

Vector-Clock Based Partial Order Reduction for JPF*

Eric Noonan
Brigham Young University
Provo, Utah
noonanes@gmail.com

Eric Mercer
Brigham Young University
Provo, Utah
egm@cs.byu.edu

Neha Rungta
NASA, Ames Research
Center
Mountain View, CA
neha.s.rungta@nasa.gov

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.

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 JPF's 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 [2]; and (ii) a comparison between JPF's native POR and the JPF clock vector partial order reduction implementation.

2. JPF'S POR OVERVIEW

Table 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 $\text{accessed}(o)$ function returns a set of SGOIDS corresponding to the threads that accessed the object referenced by o .
- The $\text{enabled}(s)$ function returns all the enabled threads in state s .
- The $\text{nextins}(p)$ function returns the next instruction to be executed by 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).

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

*Research funded by NSF grant CCF-1302524

```

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: }

```

Table 1: Pseudo-code describing the POR in JPF.

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.

3. CLOCK VECTORS

A full description of a partial order reduction using clock vectors is in [2]. 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 $C(p)$ 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

```

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:=|S'|]
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: }

```

Table 2: Pseudo-code for the clock vector POR.

vector $C(p_i) = \{c_1, \dots, c_m\}$. 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 Table 2. Definitions for understanding the table are as follows:

- S is a transition sequence represented by $\{t_1, \dots, t_m\}$ 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 Table 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.

4. EXAMPLE

The two approaches are illustrated using the following

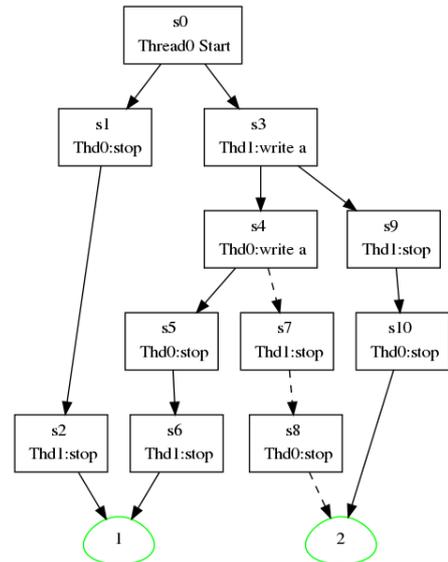


Figure 1: The JPF search space for the two PORs.

simple program that updates the global variable a :

```

    t0      t1
0 : Thread.start(t1);  a = 2;
1 : a = 1;

```

The search space explored by JPF for this program (omitting initialization code) is shown in Figure 1. 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 1. 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 interleaves are 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 $\langle 0,0 \rangle$ 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 1 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.

5. RESULTS

This section 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 3(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 [5]. 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 3(b) shows the results on another benchmark written by Mercer and Rungta called reorder [5]. The strategy when doing these experiments on the reorder benchmark 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.

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 [4]. 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 [2]. There are various methods to do this. One of these methods uses persistent and sleep sets [2]. A persistent set is a set of transitions such that all transitions outside the set are inde-

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

(a)

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

(b)

Table 3: Results from examples. (a) The Airline model. (b) The Reorder model.

pendent of the transitions inside of it [3]. The algorithm in this paper is derived based on persistent sets [2]. 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 [3]. 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 [2]. 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.

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 [2]. Af-

ter 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.

8. REFERENCES

- [1] J. Esparza and K. Heljanko. Implementing LTL Model Checking with Net Unfoldings. In M. B. Dwyer, editor, *SPIN*, volume 2057 of *Lecture Notes in Computer Science*, pages 37–56. Springer, 2001.
- [2] C. Flanagan and P. 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.
- [3] P. 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.
- [4] R. P. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenigün. Static Partial Order Reduction. In B. Steffen, editor, *TACAS*, volume 1384 of *Lecture Notes in Computer Science*, pages 345–357. Springer, 1998.
- [5] N. Rungta and E. G. Mercer. Hardness for Explicit State Software Model Checking Benchmarks. In *SEFM*, pages 247–256. IEEE Computer Society, 2007.