JPF Verification of Habanero Java Programs using Gradual Type Permission Regions*

Peter Anderson
Brigham Young University
Provo, Utah
anderson.peter@byu.edu

Nick Vrvilo
Rice University
Houston, Texas
nv4@rice.edu

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

Vivek Sarkar
Rice University
Houston, Texas
vsarkar@rice.edu

ABSTRACT

The Habanero Java Library (HJ-lib) is a Java 8 library implementation of the Habanero Java (HJ) programming model. Calls into this pure Java library provide support for all HJ primitives, including async, finish, and phasers. In previous work, we presented VR, a custom verification runtime designed to be used within Java Pathfinder (JPF) to verify a subset of HJ programs. In this work, we present VR-lib, a library implementation of HJ, which supports verification of a larger subset of programs than VR. Additionally, we present the implementation of gradually typed permission regions (GPRs). PRs provide a building block for dynamically detecting violations of conditions sufficient to guarantee race-freedom. Lastly, we present results for benchmarks using PRs in combination with VR-lib to verify HJ programs.

1. INTRODUCTION

Currently, writing correct and efficient concurrent programs is a challenge that requires a programmer with expertise in concurrency. The Habanero project at Rice University aims to remedy this situation by developing programming and execution models that enable a programmer to utilize the performance benefits of multi-core processors without directly using low-level primitives. Habanero Java (HJ), a mid-level concurrency language, is an implementation of one such programming model that provides concurrency constructs such as asynchronous tasking, message passing, and point-to-point synchronization in addition to safety guarantees, such as deadlock freedom, determinism, and serializability for well-defined subsets of the HJ language [**?**].

However, the determinism guarantee is contingent on the program being free of data races. Thus providing a guarantee that a HJ program is data race free also provides many other important guarantees. In previous work[**?**] we introduced VR, a custom verification runtime to run in Java Pathfinder (JPF). VR supported a subset of the primitives that HJ provides. Since then Habanero group at Rice has released HJ-lib, a pure library implementation of HJ using Java 8. Any programs written using HJ-lib are fully legal Java programs, and as such can take advantage of the Java compiler, debugger, etc. In as a result, we have modified VR to produce VR-lib which supports HJ-lib. In addition, VR-lib supports the phaser construct.

VR-lib works out of the box with JPF's PreciseDataListener to detect race conditions in HJ programs. For small programs this approach works well, producing results in a relatively short time period. However, for large programs, or even small programs with significant shared state, this approach is often intractable.

Gradual permission regions (GPRs) [**?**, **?**] show promise as an alternative to using JPF's PreciseDataListener. With GPRs, a programmer (or static tool) can signal to the runtime that a specific region of shared state is intended to be accessed with read/write permission levels in specific sections of the code. Simultaneous read/write access or write/write access to a GPR is illegal and should generate an exception in the runtime. Satisfying read/write permissions is a sufficient (but not necessary) condition for data race freedom; if a program execution (within JPF) terminates without throwing a permission exception for a given input, then all executions of the program are guaranteed to terminate with the same result, regardless of any scheduling decisions.

Section 2 of this paper presents an brief overview of HJ-lib, and Section 3 summarizes GPRs. Section 4 describes our use of JPF to implement verification of HJ programs that contain GPRs. Section 5 shows our preliminary results verifying programs using GPRs over explicit race detection via PreciseDataListener. Section 6 concludes this paper and highlights opportunities for future work in this area.

2. HABANERO OVERVIEW

The Habanero Java library supports asynchronous tasking, collective synchronization, mutual exclusion, message passing, and point-to-point synchronization with async, fin-

*Research funded by NSF grant CCF-1302524

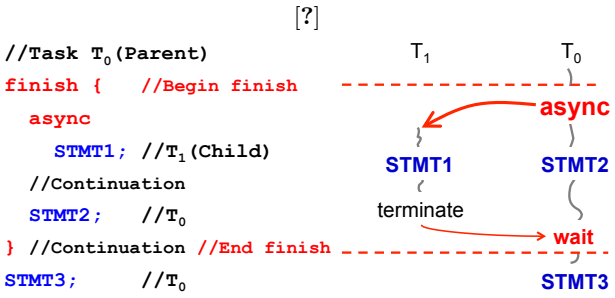

Figure 1: An example with async and finish.

```
finish(() -> {
    HjPhaser ph1 = newPhaser(SIG_WAIT);
    for (int i = 0; i < TASKS; i++) {
        // Tasks do work
        next(); //Synchronization with phaser
        // Tasks continue doing work
    }
});
```

Figure 2: Use of phasers to mimic barrier behavior

ish, isolated, actors, and phasers respectively [?]. Each of these primitives is accessed via library calls in HJ-lib. A brief description of each primitive is provided below, followed by a more thorough description for phasers. A more detailed description of all HJ primitives can be found in [?, ?].

- async: async(hjrunnable); generates an asynchronous task that executes the statements in the run method of the hjrunnable object, which can be specified succinctly in Java 8 as a lambda expression. See Figure ?? for the execution semantics.
- finish: finish(hjsuspendable); collects all spawned tasks generated in the body of the run method (or lambda) for hjsuspendable. As indicated in figure Figure ??, a finish construct waits for all asyncs in its scope to complete before transferring control to the continuation following the call to finish().
- isolated: isolated(hjrunnable); provides a guarantee that all interfering isolated regions will be executed in mutual exclusion. Interfering regions are defined as regions that access the same object with at least one of the regions containing a write-access.
- phasers: newPhaser(HjPhaserMode); creates a new phaser and registers the calling task with registration mode given by HjPhaserMode. Four possible modes are available: `signal`, `signal-wait`, `wait`, and `signal-wait-next` with each modifying the synchronization behavior of the registered task in relation to the newly created phaser. Variants of the async API enable a child task to be registered on a subset of phasers that the parent task is registered on.

Phasers extend barriers as a form of synchronization. They order execution of portions of the program into phases. Like barriers, they can restrict tasks from executing the next phase until the current phase is complete. However, unlike barriers, phasers allow tasks to specify point-to-point relationships on multiple phases.

All tasks that have designated themselves as signalers must signal the phaser in order for the phase to advanced.

```
// Wait Method
while (waiterPhase >= phase) {
    phaseCondition.await();
}
waiterPhase++;

//Signal Method
phaser.signal();
if (allSignalersSignaled) {
    phase++;
    phaseCondition.signalAll();
}
```

Figure 3: Implementation of signalers and waiters using conditional locks

```
// Write GPR
public Node pop() {
    acquireW(this);
    Node temp = this.pop();
    releaseW(this);
    return temp;
}

// Read GPR
public void empty() {
    acquireR(this);
    boolean ret = (size > 0) ? false : true;
    releaseR(this);
    return ret;
}
```

Figure 4: Stack methods integrated with GPRs

Waiters are required to stop at the phaser until the phase has been advanced. The exception to this rule are waiters that are registered on bounded phasers, in which the bound specifies a permissible "slack" in number of phases between waiters and signalers.

Signaler-Waiters function as both a signaler and a waiter. A phaser that has only signaler-waiters registered to it will function much like a traditional barrier (except that phasers allow tasks to dynamically join or leave the barrier.) Figure ?? illustrates the use of phasers as barriers.

For both signalers and waiters, synchronization is performed by a "next" call to the library. The next operation causes signalers to signal, waiters to wait, and signalers-waiters to do both.

VR-lib implements phasers using lock condition variables, as seen in Figure ??. Tasks registered as waiters wait on the lock until the condition is satisfied (all registered signalers have signaled for the respective phase). In VR-lib each Habanero task is represented as a thread. Thus we rely upon JPF's native scheduling of threads to explore all schedules introduced by the phaser.

3. GRADUAL PERMISSION REGIONS

GPRs function much like traditional race detection tools[?, ?]. However, the granularity of the dynamic checks is increased to sections of code instead of individual memory accesses. GPRs annotate variables over sections of code with either read or write permissions. Concurrent execution of tasks within two or more conflicting GPRs is detected at runtime as a violation of GPR rules and an exception is thrown. Conflicting GPRs are defined as regions that access

[?]

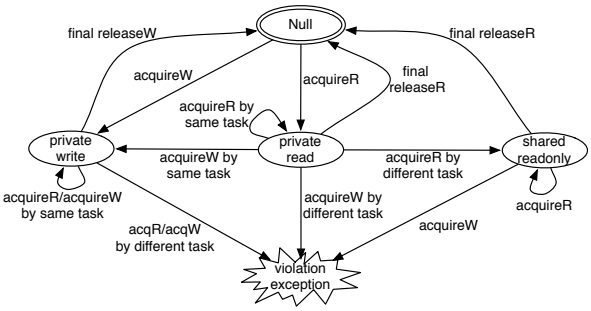

Figure 5: State machine for permission regions operating on a single object

the same variable with at least one of the regions being a "write" region. In general, GPRs can be used to annotate over multiple variables including parameters of a method with a single annotation. VR-lib does not yet support this feature. To protect multiple variables, multiple annotations must be used.

In VR-lib, a GPR is delimited by `acquireR/acquireW (Object X)` at the beginning and `releaseR/releaseW (Object X)` atthe end of a region, with R/W representing read and write respectively. A correctly annotated region uses the same object reference in the corresponding acquire/release statements. See Figure **??** for an example of using GPRs to annotate methods in a class. Note that the acquire/release calls are assertions of permissions rather than actions. A call to acquireR/acquireW asserts that it is safe to read/write the specified object in the region of code until the matching releaseR/releaseW call is encountered (unlike monitor-enter/monitor-exit operations on locks which perform operations in support of mutual exclusion). Thus, the verification problem becomes one of ensuring that no acquireR/acquireW call results in a permission exception. The standard data race detection problem can be modeled by wrapping each read/write operation in a separate permission region. Larger permission regions enable checking of invariants related to higher-level race conditions (while also guaranteeing the absence of low-level races), and—as a bonus—are amenable to more efficient verification than standard data race detection.

Our contribution to the work using GPRs is the implementation of GPRs within VR-lib, which allows for the utilization of JPF to perform dynamic checks for all legal schedules (for a given input). Thus, an HJ program, in which each shared variable is guarded by a GPRs annotation, can be verified by JPF to be free of race conditions. As a reminder, there are two important semantic guarantees for a large subset of HJ programs (those that use async, finish, future, or phaser constructs, but do not use isolated, actor, or data-driven tasks). First, if any program in this subset is guaranteed to be data race free, then it is also guaranteed to be determinate (functionally deterministic and structurally deterministic) [**?**]. Second, all programs in this subset are guaranteed to be deadlock-free.

4. JPF

```
// Permission Violation
public void race(Stack X) {
    // Task A
    async(() -> {
        acquireW(X);
        X.push(5);
        releaseW(X);
    };

    // Task B
    async(() -> {
        acquireR(X);
        X.peek();
        releaseR(X);
    };
}
```

Figure 6: Permission violation while operating on a stack

We have chosen to use a combination of a custom listener, a scheduler factory, and attribute info to extend JPF to perform permission region verification. A brief description of each follows.

Our listener, HjListener, extends JPF's property adapter class. Consequently it inherits the behavior of the property adapter class, but with two modifications: 1) executeInstruction and 2) instructionExecuted. In JPF, these two methods are used to observe/modify the system state before an instruction has been executed (executeInstruction) or after the instruction has been executed (instructionExecuted). In both cases, the method observes whether the instruction is a permission region relevant instruction (acquire/release). In the former case, HjListener will insert a choice generator with all runnable threads enabled and in the latter case it will perform appropriate permission checks on all read-/write accesses performed in the permission region on the annotated variable.

HjScheduler mimics the behavior of JPF's default schedule with one exception: choice generators are not inserted when threads access shared variables. Currently, JPF inserts shared access choice generators dynamically as it discovers sharing. JPF relies upon the placement of these choice generators to correctly detect race conditions. When a shared variable is discovered, JPF will insert a choice generator at the shared variable's location. This choice generator will cause JPF to interleave the execution of all runnable threads. If JPF (more precisely, JPF's PreciseRaceDetector) detects simultaneous read/write access of the shared variable it will notify the user of a data race. However, if one of the conflicting threads is no longer runnable (terminated, blocked, etc) then JPF will not report the error. In other words, JPF's partial-order reduction is incomplete. In contrast, when using GPRs, programmer annotations dictate the location of the inserted choice generators. As shown in previous work [**?**], a gradual type system can be used to reduce the annotation burden for programmers.

HjAttribute is a series of classes that handle the actual permission violation checks. Each object that is guarded by a permission region is assigned an attribute info. This attribute info is an object-specific state machine. This machine defines four valid states: Null, Private-Read, Private-

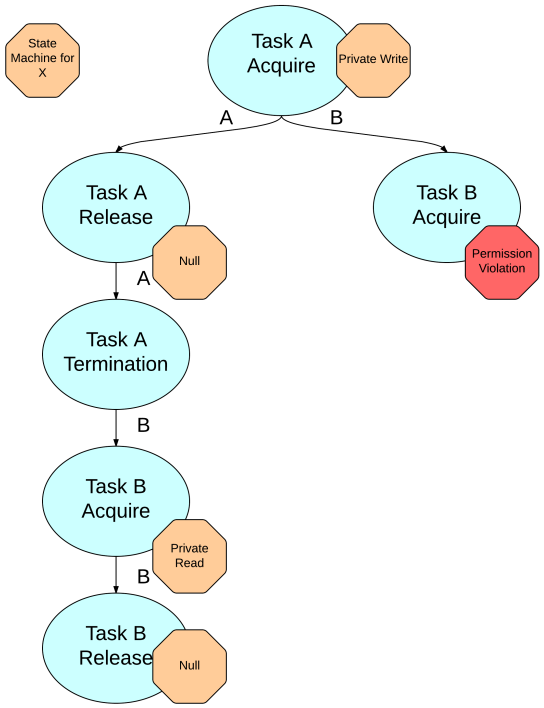

Figure 7: State space for permission violation example

Write, and Shared-Read. See Figure **??**. Null is the state of an object for which no task has acquired permissions. An object which has been acquired by a single task for read-/write permissions has state Private-Read/Private-Write respectively. Shared-Read is the state of an object when multiple tasks concurrently acquire read permissions.

When HjListener detects that an acquire/release statement has been executed on object X, it signals a state transition on the state machine belonging to X. If a legal transition is not defined then a runtime exception is thrown.

Upon backtracking, JPF performs a shallow copy on attribute infos. This means that the object itself will be restored, but none of the underlying changes to the object. HjAttribute was designed to be entirely immutable (transitions from state X to Y are simulated by replacing the machine in state X with a new machine in state Y); therefore, when JPF backtracks, all of the attribute infos are properly restored.

A walk-through of this system for a typical data race will highlight the interplay among different pieces of the system. Figure **??** shows an example of concurrent accesses on a common data structure (X). Since two threads have conflicting permission requirements that can occur in parallel, we expect that a permission violation should be detected.

The first task to begin execution will request permission acquisition on object X. HjListener will detect that an acquisition is pending and will insert a choice generator at the current location in the schedule. HjListener will also attach a state-machine to X as seen in Figure **??**. If task A is the first task to run then the state machine will be set to the private write state. JPF may now choose to either execute task B or finish the execution of task A. We will consider the case where task B is chosen. Task B will then request permission acquisition on X. HjListener will insert another choice generator and then will examine the state machine attached to X. There is no transition defined from the private write state to the private read state or vice-versa, thus an exception will be thrown. This exception is then reported by JPF as a permission violation. Similarly, if Task B executed first, then we'd get an error when task A tried to assert write permission.

Table 1: Benchmarks of HJ programs: Permission Regions vs. PreciseRaceDetector

		GPR		PRD	
Benchmark	Tasks	States	Time	States	Time
ConcReaders	2	168	00:00:00	1,977	00:00:01
PermStackFin	2	248	00:00:00	3,682	00:00:02
PermStackIso	2	1,751	00:00:02	64,879	00:00:18
PhaserTest	2	29,480	00:00:16	2,240,530	00:15:37

5. RESULTS

For comparison between JPF's PreciseRaceDetector and permission regions we collected a series of measurements on benchmarks that perform work in parallel. These benchmarks utilize a wide variety of HJ features, including: async, isolated, finish, and phasers. Increasing the complexity level of synchronization (finish $<$ isolated $<$ phasers) increased both the state-space and the running time in a significant way (as we descend the rows in Table **??** we increase synchronization complexity). However, the use of GPRs reduces the state-space by as little as one order of magnitude and as much as two orders of magnitude. The reduction in overhead can likely be attributed to the lack of shared-access choice generators.

6. CONCLUSIONS & FUTURE WORK

To summarize, we have introduced VR-lib for model checking of HJ programs in JPF. VR-lib supports the use of phasers as well as GPRs. GPRs in combination with JPF has been shown to be an effective strategy for model checking small programs for race conditions. Opportunities for future improvements in this area are as follows:

- support for actors, exceptions, and finish accumulators in VR-lib
- automatic static insertion of GPR annotations (this will relieve the burden from the programmer and provide stronger guarantees that the program is annotated correctly)
- support of multi-variable annotation to increase readability of annotated programs
- test the effectiveness of GPRs for larger HJ programs

Source for VR-lib, including benchmarks, can be found at `https://jpf.byu.edu/hg/jpf-hj`.