification methods. Chaper 4 discusses the conclusions drawn as a result of this research.

### 1.1.1 Thesis statement

The slice–n–dice algorithm, in the context of the JPF model checker, will explore fewer states on average before error discovery than other state of the art algorithms to mitigate state explosion in scheduling non-determinism including POR, heuristic guided search, and K-bounded delay, over a set of commonly accepted benchmark programs.

#### Deviations from thesis statement

We decided to compare slice–n–dice to iterative context bounding instead of delay bounded scheduling. Both iterative context bounding and delay bounded scheduling are K-bounded algorithms, so the comparison between slice–n–dice and either of the k-bounded algorithms would be about the same.

## 1.2 Related Work

The primary focus of this proposal is program verification. More specifically, program verification using model checking. Model checking is a way of exploring possible program states in search of an error. When model checking, two forms of non-determinism are explored in order to generate unique program states: non-determinism with respect to the input

to the program (data non-determinism) or non-determinism with respect to the way the program can be scheduled (scheduling non-determinism). In both cases, there are many possible program states that can result from non-determinism. The explosion of states caused by either form of non-determinism is referred to as the state explosion problem. The work in this thesis proposal is concerned with combating state explosion in model checking. This section briefly covers important technologies treating state explosion in data non-determinism (under/over approximations, bounded model checking, proving non-termination, symbolic/concolic execution) to give context to the extent of the state explosion problem, and then looks more particularly at scheduling non-determinism techniques which deal with the same class of problems the slice-n-dice algorithm addresses.

### 1.2.1 Over/under Approximations

When using counter-example guided abstraction refinement (CEGAR), a model of the program is constructed using an abstraction that represents an over-approximation of the system's actual state space [1, 2, 4]. This model is a collection of predicates that represent an abstraction of the input program. Once that model is constructed, the state space of the model is explored in search for errors. If no errors are found in the over-approximation, then the program is error-free. This method can detect erroneous errors because it starts out with an over-approximation of the actual state space [4]. To alleviate this, the model is refined whenever an error state is detected in order to determine whether or not the error was a legitimate error. This method is employed in two different model checkers, BLAST and SLAM [1, 2]. This method helps solve the state explosion problem because the abstraction contains fewer states than the actual program. This method only refines and expands the state space as needed when errors are found. As a response to the issue of detection of non-existent errors under-approximation methods were developed. One of these methods is predicate abstraction with under-approximation refinement [24].

In predicate abstraction with under-approximation refinement, predicates are assigned values differently than in CEGAR methods [24]. In a CEGAR based methods, some predicates may not evaluate to a particular value. In this case, CEGAR based methods assume the predicate has all possible values. In contrast, under-approximation refinement runs the program in order to determine a predicate value when the abstraction assigns a predicate an indeterminate value. This method trims states by not having to evaluate the result of the program under all possible predicate values for predicates of indeterminate value.

### 1.2.2 Bounded Model Checking

Property violations can be detected using constraint solving. The model checking method that detects property violations using constraint solving takes a program and compiles it into a SSA form. All of the loops and recursion in the SSA form are unwound to a certain user specified depth of looping or recursive calls. The resulting SSA form from the unwinding is translated into a SAT problem where each program definition is a specific variable in a SAT problem. The SAT problem is then checked for errors using an SMT solver [3, 6]. If the result shows an error state is satisfiable given the unwinding, then an error is reported [6].

The methods covered so far focused on model checking methods that followed a pattern. The first part of that pattern was to use the program to create an abstraction. The next part of that pattern was to prove properties about that abstraction. The final part of that pattern was to relate the properties proven back to the original program. In contrast, there are methods of model checking that focus on using the original program itself as the model to be checked. One of these methods that uses the program as a model and deals with data non-determinism is proving non-termination.

### 1.2.3 Proving Non-termination

Proving non-termination of a program is another form of verification. Recent advances have shown that proving termination or non-termination for some classes of programs is possible

and also solvable in a reasonable amount of time [5]. This is important because proving non-termination of a program can represent an error in a program, like deadlock or livelock. This type of verification is applied to systems that need to guarantee a service of some kind completes. Verification of this kind is performed on safety-critical programs or systems that need to be highly reliable, such as drivers in operating systems. Another form of model checking that deals with data non-determinism is symbolic/concolic execution methods.

### 1.2.4 Symbolic/Concolic Execution

There are methods of model checking that use the program itself as the model in order to explore the state space due to data non-determinism. One of the methods for doing this is concolic testing. In concolic testing, the program is given data for one run on the program. As the program is executed, the model checker keeps track of the path conditions that guide execution of the program. Once that is finished, the model checker uses the path conditions detected during exection to generate new data that will guide the model checker along a different branch in the program's internal structure [29]. The point of concolic testing is to automate test case generation as well as give the code full branch coverage in search for errors. Another method that operates independently of input data is generalized symbolic execution (GSE). In GSE the program's internal variables are represented as symbolic values, then as each branch is explored, it generates classes of input that cause errors based on the path conditions [19]. The last method exploits heap symmetry in order to identify equivalent states. If a heap has the same structure in two different states, then the states are considered equivalent and exploring from only one reduces the state space explored [18].

All of the types of model checking a state space covered so far relate to working with data non-determinism. Model checking the state space with regards to data non-determinism is orthogonal to model checking the state space with regards to scheduling non-determinism. This work deals with model checking with regards to scheduling non-determinism. The two types of model checking can be combined in order to model check multi-threaded programs

that take data input [28]. The rest of this related work deals with model checking a program with regards to scheduling non-determinism and how theses methods relate to slice-n-dice.

### 1.2.5 Under-approximated Heuristic Schedulers

Heuristic guided search uses heuristics to guide search of the state space when finding an error. States with a more favorable heuristic are chosen first in hopes of finding an error [8]. A simple heuristic is one that uses a boolean function, system state and valid transitions within the model. Using this information, two parts of the heuristic are computed. The first part is the number of transitions the model checker would have to take in order to violate the boolean function given the current state. The second part of the heuristic is the number of transitions the model checker would have to take out of the current state in order to satisfy the boolean function. The first part of the heuristic is used in order to check invariants on program states that represent properties that should always hold during runtime. If the model checker can use the first part of the heuristic to violate the property, then it knows an error state has been detected. The second part of the heuristic is used as a heuristic to guide the search towards assertion violations. If the model checker can guide the search in a way to violate an assertion, then an error state has been detected. Additional heuristics can be used to improve the search when using the heuristics based on boolean functions.

These heuristics are based on ideas about the structure of the state space being explored [16]. One of these heuristics is based on attempting to achieve full code coverage [16]. This heuristic gives states that are more likely to cover new branches a higher heuristic value. This helps guide the search to cover states it has not seen before, allowing errors in new branches of code to be detected. Another structural heuristic deals with thread interleaving. This heuristic gives a higher score to states that cause context switches where there is no explicit yield. This helps discover errors due to scheduling non-determinism quickly.

An additional structural heuristic helps the model checker ignore state space that is not valid in the program [15]. Many heuristic-guided searches deal with abstractions that can

be over-approximations of the actual state space of the program. These over-approximations introduce non-determinism that is actually not present in the native program. Therefore, structural heuristics have been developed. Structural heuristics are used to give favor to states that are choose-free. A state is choose-free if it is not generated due to non-determinism in the abstraction. Heuristic-guided searches have been improved by allowing the user specify error locations in addition to the normal static analysis to detect error locations. Such a method uses the heuristics discussed to try to force the model checker to any error location chosen by either the user or static analysis [26]. Heuristic guided search is faster than POR because it does not search the entire global state space like a POR, instead it tries to control the search in order to reach problem areas. Heuristic guided search is different from slice-n-dice in that it still considers the whole program when trying to guide the search to the error location, whereas slice-n-dice only considers the portion of the program that is in the current slice. Slice-n-dice also does not use heuristics to guide its search of the state space, it only interleaves relevant dependent operations when trying to reach the error state.

# Chapter 2

## Vector-Clock Based Partial Order Reduction for JPF


### Abstract

Java Pathfinder (JPF) employs a dynamic partial order reduction based on sharing and state hashing to reduce the schedules in concurrent systems. That partial order reduction is believed to be complete in the new version of JPF using search global IDs (SGOIDs) but does miss behaviors when SGOIDs are not employed. More importantly, it is not clear how such a dynamic partial order reduction, with or without SGOIDs, compares to other dynamic partial order reductions based on persistent sets, sleep sets, or clock vectors. In order to understand JPF's native dynamic partial order reduction better, this paper discusses an implementation of Flanagan and Goidefroid's clock vector partial order reduction in JPF. Then, the performance of JPF's native dynamic partial order reduction and the clock vector partial order reduction in JPF using SGOIDs will be compared in an effort to understand JPF's dynamic partial order reduction more fully. It was discovered that a clock vector POR always performs better in terms of runtime on the benchmarks chosen, and sometimes even better in terms of memory.

## 2.1 Introduction

Writing error-free concurrent programs is a difficult task. For this purpose, model checkers were developed. Model checkers execute the program and watch its state as it executes and look for errors in the states generated as the program executes. For concurrent programs, model checkers need to verify that all combinations of states in each of their individual components do not yield a global error state (such as a deadlock). Because not all combinations of states in each component are actually relevant when model checking a parallel program, partial order reductions were developed.

Java Pathfinder (JPF) is a model checker developed by NASA with the aim of verifying Java programs. JPF also has the ability to model-check concurrent programs with its own partial order reduction. This partial order reduction works using preemptive sharing detecting with Search Global Object Ids (SGOIDs). SGOIDs are used to associate objects with all of the threads that have accessed them over the course of the search. The authors have not found published research describing JPFs partial order reduction. It is believed that it explores more states than is necessary and the authors have not found published research for understanding its completeness. It is sound because it only executes reachable states. The major contributions of this paper are: (i) an implementation of Flanagan and Goidefroid's clock vector partial order reduction in JPF [13]; and (ii) a comparison between JPF's native POR and the JPF clock vector partial order reduction implementation.

## 2.2 JPF's POR Overview

Figure 2.1 gives an outline of how JPF's POR works. The following definitions are useful for understanding it:

- o refers an object and its SGOID.
- The accessed(o) function returns a set of SGOIDs corresponding to the threads that accessed the object referenced by o.

- The enabled($s$) function returns all the enabled threads in state $s$.
- The *nextins(p)* function returns the next instruction to be executed be the thread p.

From lines 6-10 it is apparent that JPF ignores all instructions that are not scheduling relevant. On lines 7 and 13, GETFIELD and PUTFIELD are the only instructions being considered "POR-relevant." In summary, JPF's POR executes instructions until it hits a POR-relevant instruction or a thread start or end instruction. Once JPF's POR reaches one of those instructions it stops. If it is dealing with a thread start instruction, it executes that instruction and creates a state from which it explores all active threads from their current state (lines 26-27), including the new thread. If the algorithm reaches a thread terminate instruction, all enabled threads are explored from that state (lines 26-27).

```
00:   S = {};
01:   s0.backtrack = Thread0
02:   s0.done = ∅
03:   S = S.push(s0)
04:   while(!S.empty()) {
05:     s = S.peek()
06:     if(∃ p ∈ (s.backtrack \ s.done) {
07:       while(nextins(p) != (threadstart || threadterminate)
07.1:       && nextins(p) is not a POR-relevant instruction) {
08:         if s is an error state, break and report
09:         s = s.execute(nextins(p))
10:       }
11:       if(nextins(p) has not been marked as executed) {
12:         mark nextins(p) as executed once
13:         if(nextins(p) is a POR-relevant instruction) {
14:           o = object nextins(p) operates on
15:           accessed(o) = accessed(o) ∪ p
16:           if(|enabled(s) ∩ accessed(o)| >= 2) {
17:             s' = s
18:             s'.done = ∅
19:             s'.backtrack = enabled(s)
20:             S.push(s')
21:           } else goto 9
22:         }
23:       } else {
24:         s.done = s.done ∪ p
25:         s' = s.execute(nextins(p))
26:         if(nextins(p) is (threadstart || threadterminate) ) {
27:           s'.backtrack = enabled(s')
28:         } else goto 8
29:         S.push(s')
30:       }
31:     } else {
32:       S.pop()
33:     }
34:   }
```

Figure 2.1: JPF's POR.

If JPF's POR reaches a POR relevant instruction, it marks the shared object the instruction accesses as being accessed by the current thread (line 15). Next, it checks if there are any enabled threads that also accessed the object (line 16). If there are, all enabled threads are added to the current state's backtrack set as potentially scheduling relevant (lines 16-19), otherwise, JPF continues executing instructions until it reaches another scheduling relevant instruction (line 21, 6-10). This means that there is at least one run through the program where there is no interleaving on POR relevant instructions. This happens because threads are executed until completion, so after one thread finishes, the next thread executed

will not interleave even if it accessed the same object as the previous thread because the previous thread is not currently enabled (see line 16). This means that the completeness of JPF's POR relies critically on interleaving on started threads. When JPF backtracks to a point where it scheduled all threads because a new one started, it will remember the sharedness from the previous run when it executes the next thread and interleave on all shared accesses afterwards.

The first obvious problem with JPF's POR is in lines 16-19 where all enabled threads are added to the backtrack set of a given state regardless of whether the thread actually accessed the object being operated on in that state. There is also a problem with trying to preemptively detect sharedness and interleave on threads that share in lines 16-19. In short, it causes redundant schedulings. This will be discussed further in section 4.

```
00:   S = {};
01:   s0.backtrack = Thread0;
02:   s0.done = ∅;
03:   s0.L = {};
04:   s0.C = {};
05:   S = S.push(s0)
06:   while(!S.empty()) {
07:     let s = S.peek()
08:     if (∃ p ∈ (s.backtrack \ s.done) {
09:       while(nextins(p) != (threadstart || threadterminate) &&
09.1:       nextins(p) is not a POR-relevant instruction) {
10:         if s is an error state, break and report
11:         s = s.execute(nextins(p))
12:       }
13:       if(nextins(p) is a marked POR-relevant instruction)
14:         let o = α(nextins(p))
15:         let cv = max(C(p), C(o))[p := |S|]
16:         let s.C = s.C[p:=cv, o:=cv]
17:         let s.L = if nextins(p) is a release
18:                     L
19:                   else L[o:=|mathitS'|]
20:         goto 12
21:       }
22:       if (nextins(p) is a non-marked POR-relevant instruction) {
23:         o = object nextins(p) operates on
24:         accessed(o) = accessed(o) ∪ p
25:         mark nextins(p)
26:         if (|accessed(o)| >= 2) {
27:           let i = s.L(α(nextins(p)))
28:           if( i != 0 and i > s.C(p)(proc(Si))
29:             if (p ∈ enabled(pre(S, i)))
30:               pre(S, i).backtrack = pre(S, i).backtrack ∪ p
31:             else
32:               pre(S, i).backtrack = enabled(pre(S, i)))
33:         } else goto 12
34:       }
35:         s.done = s.done ∪ p
36:       if(nextins(p) is (threadstart || threadterminate)) {
37:         let s' = s.execute(nextins(p))
38:       } else let s' = s
39:       s'.done = ∅
40:       S.push(s')
41:       if(nextins(p) is (threadstart || threadterminate)) {
42:         s'.backtrack = enabled(s'))
43:       } else s'.backtrack = p
44:     } else S.pop()
45:   }
```

Figure 2.2: The clock vector POR.

## 2.3 Clock Vectors

A full description of a partial order reduction using clock vectors is in [13]. A brief description of key components is in this paragraph. A program is comprised of a finite set $P$ where individual members of $P$ are denoted by $p$. Individual members of $P$ can refer to a thread or a process. A clock vector $\mathrm{C}(p)$ is a way of tracking dependencies between the current thread or process states and transitions that have already occurred in the search. If

thread $p_i$ has a clock vector C($p_i$) = {$c_1$, ... , $c_m$}. Then $c_j$ is the index of last transition in the search space executed by thread $p_j$ that had to happen in order for thread $p_i$ to reach its current state. Similarly, clock vectors can be made for objects (denoted C($o$)). When a clock vector corresponds to an object, the clock vector is tracking dependencies of the accessed object's current state on transitions performed during the state space search by each of the threads or processes in the system. The pseudocode for a clock-vector partial order reduction implemented in JPF is given in Figure 2.2. Definitions for understanding the table are as follows:

- $S$ is a transition sequence represented by {$t_1$, ..., $t_m$} where $t_j$ refers to transition $j$ in the sequence.
- $C$ is a data structure for storing clock vectors.
- $L$ is an object that stores the last transition to access an object o. *L(o)* denotes the index of the last transition in $S$ that accessed o.
- $s$ is a global state of the system.
- *last(S)* denotes a global state $s$ generated after the last transition in $S$ executed.
- $i$ refers to an index in $S$.
- $\alpha(t)$ returns a reference to the object that an instruction operates on.
- *proc($S_i$)* is the process or thread that executed transition $t_i$ in $S$.
- *pre($S$, $i$)* is the state $s$ of the system before transition $t_i$ was executed.
- *max(*C($p_i$), C($p_j$)*)* is the maximum of the two clock vectors C($p_i$) and C($p_j$) (the result is a clock vector where each index contains the maximum of that index in the two other clock vectors).

Keeping track of clock vectors for both the object and the process while taking the point-wise maximum of the two on line 15 of Figure 2.2 essentially means that a process inherits the maximum of all clock vectors of objects that it accesses with each clock vector's state corresponding to its state when the process accessed them. Clock vectors are used to determine if the dependent transition detected on line 27 was not a transition needed to

generate the current state. If the dependent transition was not needed to generate the current state, then the two dependent transitions can be co-enabled and interleaved as on lines 26-32. The interleaving is done by finding the last dependent transition and scheduling the current thread to execute from the state before the last transition that operated on the same object.

Each state stored on the stack in the pseudo-code corresponds to a choice generator. Every time JPF creates a choice generator that corresponds to a POR instruction being executed, the data structure corresponding to $L$ and $C$ is stored with it. Each choice generator actually stores two copies of $L$ and $C$. One corresponds to the transition being taken. The other corresponds to the transition not being taken. To make the algorithm equivalent, but easier to follow, that version of the data structure is calculated when the transition is actually taken on lines 13-21. A custom choice generator that allows for new threads to be added to its current choices was implemented in order to provide the functionality in lines 30 and 32.

When a thread start or thread terminate instruction are reached, we perform the same actions as JPF's original POR (Lines 41-42). The logic represented in lines 22-44 are captured by the interactions between schedulers, choice generators made by the scheduler and the virtual machine. A custom VM had to be implemented in order to perform the check on line 26 differently. The method in JPF's virtual machine class used to check if there are other enabled threads that accessed the object at the current state is only used by the POR, so by overriding this method we were able to perform the check the way we wanted to without having to modify classes for individual instructions that check for POR boundaries.

The clock vector algorithm executes in a very similar manner to JPF's POR algorithm. The check on line 26 of the clock vector algorithm is very similar to the check on line 16 of JPF's POR algorithm. The main difference is that the clock vector algorithm does not ensure that other threads are enabled before it does its logic for interleaving on lines 22-34. The way choice generators are made for thread start and stop is the same.

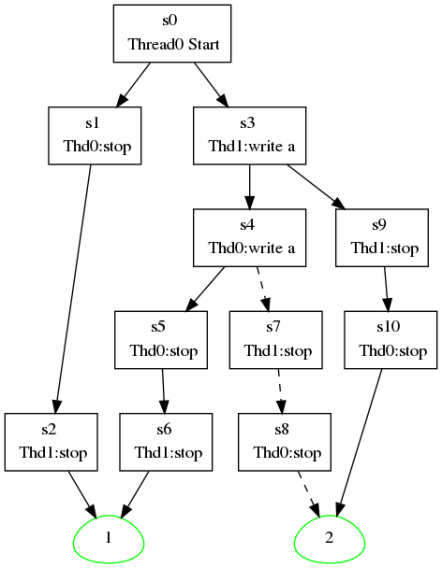


Figure 2.3: The JPF search space for the two PORs.

## 2.4 Example

The two approaches are illustrated using the following simple program that updates the global variable $a$:

$$\begin{array}{lll} & t_0 & t_1 \\ 0: & \text{Thread.start}(t_1); & a = 2; \\ 1: & a = 1; & \end{array}$$

The search space explored by JPF for this program (omitting initialization code) is shown in Figure 2.3. A non-end state is represented as a box. Each box contains a state label and the instruction executed that generated the state. Unique end states are represented as numbered green ovals. The dashed lines represent edges to states generated by execution of JPF's POR and those states were not visited by the clock vector algorithm.

JPF's POR algorithm begins by executing $Thread_0$ until it hits line 0 in the program. This corresponds to lines 0-10 in the algorithm. Lines 13-22 in the algorithm are skipped because a thread start is not a POR relevant instruction. Next, in lines 25-27 of the algorithm, a new state is generated with the started thread. This new state will explore all enabled threads (line 27) and means JPF created a choice generator for started thread. This new

state corresponds to state s0 in Figure 2.3. This new state gets pushed onto the stack in the algorithm on line 29. Now, the algorithm starts exploring the next state that is pushed onto the stack in the last run of the loop, the new state. $Thread_0$ is chosen to run again on line 6. When $Thread_0$ runs again, it executes its line 1 in the program. This causes the sharedness to be updated in line 15, but it jumps back to executing other instructions in line 21 because the access to "a" doesn't show it as being shared (line 16). After line 1 of $Thread_0$ is executed, the thread is terminated and state s1 is generated. Then $Thread_1$ is executed in a similar fashion. Again, no sharedness is detected on line 16 because $Thread_0$ is not currently running. At this point, the program is finished, so JPF backtracks to state s0. In the algorithm, state s2 and s1 are popped of the stack because they've exhausted all available thread in their respective backtracks sets.

Then, the algorithm starts off executing $Thread_1$ from s0 because $Thread_0$ has already been executed. Line 0 of $Thread_1$ is a POR relevant instruction. This time, the check on line 16 passes because the search history recorded "a" as being accessed by 2 threads on the previous execution of the program. This means that this state is saved and a choice generator is constructed with all running threads (lines 17-20). This is what generates state s3 in the figure. The state s4 is generated by choosing $Thread_0$ from the last choice generator and reaching line 1 in $Thread_0$. Line 1 is a POR-relevant instruction, so the algorithm saves the program state and executes all possible threads from that state. States s5, s6, s7, s8 and s9 are generated from threads terminating in the order they're executed.

We choose to discover sharing in the same manner as JPF's POR when performing the clock vector algorithm. This means that the clock vector algorithm generates states s0, s1 and s2 in the same manner as JPF's POR. The main difference happens after the backtrack up to state s0. Reaching the PUTFIELD instruction in line 0 of $Thread_1$ triggers the logic in lines 26-32. In this state, there are no clock vectors or operations on the shared object, so no interleave is calculated with a previous state. But on lines 38 and 43 a new state is generated with a choice generator that only initially contains the currently running thread

($Thread_1$) this choice generator corresponds to state s3. State s3 is the state just before line 0 is executed in $Thread_1$.

Then, as $Thread_1$ is executed further, its clock vectors are updated with the operation on the variable "a" (lines 13-21). $Thread_1$ finishes execution to generate state s9. Then, $Thread_0$ is executed. Line 1 on $Thread_0$ is a POR-relevant instruction. This means that lines 26-32 of the algorithm are triggered. $L(a) = 1$ (the index of the transition that operated on shared variable "a"). The clock vector for the current process $Thread_0$ is ¡0,0¿ because no POR-relevant instructions have been executed in $Thread_0$ yet. The check on line 28 returns true because $Thread_0$'s state is not dependent on any operations that have occurred in $Thread_1$. On line 29, it is determined that $Thread_0$ was enabled in the state before the last transition occurred that operated on "a." This causes $Thread_0$ to be added to the choice generator that corresponds to state s3 in the diagram. $Thread_0$ then creates a state corresponding to this transition about to operate on "a." This state is not on the graph, but there is no branching on that state because there are no other operations on "a" in this branch of the search. $Thread_0$ then completes execution and this particular run of the example program terminates.

Next, the algorithm pops all states off the stack generated before s3. From s3, $Thread_0$ is executed. $Thread_0$ reaches line 1. The clock vector algorithm again performs the computations on lines 26-32. Because state s3 corresponds to a state just before $Thread_1$ executes its line 0, there are no previous dependent operations to interleave with so no interleaving occurs. A state is generated on lines 38 and 43 with $Thread_0$ as the only choice in the choice generator. $Thread_0$ is then executed until completion to generate state s5. $Thread_1$ is then executed to completion with no interleaving because the write to "a" was already marked when s3 was created.

Looking at the state space of the example program in Figure 2.3 it becomes obvious that JPF's POR explores an unnecessary branch. This demonstrates preemptive sharing detection with interleaving guarantees the same ordering on a variable access occurring at

least one more time than needed. This expands the state space greatly when there are more accesses to the same variable from even more threads. This example shows why it is advantageous to perform interleavings based on the history of the search as opposed to preemptive sharing detection.

## 2.5 Results

This sections details the results of running benchmarks for JPF's native POR and a clock vector based POR algorithm. Both sets of experiments were run on a machine with an AMD phenom quad core processor with an approximate 1.8 Ghz speed and 8 GB of RAM. The Java virtual machine was allocated with 2 GB of memory. The custom VM for the clock vector POR was only used in experiments involving the clock vector POR. Two benchmarks are run using enabled randomization on all scheduling relevant choice generators. Average run time and average total number of states is recorded for three samplings from the distribution from the possible runs for both algorithms. The number of states varies slightly in the clock-vector POR based on the sharing detected on the random first run.

Table 2.1(a) shows the results of increasing the number of threads while doing various experiments on the cushion variable in the airline benchmark from Mercer and Rungta [27]. The table shows the results of increasing threads as well as how different values of the cushion variable play out in the runs. The table is organized such that threads increase from left to right and down. If the number of threads stays the same while reading cells in this manner, then the value of the cushion variable is increasing. The cushion variable increases the depth of the error. From the table it is apparent that JPF's native POR always takes longer to run and explores more states than the clock vector POR. An anomaly in both algorithms is the fact that the number of states decreases in both algorithms when the cushion is increased with four threads being run.

Table 2.1(b) shows the results on another benchmark written by Mercer and Rungta called reorder [27]. The strategy when doing these experiments on the reorder benchmark

| Airline | | | |
|---|---|---|---|
| JPF | threads | 3 | 4 |
| Native | cushion | 4 | 2 |
| POR | runtime(seconds) | 6 | 69.3 |
| | total states | 9025 | 307561 |
| | threads | 4 | 5 |
| | cushion | 4 | 1 |
| | runtime(seconds) | 75.7 | 587.7 |
| | total states | 281105 | 3966210 |
| JPF | threads | 3 | 4 |
| Clock | cushion | 4 | 2 |
| Vector | runtime(seconds) | 3 | 28 |
| POR | total states | 2786.7 | 107112 |
| | threads | 4 | 5 |
| | cushion | 4 | 1 |
| | runtime(seconds) | 28.3 | 154 |
| | total states | 87214 | 1050418 |

(a)

| Reorder | | | |
|---|---|---|---|
| JPF | setters | 1 | 2 |
| Native | runtime(seconds) | 1 | 2 |
| POR | total states | 285 | 3859 |
| | setters | 3 | 4 |
| | runtime(seconds) | 13.7 | 100 |
| | total states | 54555 | 532101 |
| JPF | setters | 1 | 2 |
| Clock | runtime(seconds) | 1 | 2 |
| Vector | total states | 181.33 | 2222.7 |
| POR | setters | 3 | 4 |
| | runtime(seconds) | 8.7 | 49 |
| | total states | 26576 | 228874 |

(b)

Table 2.1: Results from examples. (a) The Airline model. (b) The Reorder model.

was to increase the number of setter threads that cause the error while holding the number of threads that manifest the errors (checkers) constant at the value of one thread. This strategy was chosen because it would always increase the number of states explored in the full state space before the checker thread found the error. As can be seen from the chart, the clock vector POR performed better in terms of both time and total number of states visited.

## 2.6 Related Work

The dynamic partial order reduction implemented in this paper is based on a strategy of determining partial orders through collecting information about the state space as the search continues. The main other kind of partial order reduction is static partial order reductions. In static partial order reductions (SPOR), the program is analyzed at compile time to determine dependencies between operations [20]. Whereas in dynamic partial order reductions (DPOR) such as the one in this paper, the program determines dependence based on changes in program state as it is run [13]. There are various methods to do this. One of these methods uses persistent and sleep sets [13]. A persistent set is a set of transitions such that all transitions outside the set are independent of the transitions inside of it [14]. The algorithm in this paper is derived based on persistent sets [13]. The idea behind sleep sets is that they are sets of transitions that are independent with each other (exploring a schedule with an interleaving of two transitions in a sleep set yields the same global state). Once one interleaving of the independent transitions has been explored, it is not explored again in the opposite order [14]. The idea behind a DPOR using the sleep set and persistent set technique is to schedule every transition in a persistent set for each state and execute those schedules without repeating any interleavings in a sleep set more than once. If this sort of exploration is done, then it is guaranteed that all possible global results for a given program have been explored [13]. The primary advantage of a DPOR is that it has more information about the program available to it during execution even though it takes more memory than SPOR. A

dynamic partial order reduction was chosen for this paper because it is more powerful than an SPOR and JPF gave all of the relevant information needed in order to perform a DPOR.

## 2.7 Conclusions and Future Work

The clock vector based partial order reduction, when implemented in JPF is a more powerful partial order reduction than JPF's native POR. Even though it sometimes takes more memory, the clock vector POR visits fewer states and finishes execution faster than JPF's native POR and is therefore far more powerful. This is because JPF's native POR does not take into account whether or not a thread is shared before adding it to the choice generator for POR transition boundaries.

It is believed that JPF adds too many threads to the start and end CGs for threads, further work needs to be done to determine which threads do not need to be added. These changes could yield even better results. Also, additional data structures could keep track of which threads start other threads so that when a thread is not enabled on a shared object access. This could be used to reduce the number of threads added to the shared object access choice generator on line 35 of the algorithm. This suggestion is made in the research containing the clock vector algorithm [13]. After these changes are made, a new version of the algorithm should be published along with a formal proof of the completeness and soundness of the algorithm. This will give future users of JPF a better understanding of what JPF's POR provides as well as how it works.

# Chapter 3

## Slicing and Dicing Bugs Against Other Verification Techniques


### Abstract

The slice–n–dice algorithm is an algorithm that under approximates the state space of the program by only executing the parts of the program that are relevant for reaching an error. The state space of the program is gradually expanded until the error is found. This paper presents a full implementation of slice–n–dice in Java PathFinder (JPF). This paper also contains a comparative study over a benchmark set with slice–n–dice, clock vector partial order reduction, JPF partial order reduction, heuristic guided search and iterative context bounding. Slice–n–dice only outperformed all of the other algorithms in the comparative study on parallel programs where a few non-specific threads were needed to reach the error state.

## 3.1 Introduction

Concurrent software systems often operate on critical systems whose failure could lead to disastrous real-life consequences. Model checkers are one type of tool used to find errors in concurrent systems. Java Pathfinder (JPF) is a model checker that takes a Java program in as input and outputs whether or not it found an error in the state space of the program. Model checkers like JPF explore the entire state space of the system. Model checkers that do this suffer from the state explosion problem, which put simply, means that exploring all possible states in a non-trivial concurrent system is intractable. The slice–n–dice algorithm is a way of dealing with the state explosion problem.

The assumption behind the slice–n–dice algorithm is this: If there is an error state that can be represented as the program reaching a certain location (such as an exception being thrown), then it is possible to determine if an error state is reachable by detecting reachability to the target location that represents the error state. Slice–n–dice works by executing the thread that contains the target location in isolation. This thread is called the initial program slice. If the target location is not reachable by executing the initial slice, then another thread is added to the initial program slice that could modify data that affects control flow to the target location in the thread where the property violation exists. This process of adding threads that could potentially help the program reach the error state is called refinement. After refinement, the slice is executed again multiple times, attempting all interleavings of the threads in the slice in order to detect the error. If the error is not found, then another refinement and round of execution is done. This process of expanding the set of threads to interleave on and execution is repeated until the error is found or no additional refinements can be made. In order to understand the effectiveness of slice–n–dice, it must be compared to other well-known forms of program verification techniques that manage state explosion.

One other method of managing state explosion is partial order reduction (POR) [10, 13, 14, 20]. PORs work by only interleaving on object accesses that generate unique

global states in order to generate all unique global states. In contrast, another model checking strategy, iterative context bounding does not explore all global states [21]. Iterative context bounding works by restricting the number of preemptive context switches in any given execution path to at most an input number called K. This method under approximates the state space of the system and can model check larger programs than POR, but unlike POR is not guaranteed to find an error if there is one. Heuristic guided search (HGS) is another way to find an error without exploring the whole state space of a program. In HGS, heuristics are used to guide the program's execution towards an error state and ignore states not useful for finding an error [8, 15, 16, 26].

In order to understand the efficacy of slice–n–dice, this paper compares it against mainstream ways of dealing with state explosion: POR, iterative context bounding and HGS. This paper discusses the implementation of iterative context bounding and slice–n–dice in JPF in order for this comparison to be performed. Previous papers describe the implementation of a vector clock POR in JPF [22] and the heuristic guided search [26]. The results of comparing slice–n–dice to a vector clock POR, JPF's POR, iterative context bounding and HGS were that slice–n–dice performed best against the other verification techniques when a non-specific few threads are needed to reach the error state.

The major contributions of this paper are:

- The first full implementation of the slice–n–dice algorithm.
- An implementation of iterative context bounding in JPF.
- Specific strategies used in order to implement a full version of the slice–n–dice algorithm. These strategies handle multiple target locations, cause the algorithm to execute more efficiently and deal with multiple threads capable of reaching a single target location.
- The first comparison of slice–n–dice to other mainstream reduction algorithms.

The rest of this paper is organized as follows: Section 2 describes the slice–n–dice algorithm. Section 3 describes the basics of how JPF works as a model checker for parallel

| int x = 0, z = 0 | | | |
|---|---|---|---|
| | $t_0$ | $t_1$ | $t_2$ |
| 0: | `x++` | `int h = 5` | `int i` |
| 1: | `if(x > 6){` | `int y = h + 1` | `for(i=0; i < 5; i++){` |
| 2: | `throw exception` | `x += y` | `z += x` |
| 3: | `}` | `y *= 2` | `}` |
| 4: | | `int m = x * y` | |
| 5: | | `if(m >= 84){` | |
| 6: | | `y *= 4` | |
| 7: | | `}` | |

Table 3.1: Example program

programs. Section 4 describes implementation details of the various algorithms in JPF and the specific strategies used to implement slice–n–dice. Section 5 reports and discusses the results of the algorithms on the various benchmarks. Section 6 discusses work related to this one. Section 7 includes conclusions as well as discussion on future work.

### 3.2 Slice–N–Dice Description

The slice–n–dice algorithm is an algorithm that under approximates the state space of the program by only executing the parts of the program that are relevant for reaching an error. The slice–n–dice algorithm takes two inputs: a program and a program location, that if reached, represents an error state. The output of the slice–n–dice algorithm is whether or not an error state is reachable.

To simplify the presentation, the following assumptions are made about the input program and target machine. Section 3.4.4 describes how the assumptions are relaxed in the actual JPF implementation:

- The input program must be a closed program that receives no input.
- The slice–n–dice algorithm operates on a machine where there is no dynamic creation of threads so that the threads in the program can be determined statically.

- All threads in the program are running from the initial program state.
- Each program location can only be executed by one thread.

The algorithm is more directly understood through example. Consider the program in Table 3.1 as input to the algorithm. In this example, there are three threads $t_0$, $t_1$ and $t_2$ that all in some way access global variable $x$. Thread $t_0$ is capable of throwing an exception on line 2. This is the target location passed in as input to the program. Thread $t_1$ mutates $x$ in a way that makes reaching the exception target in $t_0$ possible. Thread $t_2$ only reads the global variable $x$ and does not interact with $t_0$ in any way that aids in hitting the target.

The program and the target location line 2 of $t_0$, indicated by $t_0 : 2$, are passed in as input to the algorithm. The exception on $t_0 : 2$ is only reached if $x$ is above 6, so the target is control dependent on $t_0 : 1$. Further, the value of $x$ is modified at $t_0 : 0$, so the exception is data-dependent on that location. These locations, $t_0 : 2$, $t_0 : 1$, and $t_0 : 0$ together are a backwards slice of thread $t_0$ from the target location. The backward slice is an abstraction of the actual program. From the perspective of slice–n–dice, the program now only consists of the locations, connected by the control flow, in the slice. All other locations in the input program are effectively ignored.

As the slice–n–dice algorithm has abstracted the original input program to just the locations from the backward slice in thread $t_0$, it executes $t_0$ only to see if the exception is triggered, effectively ignoring all other threads. They are not allowed to run since to slice–n–dice, they are not relevant to throwing the exception. Figure 3.1 shows the result of this execution in the left-most branch. Each node in the graph is a line in the abstraction. The arrows are labeled with the thread that executes as well as the value of the variable of interest. From the graph, it is apparent that $t_0 : 2$ in the backwards slice is not reached.

The algorithm uses the unreached location $t_0 : 2$ in the backwards slice to perform further analysis on the input program to expand the abstraction to include other locations and threads that may assist in enabling the exception to be thrown. During the execution of $t_0$ in isolation, the value of $x$ was never above 6, making the target unreachable. So slice–n–dice

```
   Data: P, the input program, l_t, the target location
   Result: A notification telling the user whether or not the error state was found
 1 begin
 2 |  (s,L_s,T_s,T_a) = initialize(P,l_t);
 3 |  repeat
 4 |  |  L_r = explore(s,T_s,L_s,T_s);
 5 |  |  (L'_s, T'_s) = refine(restrict(L_s, T_s),L_r,T_a,T_s);
 6 |  |  if L'_s = L_s ∧ T'_s = T_s then
 7 |  |  |  report no error found and exit;
 8 |  |  end
 9 |  |  L_s = L'_s;
10 |  |  T_s = T'_s;
11 |  until true;
12 end
```

**Algorithm 1:** This algorithm is the driver of the slice–n–dice algorithm. Its name is *slice–n–dice*$(P,l_t)$.

looks for other locations that mutate $x$. This analysis initially yields $t_1 : 2$. Slice–n–dice constructs a backwards slice for $t_1 : 2$ in the same way it constructed a backwards slice for $t_0 : 2$. This new backwards slice initially contains $t_1 : 2$, $t_1 : 1$ and $t_1 : 0$ but $t_1 : 1$ and $t_1 : 0$ are not included because they do not perform operations on global variables. Slice–n–dice adds $t_1 : 2$ to the abstraction of the program originally generated to the target location. This new abstraction now contains threads $t_0$ and $t_1$ with their respective locations from the backward slices. The program is run again, this time interleaving both threads in the abstraction on the locations in the abstraction. The target is reached and a trace of that execution is found in Figure 3.1 in the right-most branch of execution shown in the figure.

A program state consists of the state of objects on the global heap and the individual states of the threads. The core parts of the slice–n–dice algorithm are in *slice–n–dice* (Algorithm 1), *explore* (Algorithm 2), and *refine* (Algorithm 3). The *slice–n–dice* portion of the algorithm is the top-level driver that makes calls to *explore* and *refine* until the error is found or no more refinement for reaching the error is possible. The *explore* portion of slice–n–dice traverses the state space of the program, using the abstraction to guide it. The *refine* portion expands the abstraction in order to make the error reachable if a call to *explore*

failed to find the error. Slice–n–dice also increases the set of threads interleaved on each program simulation by only one. For this purpose, the abstraction is broken into two pieces, the locations in the abstraction and the threads in the abstraction. The following definitions are useful for understanding those three algorithms that comprise the slice–n–dice algorithm:

- Let $P$ be the input program.
- Let $T_s$ refer to all threads in the abstraction.
- Let $T_a$ refer to all threads in the program.
- Let *enabled*($s$) return all threads that are enabled in program state $s$.
- Let $L_r$ be the set of all locations reached during execution of the input program.
- Let $L_s$ be the set of all locations in the abstraction.
- Let *restrict*($L, T$) return a set of locations $L_t$ which contains all locations in $L$ that are associated with threads in $T$.
- Let *initialize*($P$,$l_t$) return a tuple ($s$,$L_s$,$T_s$,$T_a$) where $s$ is the initial program state, $L_s$ is the relevant locations isolated in a backwards slice to the target location $l_t$, $T_s$ is a set of threads containing only the initial thread in the abstraction that may reach the target location and $T_a$ is all threads in the program, determined statically.
- Let *newLocs*($l_u$,$P$) return the set of locations generated from creating backwards slices on locations involved in manipulating control flow to unreached location $l_u$ in $L_s$.
- Let *canExecute*($t$,$L$) return true if thread $t$ is associated with some program location in $L$, a set of program locations otherwise, it returns false.
- Let *execute*($s$,$t$,$L_s$) return a pair of values ($s'$, $L_e$) where $s'$ is the program state generated by executing thread $t$ from state $s$ until a member $l$ of $L_s$ is reached or $t$ terminates and $L_e$ is the set of all locations reached by thread $t$.

```
Data: s a start state, T the set of threads to execute from s, L_s, the set of relevant
      locations and T_s, the set of all threads in the abstraction.
Result: L_r: the set of all locations visited during state space exploration of the
        threads in the abstraction of the original program
1  begin
2  |  L_r = ∅ foreach t ∈ T do
3  |  |  (s', L_e) = execute(s,t, L_s);
4  |  |  if s' is an error state then
5  |  |  |  report error and exit;
6  |  |  end
7  |  |  L_r = L_r ∪ L_e ∪ explore(s', enabled(s') ∩ T_s);
8  |  end
9  |  return L_r;
10 end
```

**Algorithm 2:** This algorithm encapsulates exploring the state space of the threads in the abstraction, using the identified relevant locations in the slice as scheduling points for the threads in the abstraction of the original program. Its name is *explore*($s$,$T$,$L_s$,$T_s$).

### 3.2.1 Example Algorithm Execution

The example program in Table 3.1 will be used again to illustrate how the individual slice–n–dice algorithm parts work. The program and the target location $t_0 : 2$ are passed in as input to *slice–n–dice*. The initial backwards slice created by *initialize*($P$,$l_t$) yields $t_0 : 0$, $t_0 : 1$ and $t_0 : 2$ of the example program as the initial locations in the abstraction ($L_s$). Thread $t_0$ is the first thread in the abstraction because because $t_0$ is the thread that may execute the target location ($T_s = \{t_0\}$). Next, *slice–n–dice* calls *explore* with the initial state $s$ as parameter $s$, the set of all threads in the abstraction($T_s$) as parameter $T$, the set of abstraction locations $L_s$ as parameter $L_s$ and $T_s$ passed in as parameter $T_s$. During execution of *explore*, the set of all locations executed ($L_r$) is recorded. After *explore* finishes, *slice–n–dice* calls *refine* because the target was not reached. The parameters passed into *refine* are the input program on Table 3.1 as parameter $P$, $L_s = \{t_0 : 0, t_0 : 1, t_0 : 2\}$, $L_r = \{t_0 : 0, t_0 : 1\}$, $T_a = \{t_0, t_1, t_2\}$ and $T_s = \{t_0\}$.

The only unreached location in the abstraction is $t_0 : 2$. So *refine* calls *newLocs*($l_u$,$P$) with $t_0 : 2$ as *ul*. The result of this call returns $t_1 : 2$ in the input program using the same

**Data**: $P$, the input program, $L_s$, the set of locations identified by control flow analysis and $L_r$, all locations visited during program execution, $T_a$, all threads in the program, $T_s$ all threads in the abstraction of the original program and $P$ the input program

**Result**: $L_s$: the new set of relevant locations and $T_s$: the new set of threads in the abstraction of the original program

```
1 begin
2 |  if ∃l_u ∈ (L_s − L_r) : (newLocs(l_u)−L_s) ≠ ∅ then
3 |  |  L_s = L_s ∪ newLocs(l_u,P);
4 |  end
5 |  if ∃t ∈ T_a : t ∉ T_s ∧ canExecute(L_s,t) then
6 |  |  T_s = T_s ∪ t;
7 |  end
8 |  return (L_s, T_s);
9 end
```

**Algorithm 3:** This algorithm encapsulates the refinement process of finding new relevant locations to add to the slice as well as adding new threads to the abstraction of the original program. Its name is *refine*($L_s$,$L_r$,$T_a$,$T_s$,$P$).

analysis previously performed on the example program using $t_0 : 2$ as the unreached location. Location $t_1 : 2$ is subsequently added to the abstraction, $L_s$. Then *refine* adds an additional thread, $t_1$ to the set of abstraction threads, $T_s$. This is because $t_1$ is the only thread not in the set of abstraction threads that can execute members of $L_s$ ($t_1 : 2$). Lines 5-6 of *refine* perform this operation, with *canExecute*($t$,$L$) returning true when thread $t_1$ is passed in as $t$ and $L_s$ passed in as $L$, location $t_1 : 2$ is what causes this evaluation to return true. Because *refine* returned updated versions of $T_s$ and $L_s$, *slice–n–dice* makes another call to *explore* with the same input parameters as the initial call except that the new $T_s$ and $L_s$ from the pair returned by *refine* are passed into *explore*. The *explore* portion of the algorithm finds the error while exploring the state space of the threads in the abstraction.

A strength of slice–n–dice is illustrated using the following example. What if the target location was still found to be unreachable using the current threads in the abstraction, $t_0$ and $t_1$? The call to algorithm 3 would reveal that $t_2$ does not affect the value of $x$, the variable involved in reaching the target location. Therefore, the algorithm would detect it was done and would exit informing the user that no error was found.

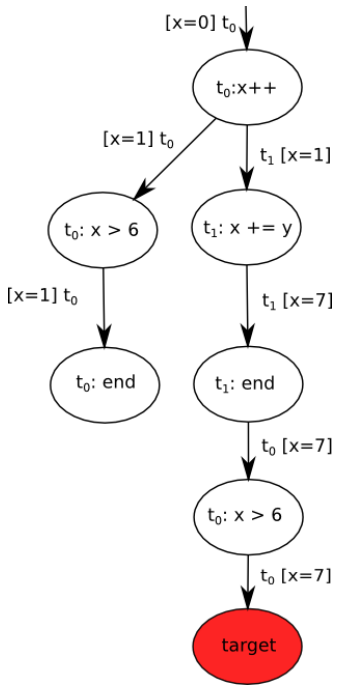


Figure 3.1: Sample executions of the example program using the slice–n–dice algorithm.

## 3.3 JPF Primer

In order to understand the implementations of the various algorithms, basic understanding of how JPF works is needed. The purpose of this section is to describe how JPF works and the mechanisms that can be used in order to alter how it executes. JPF works by simulating the Java Virtual Machine.

The basic parts of JPF that are alterable are choice generators, the scheduler, the listeners and the search strategy. Choice generators are objects that store lists of threads to execute from a program state. The scheduler is an entity that creates and saves choice generators to the virtual machine in response to shared object accesses and notifications about changes in thread states (such as threads being started). Listeners are objects that a user can write that receive notifications from JPF about the simulation of the Java Virtual Machine. Listeners also have the ability to save choice generators to the virtual machine during a notification. When a choice generator is saved to the virtual machine, its state is saved along with the choice generator. A program state contains a current heap state and

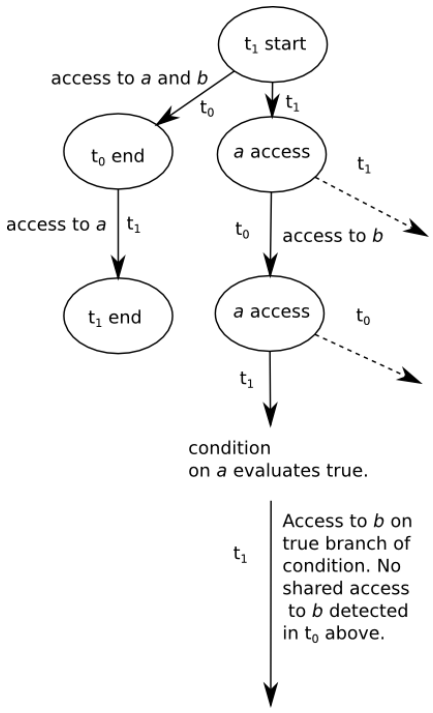


Figure 3.2: An example failing of JPF Sharedness discovery.

the states of the individual threads' stacks and program counters. JPF simulates the states generated by executing each thread in the choice generator from a saved VM state in the order specified by the search strategy. During execution of a thread from a saved VM state, another choice generator may be saved. This creates execution graphs as in Figure 3.2.

In Figure 3.2, circles represent choice generators and the state saved with them and arrows represent execution of a thread. Each arrow is labeled with the thread executing from the previously saved state. The graph shows how JPF recursively explores the state space of an input program by executing a thread from a choice generator until another choice generator is created. Choice generators and the search strategy drive execution of the program and their creation is controlled by listeners and schedulers. A more in-depth description of JPF's execution engine is in Appendix 3.8.

### 3.3.1 A Note on Sharedness Discovery

JPF detects sharedness on objects dynamically. When JPF executes an instruction accessing an object, JPF notifies the object that it is accessed by the thread executing the instruction. When the number of threads accessing an object is greater than two, JPF requests a choice generator from the scheduler corresponding to the shared object access. This is how JPF detects sharedness and schedules interleavings on shared object accesses.

A nuance of this form of sharedness detection is that an object is not recognized as shared until accessed by an additional thread. This results in no interleaving occurring during the first access to the shared object. In order to capture this first interleaving, JPF runs a path of execution from a given thread start state and uses the information gathered about the objects' sharedness to interleave properly on the next choice from a thread start. There is a hypothetical situation where an interleaving is still missed using this form of sharedness discovery.

The hypothetical situation is illustrated in Figure 3.2. Accesses to global objects that are not detected as shared are labeled on arrows. On the left-most execution path in the program, it is discovered that object $a$ is shared.

The next execution path considered is thread $t_1$ executing from the state saved with the thread start choice generator corresponding to $t_1$ starting. In this path, $t_0$ and $t_1$ are interleaved on accesses to object $a$. The result of this interleaving is a modification to $a$ that causes a condition executed by thread $t_1$ to become true. When thread $t_1$ executes the true branch of this condition, it accesses object $b$.

Object $b$ is now recognized as shared and an interleaving is performed. Even though an interleaving on $b$ happens at this state, there is no way to interleave on the first access to $b$ by $t_0$ along this path because no state was saved corresponding to the first access to $b$. This means that one interleaving between the two threads was missed! Therefore, we believe that no algorithm that relies on sharedness detection in JPF is complete. However, we have not yet encountered a situation where an error was missed due to this problem, so we believe

it is acceptable to use sharedness discovery tactics in order to decide which global objects to actually interleave on.

## 3.4 JPF Algorithm Implementations Overview

### 3.4.1 Vector Clock POR

This algorithm is described by Flanagan et al [13]. A vector clock POR works by tracking dependence between thread operations on shared objects. Two operations on a shared object are considered dependent if:

- They are co-enabled. Two operations are co-enabled if there is some program state in which either operation could be executed before the other.
- They operate on the same object.

Any two operations that do not match these criteria are considered independent. The objective of a vector clock POR is to try all orderings of dependent operations while not scheduling different orderings of independent operations. This is accomplished by looking at the search history after a shared object access. If a shared object access just took place, vector clocks are used to determine if the current shared object access is dependent with a previous shared object access. If the two accesses were dependent, the currently running thread is scheduled in the state just before the last dependent shared object access. Doing this on all shared object accesses traverses the entire unique global state space of the program.

A description of this implementation is in [22].

### 3.4.2 Iterative Context Bounding

Musuvathi, et al describes this algorithm [21]. This algorithm takes the input program, and a number K as input. K determines the number of preemptive context switches allowed in any given path of execution of the program. As the model checker simulates the input program, if a shared object access takes place, the model checker preempts the access and allows all

runnable threads to interleave at that point if the number of preemptive context switches taken in the current path of execution is less than K.

This algorithm is implemented in JPF by using a scheduler. This scheduler schedules threads in a way so that one path of execution takes place with no interleaving. The scheduler schedules this path of execution in order to discover shared object accesses between threads so interleaving on shared object accesses can occur in another path of execution. The scheduler schedules that other path of execution from the initial program state in order to interleave threads as described by Musuvathi et al. In order to interleave the threads in this way, this scheduler returns a special type of choice generator.

These choice generators store the number of preemptive context switches that have occurred so far on the path of execution. Because choice generators are stored with a program state, the scheduler only has to ask the last saved state for its choice generator in order to determine the number of preemptive context switches that have occurred so far on the current execution path. This information can be used in order to perform the algorithm by Musuvathi et al.

### 3.4.3 Heuristic guided search (HGS)

The only difference between HGS and a full exploration a program's state space is the order in which the states are explored and how many states are explored [8]. In HGS, when successor states are created in response to scheduling non-determinism, those successor states are added to a priority queue. The priority of a state in the queue is determined by its heuristic value. In many implementations of HGS, the size of the queue is bounded by a user input number and a state is only added to the queue in either of the following conditions:

- The queue is not full
- A state in the queue has a lower heuristic value than the state being added to the queue. In this case, the state with the lower heuristic value is removed and the new state is added to the queue.

In HGS, the model checker explores the state space of the program by removing the state with the highest heuristic value from the priority queue. The model checker executes the program from the removed state until new successor states are generated. The successor states are added to the priority queue if possible. The model checker then chooses another state from the priority queue to continue the search from. This process is repeated until an error state is found or there are no more states in the priority queue. This paper uses the implementation of the HGS in [26].

### 3.4.4 Slice–N–Dice

#### JPF Implementation Overview

The first major modification to JPF made to implement slice–n–dice is the implementation of a custom search strategy. This search strategy allows for the replay of a state space search after one simulation of the program is completed. The search strategy is responsible for performing the *slice–n–dice* and *refine* portions of the algorithm. The second major modification made is the implementation of a scheduler that only schedules a shared access choice generator containing abstraction threads if the line being executed and the thread executing the line is a part of the abstraction constructed by the algorithm. This encapsulates the search described in *explore*. Once all threads inside the abstraction finish execution, threads outside the abstraction are executed with no interleaving.

#### Handling Algorithm Assumptions in JPF

The assumptions in Section 3.2 do not correlate to how actual programs in Java are written. The locations isolated during the abstraction construction are associated with thread classes, not actual instances of threads. In an actual Java Program, dynamic thread creation occurs and not all threads are running from the initial program state. In order to handle the assumptions made in order to perform the slice–n–dice algorithm, slice–n–dice executes the program once with no interleaving. Executing the program once with no interleaving allows

JPF to collect dynamic information to tie actual thread instances started dynamically to classes of threads isolated during abstraction construction and this execution also allows JPF to collect information about which thread instances start other thread instances. This information is used to identify the actual thread instances that the slice–n–dice algorithm adds to the abstraction and to execute the program until a point where all threads in the abstraction are started so the algorithm can be performed.

### 3.4.5 Implementation Details of the Slice–N–Dice algorithm

#### Multiple target locations

This change allows an error to be represented by a set of multiple target locations, $L_t$ passed into *slice–n–dice* instead of a singular target location, $l_t$. In order for the error state to manifest, at least one thread must be at each target location during some execution of the target program. The *initialize*($P$,$L_t$) function constructs the initial abstraction by creating a backwards slice and identifying a thread for each target location in $L_t$ and adding them all to the initial abstraction. The algorithm remains unchanged, the only difference is what is passed into the initial call to *slice–n–dice*.

#### Refine On All Unreached Locations

During a call to *refine* instead of choosing a singular unreached location to refine on in lines 2-4, *refine* refines on all unreached locations. This guarantees that at least one thread can be added at each refinement round if possible. If this strategy is not used, the algorithm could make multiple calls to *explore* where no threads are added if refinement yields new locations in a thread already in the abstraction. We refine on all locations because adding an additional thread whenever possible makes it so the error will manifest in fewer calls to *explore*, cutting the number of times the state space of the threads in the abstraction is explored.

#### Multiple Error Manifesting Threads

This section describes how *refine* is performed when multiple threads can reach the target location. The initial abstraction $T_s$ only contains one thread that can reach the target. Let $T_t$ be the set of all threads that may execute a target location not in the set $T_s$, the threads in the abstraction. Let $T_o$ be the set of threads not in the sets $T_t$ and $T_s$ that may execute locations in the abstraction. Threads in $T_o$ may help some thread in $T_s$ reach a target location. When $T_t$ is non-empty the *refine* function is capable of adding a thread in $T_t$ to the abstraction instead of a thread in $T_o$. The problem with this is that refinements adding threads in $T_o$ are more likely to cause a call to *explore* to actually reach an error state because another thread in $T_t$ is no more likely to reach the target without interaction with other threads than the one already in $T_s$. The *refine* function is changed in this implementation of slice–n–dice so that threads in $T_o$ were added to the abstraction before threads in $T_t$ in order to alleviate this issue.

### 3.5 Results

We picked a few well-known benchmarks for our experiments on the slice–n–dice algorithm [11, 12]. These benchmarks are unique in that the difficulty for finding the error can be adjusted by parameters passed into the input program. We selected benchmarks that demonstrate how slice–n–dice performs under different patterns of thread interaction that are required for an error to manifest. We expect slice–n–dice to perform better than the other algorithms in terms of total states explored and total run time in order to find the error.

#### 3.5.1 Benchmark Experiments

*AccountSubtype*

We run experiments on the *AccountSubtype* benchmark because it tests the effectiveness of the algorithms when a specific few threads of many need to interact with the error manifesting

thread in a certain way in order to manifest the error. We expect slice–n–dice to perform well against the other algorithms because slice–n–dice only interleaves at locations indentified in the abstraction, whereas all other algorithms interleave at every shared object acccess, so even if slice–n–dice ends up adding all threads to the abstraction, it should perform well against the other algorithms. The *AccountSubtype* benchmark is a medium-sized benchmark with ninety-one significant locations. In the *AccountSubtype* benchmark two different account thread types exist, personal accounts and business accounts. The error only manifests if at least one personal account and another account interleaves in a particular way. The personal account causes the error. The three experiments for this benchmark all only run one personal account. The first experiment uses one business account, the second uses four business accounts and the third uses nine business accounts. Increasing the number of business accounts increases the state space without increasing the number of execution paths that manifest the error.

*Airline*

We run experiments on the *Airline* benchmark because it tests the effectiveness of the algorithms when many threads must interact with the error manifesting thread in order to find the error. We expect slice–n–dice to perform well against the other algorithms because slice–n–dice only interleaves on shared object accesses identified in the abstraction, whereas the other algorithms interleave on every shared object access, so even though slice–n–dice will eventually add many threads to the abstraction, we expect it to beat the other algorithms because it is interleaving on fewer shared object accesses. Three experiments are performed on *Airline*. Each one corresponds to a different number of ticket issuing threads. Experiment one uses one ticket issuing thread, experiment two uses five ticket issuing threads and experiment three uses ten ticket issuing threads. Increasing the number of ticket-issuing threads increases the state space necessary to find the error. In the first experiment featuring only one ticket-issuing thread, a cushion of one is used because that is the only cushion value

that will manifest the error based on interaction between the two threads. In all others a cushion of three is used. The cushion variable controls how deep the error is in the state space of the program. Higher cushion values make the error easier to find.

*Reorder*

We run experiments on the *Reorder* benchmark because it tests the effectiveness of the algorithms when the error manifesting thread only needs to interact with one of many threads in order to manifest the error. We expect slice–n–dice to perform the best on this benchmark because *Reorder* is the only benchmark where slice–n–dice is guaranteed to terminate after one phase of refinement. The *Reorder* benchmark is a small benchmark with forty-seven significant locations. The *Reorder* benchmark has two types of threads: Checkers and Setters. Setters cause the errors, the Checkers manifest it. The number of Checkers will be held constant at one and the number of Setters is increased in order to make the error more difficult to find. The three experiments for this benchmark that manifest the error have Setter thread counts of one, five and ten.

**Experiments With No Error**

The slice–n–dice algorithm has not been determined to be complete or incomplete. We perform experiments on slice–n–dice when there is no error in the previous benchmarks in order to show how slice–n–dice would perform against other algorithms if it is determined in future work that slice–n–dice is complete. The first experiment is on an error-free *Airline* benchmark, with four ticket issuing threads. This setup demonstrates how the various algorithms perform at a moderate thread count when they have to explore the whole state space. An interesting attribute of this set up is that slice–n–dice will detect that the other threads share with the error-manifesting thread, even though no error is possible. The second experiment is with *Reorder*. There will be no checker thread but there will be four setter threads to see how the algorithms perform when there is no error at a moderate thread count.

An interesting attribute of this experiment on slice–n–dice is no refinement will be possible that adds new threads to the abstraction. The third experiment is on an error-free version of *AccountSubtype*. Just like in *Airline* with no error, all threads in the system will eventually be added to the abstraction.

### 3.5.2 Experimental Setup

This sections details the results of running benchmarks for JPF's native POR, a vector clock based POR algorithm, K-bounded iterative context bounding, slice–n–dice and HGS. All sets of experiments are run on a machine with an AMD phenom quad core processor with an approximate 1.8 GHZ speed and 8 GB of RAM. The Java virtual machine is allocated with 2 GB of memory for each experiment. Three benchmarks are run using enabled randomization on all scheduling relevant choice generators. For each experiment on each benchmark, average run time and average total number of states is recorded for five samplings from the distribution from the possible runs for all algorithms. Execution is stopped on the first manifestation of the error. If a run takes longer than 300 seconds to complete or runs out of JVM heap memory, the run is considered failed. Otherwise, a run is considered complete. The time cap of 300 seconds was chosen because JPF would normally run into its max heap memory limit if the algorithm did not complete in that time frame or shortly thereafter.

### 3.5.3 Discussion

#### Description of the Results Tables

The following tables contain the results of all experiments: Table 3.2, Table 3.3, Table 3.4 and Table 3.5. The first column in all results tables detail the benchmark and the benchmark-specific parameters used in the particular experiment. The next column is the metrics used. The first metric is the average run time of the experiment in seconds. The next metric is the average number of total states explored. In the case of slice–n–dice this includes all rounds of execution of the program. The "% complete" metric gives the percentage of runs

| Benchmark | Metrics | Vector Clock POR | JPF POR | KBOUND (k=5) | HGS | slice–n–dice |
|---|---|---|---|---|---|---|
| *AccountSubtype* | run time (seconds) | 1.29 | 1.66 | 1.22 | 1.31 | 1.69 |
| Business=1 | States | 246 | 96 | 272 | 173 | 307 |
| Personal=1 | % complete | 100 | 100 | 100 | 100 | 100 |
| *AccountSubtype* | run time (seconds) | 8.94 | 19.6 | 5.61 | 2.50 | 4.57 |
| Business=4 | States | 22200 | 82600 | 11700 | 1318 | 5150 |
| Personal=1 | % complete | 100 | 100 | 100 | 100 | 100 |
| *AccountSubtype* | run time (seconds) | 58.9 | - | 26.7 | 26.7 | 10.4 |
| Business=9 | States | 146000 | - | 92700 | 21100 | 15700 |
| Personal=1 | % complete | 40 | 0 | 100 | 100 | 60 |

Table 3.2: The results of the algorithms on the *AccountSubtype* benchmark.

of the five that completed in the allotted time frame and memory limits. This is a necessary metric because some experiments have good results, but only on the experiments where the randomness in scheduling or the algorithm caused the algorithm to find the error within the allotted time frame. When no experiments managed to finish within the allotted time frame or memory limits, a '-' is put in the place of metrics for run time and number of states explored. The third column from the left corresponds to the results of the vector clock POR on this benchmark. The fourth column from the left correspond to the results from iterative context deepening with a K of 5 (labeled KBOUND (k=5)). The Fifth column from the left shows the results from JPF's POR, the sixth column shows the result of slice–n–dice and the seventh column shows the results of HGS.

*AccountSubtype*

The results on the *AccountSubtype* benchmark are given in Table 3.2. Slice–n–dice does not perform well on the experimental set-up with the smallest thread count possible to find an error. This is not surprising because all threads in the system are necessary in order for the error to manifest, but slice–n–dice does not have all threads in the system in its abstraction until it adds both the personal account and business account to the program, because the main thread detects the error. As the number of threads is increased, slice–n–dice starts to perform better against all other methods except heuristic guided search and iterative context bounding.

A core problem with the slice–n–dice algorithm is demonstrated in Table 3.2 by the results on this benchmark. This is shown by the experiments with nine business accounts. When slice–n–dice succeeded on experiments with nine business accounts, it performed better than all other algorithms. The table also shows that slice–n–dice fails in two of the five experiments performed with nine business accounts. The reason why the slice–n–dice algorithm fails to finish at high thread counts on some experiments is because a specific two threads need to be scheduled with the thread manifesting the error and every other thread is eligible to be added during a refinement phase. This means that at each refinement phase, if there are $n$ non-slice threads, that there is only a $1/n$ chance of adding a thread relevant to finding the error. If the wrong threads are chosen during refinement, the algorithm is guaranteed to not find the error during the following execution phase. An additional consequence of adding the wrong threads is when the appropriate threads are added in a successive refinement phase, the state space to find the error in is significantly larger than need be during the following phase of execution. HGS performed the best on this benchmark because it completed all of its runs in the lowest amount of time and explored the fewest number of states. Slice–n–dice had an average lower run time on high thread counts, but failed on two of its experiments at high thread counts. Slice–n–dice had that lower average run time than HGS on successful experiments for the following reason: When the slice–n–dice algorithm succeeded, it had chosen the right threads within the first five phases of refinement. This fact effectively means that slice–n–dice was exploring a state space with fewer threads than HGS, making the error easier for slice–n–dice to find.

*Airline*

The results on the *Airline* benchmark are given in Table 3.3. Slice–n–dice does not perform well on the experimental set-up with the smallest thread count possible to find an error. This is not surprising because just like in *AccountSubtype* all threads in the system are necessary

| Benchmark | Metrics | Vector Clock POR | JPF POR | KBOUND (k=5) | HGS | slice–n–dice |
|---|---|---|---|---|---|---|
| *Airline* | run time (seconds) | .831 | .786 | .833 | 1.20 | 1.61 |
| Issuers=1 | States | 8.2 | 3.8 | 8.6 | 12 | 15 |
| Cushion=1 | % complete | 100 | 100 | 100 | 100 | 100 |
| *Airline* | run time (seconds) | 11.8 | 4.27 | 1.00 | 3.03 | 1.96 |
| Issuers=5 | States | 33500 | 17100 | 69.2 | 249 | 186 |
| Cushion=3 | % complete | 100 | 100 | 100 | 100 | 100 |
| *Airline* | run time (seconds) | - | 54.4 | 1.29 | 13.8 | 8.09 |
| Issuers=10 | States | - | 326000 | 460 | 16700 | 7820 |
| Cushion=3 | % complete | 0 | 20 | 100 | 60 | 100 |

Table 3.3: The results of the algorithms on the *Airline* benchmark.

in order for the error to manifest, but slice–n–dice does not have all threads in the system until it adds the all threads during phases of execution and refinement.

According to Table 3.3, K-bounded iterative context bounding performs the best on all experiments. Slice–n–dice does not do well compared iterative context bounding on *Airline*. *Airline* requires that a certain number of threads in the system be present in order to discover the error. This means every phase of adding a thread during refinement and resulting execution phase of slice–n–dice will fail before the correct number of threads exists in the slice. Iterative context bounding does not suffer from this problem because it already considers all threads while performing an under approximation. When the total number of required interacting threads for finding the error is increased, the slice–n–dice algorithm still outperforms JPF's POR and the vector clock POR, because slice–n–dice interleaves on far fewer shared object accesses.

Table 3.3 shows that the vector clock POR does not perform significantly better than JPF's POR when both algorithms terminate after the error is found. In fact, on the *Airline* benchmark experiments it performs worse. But, in Table 3.5 it is obvious that in experiments where no error is possible that the vector clock POR performs better. Previous research also shows that on a full state explore the vector clock POR performs better [22]. The vector clock POR performing better than JPF's POR during full program state exploration means that there is a difference in error density of the state spaces explored by the two different approaches when the algorithms terminate after finding an error.

| Benchmark | Metrics | Vector Clock POR | JPF POR | KBOUND (k=5) | HGS | slice–n–dice |
|---|---|---|---|---|---|---|
| *Reorder* | run time (seconds) | 1.01 | 1.22 | .997 | 2.11 | 1.61 |
| Setters=1 | States | 62 | 285 | 73.4 | 137 | 233 |
| Checkers=1 | % complete | 100 | 100 | 100 | 100 | 100 |
| *Reorder* | run time (seconds) | 229 | 297 | 4.40 | 8.13 | 1.70 |
| Setters=5 | States | 1140000 | 1520000 | 8870 | 11900 | 283 |
| Checkers=1 | % complete | 100 | 20 | 100 | 100 | 100 |
| *Reorder* | run time (seconds) | - | - | 55.0 | 90.7 | 1.70 |
| Setters=10 | States | - | - | 179000 | 6991 | 251 |
| Checkers=1 | % complete | 0 | 0 | 100 | 60 | 100 |

Table 3.4: The results of the algorithms on the *Reorder* benchmark.

*Reorder*

According to Table 3.4 this is the benchmark where the slice–n–dice algorithm really shined. The reason for this is the error manifesting thread only shares object accesses with threads that can help it manifest the error, and only one of those threads is needed to help it reach that error. This means that only one stage of refinement is needed for slice–n–dice to succeed regardless of the number of threads in the experimental setup. It is apparent from the results of the slice–n–dice algorithm on this benchmark that on this benchmark, there is not a significant change in the number of states or run time required to manifest the error as threads are increased. In contrast, all other verification techniques experience significant increases in the number of states and run time required to find the error as the number of threads is increased. Slice–n–dice methods even beat iterative context bounding at high thread counts because iterative context bounding is still exponential in the number of context switches allowed and the number of possible context switches at any depth less than K scales against the number of total threads in the system [21].

Another interesting result on this benchmark is from the HGS algorithm. The experiments that succeeded with ten setters explored fewer states than the experiments that succeeded with five setters, but the run time was significantly higher with ten setters anyway. This is because the HGS algorithm has more overhead associated with it as the number of threads is increased and JPF started to experience thrashing due to memory constraints.

| Experiments With No Error | | | | | | |
|---|---|---|---|---|---|---|
| Benchmark | Metrics | Vector Clock POR | JPF POR | KBOUND (k=5) | HGS | slice–n–dice |
| *Airline* | run time (seconds) | 10.2 | 29.9 | 5.99 | - | 2.48 |
| Issuers=4 | States | 29700 | 183000 | 16700 | - | 736 |
| Cushion=0 | % complete | 100 | 100 | 100 | 0 | 100 |
| *Reorder* | run time (seconds) | 8.24 | 17.2 | 5.77 | - | 1.10 |
| Setters=4 | States | 20200 | 72500 | 14000 | - | 65 |
| Checkers=0 | % complete | 100 | 100 | 100 | 0 | 100 |
| *AccountSubtype* | run time (seconds) | - | - | 54.1 | - | 85.1 |
| Business=3 | States | - | - | 243000 | - | 295000 |
| Personal=1 | % complete | 0 | 0 | 100 | 0 | 100 |

Table 3.5: The results of the algorithms on each benchmark with no error present.

### Error Free Experiments

Table 2.1 shows that slice–n–dice performed well on these experiments. Slice–n–dice explored fewer states than the other verification methods by a factor of ten on *Airline* and *Reorder*. The main reason being that the backwards slices in the abstraction under approximate the object interactions needed to reach the error. In the case of the *Airline* experiment, even though all threads end up being interleaved, it still performs well against the other methods because it does not interleave on as many shared object accesses. Slice–n–dice never added more threads to the abstraction in the *Reorder* example and exited after one run of the program as is expected when no error manifesting thread is present. HGS does not do well on this experiment because it is constantly doing analysis on the states generated in the search in order to force the search to an unreachable state. If given more time, HGS would have eventually finished and reported that it could not find the error. On *AccountSubtype*, slice–n–dice is one of the few algorithms that succeeded. This is due to the fact that slice–n–dice only interleaves on locations it identifies in its abstraction. Iterative context bounding is the only other algorithm that completed, but it is known that this method is incomplete, so the results on that algorithm when no error is present do not tell us anything about the program.

### 3.6 Related Work

In previous work, Rungta implemented a system where the state space of a program was searched using heuristics based on static analysis in order to hit a target location [26]. This system worked by using the heuristic to choose a thread to run at scheduling relevant points during execution. Slice–n–dice came about because through experiments, it became obvious that in certain cases, only a subset of the threads were needed to hit a certain target location. The original description of slice–n–dice was first described by Rungta [25].

The dynamic partial order reduction algorithm used in this paper as part of dynamic slice–n–dice and as a comparison algorithm is based on a strategy of determining partial orders through collecting information about the state space as the search continues. The main other kind of partial order reduction is static partial order reductions. In static partial order reductions (SPOR), the program is analyzed at compile time to determine dependencies between operations [20]. Whereas in dynamic partial order reductions (DPOR) such as the one in this paper, the program determines dependence based on changes in program state as it is run [13]. There are various methods to do this. One of these methods uses persistent and sleep sets [13]. A persistent set is a set of transitions such that all transitions outside the set are independent of the transitions inside of it [14]. The vector clock POR in this paper is derived based on persistent sets [13]. The idea behind sleep sets is that they are sets of transitions that are independent with each other (exploring a schedule with an interleaving of two transitions in a sleep set yields the same global state). Once one interleaving of the independent transitions has been explored, it is not explored again in the opposite order [14]. The idea behind a DPOR using the sleep set and persistent set technique is to schedule every transition in a persistent set for each state and execute those schedules without repeating any interleavings in a sleep set more than once. If this sort of exploration is done, then it is guaranteed that all possible global results for a given program have been explored [13]. The primary advantage of a DPOR is that it has more information about the program available

to it during execution even though it takes more memory than SPOR. POR methods have drawbacks.

POR strategies try to force multiple threads to make a concurrent access to shared objects. Sometimes a concurrent access to a shared object is not possible but the POR's methods of determining dependence do not capture this, resulting in extra interleavings. There are strategies for determining which heap accesses can actually occur concurrently in order to further trim the state space of POR methods [7]. This method of trimming state space has also been used on JPF's POR [23].

Another type of algorithm that slice–n–dice is compared to is a K-bounded method. K-bounded methods work by putting a bound on the state space explored. Iterative context bounding was chosen for this paper as a comparison algorithm. In this algorithm, K preemptive context switches are added to the threads and then executed in hopes of detecting an error [21]. All combinations of K preemptive context switches within the threads are tried. All possible combinations of the next thread after the context switch are also tried. This ensures that all errors found within K context switches are guaranteed to be found. Another K-bounded method of under-approximating a system's state space to yield faster results for detecting errors is K-bounded delay scheduling. K-bounded delay scheduling takes a deterministic scheduler operating on threads [9]. In this case, deviations from the normal scheduler are taken and K is the bound on the number of times any given execution of the program can deviate from the deterministic scheduler [9]. The main problem with K-bounded methods is that no guarantee is given that all errors will be found within the given K. The heuristic guided search is a better search prioritization method when catching errors that occur with a high number of context switches than K-bounded methods [27].

## 3.7 Conclusions and Future Work

### 3.7.1 Conclusion

The slice–n–dice algorithm performs best against other verification techniques in the case where there are many threads, but a given error manifesting thread only interacts with threads that help it manifest the error and only a non-specific few of those threads are needed in the slice in order to manifest the error. From the experiments on slice–n–dice, it becomes obvious that interleaving on fewer object accesses can be clearly advantageous in finding the error because slice–n–dice yields far fewer states than other algorithms when the whole state space of the program needs to be explored on the *Airline* and *Reorder* benchmarks. The advantages of this under-approximation are also demonstrated by the *Airline* experiment, because slice–n–dice still performed well against POR methods at high thread counts even though many threads were required to find the error. When slice–n–dice performs poorly against other verification techniques, it is due to mistakes made during the refinement step. The mistake that occurs is either not adding enough threads or adding the wrong ones.

### 3.7.2 Future Work

An improvement for the slice–n–dice algorithm would be finding a way to pick threads more intelligently. The first improvement would be a deeper analysis of how many threads are needed to reach the error. If the refinement step had a way knowing how many new threads isolated during static analysis to add, then fewer refinement and successive execution phases would be needed in reaching an error. This sort of improvement would make slice–n–dice stronger on examples like *Airline*.

## 3.8 Appendix A

Algorithm 4 outlines in more detail how JPF explores program states of an input program. The following definitions are necessary for understanding Algorithm 4:

- Let *getPC*($t$) return the instruction that thread $t$'s program counter currently references, it returns 0 if thread $t$ is terminated.
- Let *setPreExecute*($t$) notify thread $t$ that it is in the pre-execution phase of the instruction at its current program counter.
- Let *clearPreExecute*($t$) notify thread $t$ that it is no longer in the pre-execution phase of the instruction at its current program counter.
- Let *isPreExecute*($t$) return true if thread $t$ is in the pre-execution phase of the instruction at its current program counter, false otherwise.
- Let *advancePC*($t$) move thread $t$'s program counter to the next instruction to be executed.
- Let $L$ be the set of listeners JPF was configured to run with.
- Let *notifyPreExecute*($L$,$t$,$i$) notify all listeners in the set $L$ that thread $t$ is about to execute instruction $i$.
- Let *notifyPostExecute*($L$,$t$,$i$) notify all listeners in the set $L$ that thread $t$ is about to execute instruction $i$.
- Let *object*($i$) return the object that instruction $i$ operates on.
- Let *accessing*($o$) return the set $A$, the set of all threads that executed an instruction that accessed object $o$.
- Let *updateAccessing*($o$,$t$) add thread $t$ to $A$ the set of all threads that accessed object $o$.
- Let *threadsIn*($cg$) return the set of threads, $T$ in choice generator $cg$.
- Let *isStart*($i$) return true if instruction $i$ is a thread start instruction.
- Let *enabled*($s$) return the set of instructions enabled in state $s$.

- Let *execute*($t$,$s$) return a program state corresponding to executing the instruction referenced by thread $t$'s current program counter from state $s$.
- Let *getAccessCG*($o$,$S$) return a choice generator corresponding to a shared access to object $o$ from scheduler $S$.
- Let *getStartCG*($t$,$S$) return a choice generator corresponding to thread $t$ starting another thread from scheduler $S$.

Algorithm 4 outlines how JPF executes instructions in different threads in a choice generator in order to traverse the state space of an input program. The initial call to Algorithm 4 passes in the following parameters: a set of user-specified listeners($L$), a user-specified scheduler($S$), the initial system state($s$) and a choice generator only containing the main thread(*cg*). On line 11, the algorithm updates the sharedness of the object $o$ accessed by the instruction at the thread's current program counter. On lines 13-16, if the set of threads that access $o$ is greater than one and there is more than one thread enabled in the current state $s$, then the algorithm calls the scheduler asking it for a choice generator associated with the access to $o$. This algorithm shows how JPF saves the state just before the execution of the instruction at the thread's current program counter if it corresponds to a shared object access (lines 13-17). In contrast, in lines 24-28 it is obvious that the state just after execution of the instruction is saved in the case of choice generators being created for threads being started. Lines 9-17 encapsulate the pre-execution phase of instruction execution. In this phase of execution, sharedness of the object $o$ accessed by the instruction is updated (line 11) and a call to the user-specified listeners is made, notifying them that the instruction is about to be executed (line 12). JPF notifies the listeners again just after execution of the instruction (line 20).

```
Data: L: the set of listeners configured at start-up. S: The scheduler configured at
      start-up. s: The current system state. cg: The choice generator containing
      threads to execute from state s.
Result: Whether or not an error state was reached
1  begin
2      foreach t ∈ threadsIn(cg) do
3          repeat
4              if s is an error state then
5                  report error and exit;
6              end
7              i = getPC(t);
8              o = object(i);
9              if isPreExecute(t) then
10                 clearPreExecute(t);
11                 updateAccessing(o,t);
12                 notifyPreExecute(L,t,i);
13                 if |accessing(o)| > 1 ∧ |enabled(s)| then
14                     cg′ = getAccessCG(o,S);
15                     explore(L,S,s, cg′);
16                     break;
17                 end
18             end
19             s′ = execute(t,s);
20             notifyPostExecute(L,t,i);
21             advancePC(t);
22             setPreExecute(t);
23             s = s′;
24             if isStart(i) then
25                 cg′ = getStartCG(t,S);
26                 explore(L,S,s, cg′);
27                 break;
28             end
29         until getPC(t) ≠ 0;
30     end
31 end
```

**Algorithm 4:** This algorithm encapsulates the JPF execution engine.

# Chapter 4

## Conclusions

The results of the first paper in Chapter 2 show that PORs based on formal methods (such as the vector clock POR) are stronger than JPF's POR for exploring the entire state space of a program. The results of the second paper in chapter 3 confirm this. From the results of the second paper, it is clear that slice–n–dice performs well against both POR methods on all benchmarks presented in that paper. Slice–n–dice performs well against iterative context deepening and heuristic guided search when the error-manifesting thread only interacts with threads that help it manifest the error and a few non-specific threads are needed to help the error-manifesting thread reach the error. From the results of the paper in Chapter 3, slice–n–dice appears to be a promising algorithm. Future work on slice–n–dice would determine the completeness of the algorithm in order to determine when it is safe to use as a sole verification technique. One major weakness of the slice–n–dice algorithm is that the user has to identify the error in some way themselves. Therefore, other future work would also include automated ways of determining locations that represent error states.

## References


[1] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. In Eerke A. Boiten, John Derrick, and Graeme Smith, editors, *IFM*, volume 2999 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2004. ISBN 3-540-21377-5. URL `http://dblp.uni-trier.de/db/conf/ifm/ifm2004.html#BallCLR04`.

[2] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The Software Model Checker blast. *STTT*, 9(5-6):505–525, 2007.

[3] Edmund M. Clarke, Daniel Kroening, and Karen Yorav. Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking. In *DAC*, pages 368–371. ACM, 2003. ISBN 1-58113-688-9. URL `http://dblp.uni-trier.de/db/conf/dac/dac2003.html#ClarkeKY03`.

[4] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement for Symbolic Model Checking. *Journal of the ACM*, 50(5): 752–794, September 2003.

[5] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Proving Program Termination. *Commun. ACM*, 54(5):88–98, 2011. URL `http://dblp.uni-trier.de/db/journals/cacm/cacm54.html#CookPR11`.

[6] Lucas Cordeiro, Bernd Fischer 0002, and Joo P. Marques Silva. SMT-Based Bounded Model Checking for Embedded ANSI-C Software. *CoRR*, abs/0907.2072, 2009. URL `http://dblp.uni-trier.de/db/journals/corr/corr0907.html#abs-0907-2072`.

[7] Matthew B. Dwyer, John Hatcliff, Robby, and Venkatesh Prasad Ranganath. Exploiting object escape and locking information in partial-order reductions for concurrent object-oriented programs. *Form. Methods Syst. Des.*, 25(2-3):199–240, September 2004. ISSN 0925-9856. doi: 10.1023/B:FORM.0000040028.49845.67. URL `http://dx.doi.org/10.1023/B:FORM.0000040028.49845.67`.

[8] Stefan Edelkamp, Alberto Lluch-Lafuente, and Stefan Leue. Directed Explicit Model Checking with HSF-SPIN. In Matthew B. Dwyer, editor, *SPIN*, volume 2057 of *Lecture*

*Notes in Computer Science*, pages 57–79. Springer, 2001. ISBN 3-540-42124-6. URL http://dblp.uni-trier.de/db/conf/spin/spin2001.html#EdelkampLL01.

[9] Michael Emmi, Shaz Qadeer, and Zvonimir Rakamaric. Delay-Bounded Scheduling. In Thomas Ball and Mooly Sagiv, editors, *POPL*, pages 411–422. ACM, 2011. ISBN 978-1-4503-0490-0. URL http://dblp.uni-trier.de/db/conf/popl/popl2011.html#EmmiQR11.

[10] Javier Esparza and Keijo Heljanko. Implementing LTL Model Checking with Net Unfoldings. In Matthew B. Dwyer, editor, *SPIN*, volume 2057 of *Lecture Notes in Computer Science*, pages 37–56. Springer, 2001. ISBN 3-540-42124-6. URL http://dblp.uni-trier.de/db/conf/spin/spin2001.html#EsparzaH01.

[11] Yaniv Eytani, Klaus Havelund, Scott D. Stoller, and Shmuel Ur. Toward a Framework and Benchmark for Testing Tools for Multi-Threaded Programs. *Conc. & Comp.: Practice & Experience*, 19, 2007.

[12] Eitan Farchi, Yarden Nir, and Shmuel Ur. Concurrent Bug Patterns and How to Test Them. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, IPDPS '03, pages 286.2–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1926-1. URL http://dl.acm.org/citation.cfm?id=838237.838485.

[13] Cormac Flanagan and Patrice Godefroid. Dynamic Partial-order Reduction for Model Checking Software. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 110–121, New York, NY, USA, 2005. ACM. ISBN 1-58113-830-X. doi: http://doi.acm.org/10.1145/1040305.1040315.

[14] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996. ISBN 3-540-60761-7.

[15] Alex Groce and Willem Visser. Model Checking Java Programs Using Structural Heuristics. In *ISSTA*, pages 12–21, 2002. URL http://dblp.uni-trier.de/db/conf/issta/issta2002.html#GroceV02.

[16] Alex Groce and Willem Visser. Heuristic Model Checking for Java Programs. In Dragan Bosnacki and Stefan Leue, editors, *SPIN*, volume 2318 of *Lecture Notes in Computer Science*, pages 242–245. Springer, 2002. ISBN 3-540-43477-1. URL http://dblp.uni-trier.de/db/conf/spin/spin2002.html#GroceV02.

[17] Orna Grumberg, Flavio Lerda, Ofer Strichman, and Michael Theobald. Proof-Guided Underapproximation-Widening for Multi-Process Systems. In Jens Palsberg and Martn Abadi, editors, *POPL*, pages 122–131. ACM, 2005. ISBN 1-58113-830-X. URL `http://dblp.uni-trier.de/db/conf/popl/popl2005.html#GrumbergLST05`.

[18] Radu Iosif. Exploiting Heap Symmetries in Explicit-State Model Checking of Software. In *ASE*, pages 254–261. IEEE Computer Society, 2001. ISBN 0-7695-1426-X. URL `http://dblp.uni-trier.de/db/conf/kbse/ase2001.html#Iosif01`.

[19] Sarfraz Khurshid, Corina S. Pasareanu, and Willem Visser. Generalized Symbolic Execution for Model Checking and Testing. In Hubert Garavel and John Hatcliff, editors, *TACAS*, volume 2619 of *Lecture Notes in Computer Science*, pages 553–568. Springer, 2003. ISBN 3-540-00898-5. URL `http://dblp.uni-trier.de/db/conf/tacas/tacas2003.html#KhurshidPV03`.

[20] Robert P. Kurshan, Vladimir Levin, Marius Minea, Doron Peled, and Hsn Yenign. Static Partial Order Reduction. In Bernhard Steffen, editor, *TACAS*, volume 1384 of *Lecture Notes in Computer Science*, pages 345–357. Springer, 1998. ISBN 3-540-64356-7. URL `http://dblp.uni-trier.de/db/conf/tacas/tacas98.html#KurshanLMPY98`.

[21] Madanlal Musuvathi and Shaz Qadeer. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. In Jeanne Ferrante and Kathryn S. McKinley, editors, *PLDI*, pages 446–455. ACM, 2007. ISBN 978-1-59593-633-2. URL `http://dblp.uni-trier.de/db/conf/pldi/pldi2007.html#MusuvathiQ07`.

[22] Eric Noonan, Eric Mercer, and Neha Rungta. Vector-clock Based Partial Order Reduction for JPF. *SIGSOFT Softw. Eng. Notes*, 39(1):1–5, February 2014. ISSN 0163-5948. doi: 10.1145/2557833.2560581. URL `http://doi.acm.org/10.1145/2557833.2560581`.

[23] P. Parizek and O. Lhotak. Identifying future field accesses in exhaustive state space traversal. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 93–102, Nov 2011. doi: 10.1109/ASE.2011.6100154.

[24] Corina S. Pasareanu, Radek Pelnek, and Willem Visser. Predicate Abstraction with Under-approximation Refinement. *CoRR*, abs/cs/0701140, 2007. URL `http://dblp.uni-trier.de/db/journals/corr/corr0701.html#abs-cs-0701140`.

[25] Neha Rungta and Eric Mercer. Slicing and Dicing Bugs in Concurrent Programs. In Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastin Uchitel, editors,

*ICSE (2)*, pages 195–198. ACM, 2010. ISBN 978-1-60558-719-6. URL http://dblp.uni-trier.de/db/conf/icse/icse2010-2.html#RungtaM10.

[26] Neha Rungta and Eric G. Mercer. Guided Model Checking for Programs with Polymorphism. In *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '09, pages 21–30, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-327-3. doi: 10.1145/1480945.1480950. URL http://doi.acm.org/10.1145/1480945.1480950.

[27] Neha Rungta and Eric G. Mercer. Hardness for Explicit State Software Model Checking Benchmarks. In *SEFM*, pages 247–256. IEEE Computer Society, 2007. ISBN 978-0-7695-2884-7. URL http://dblp.uni-trier.de/db/conf/sefm/sefm2007.html#RungtaM07.

[28] Koushik Sen and Gul Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *Proceedings of the 18th International Conference on Computer Aided Verification*, CAV'06, pages 419–423, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-37406-X, 978-3-540-37406-0. doi: 10.1007/11817963_38. URL http://dx.doi.org/10.1007/11817963_38.

[29] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A Concolic Unit Testing Engine for C. In *ESEC/FSE'05*. ACM Press, 2005.