2015 30th IEEE/ACM International Conference on Automated Software Engineering

Model Checking Task Parallel Programs using Gradual Permissions

Eric G Mercer and Peter Anderson
Brigham Young University
Provo, Utah, USA
Email: eric.mercer,anderson.peter@byu.edu

Nick Vrvilo and Vivek Sarkar
Rice University
Houston, TX, USA
Email: nick.vrvilo,vsarkar@rice.edu

Abstract—**Habanero is a task parallel programming model that provides correctness guarantees to the programmer. Even so, programs may contain data races that lead to non-determinism, which complicates debugging and verification. This paper presents a sound algorithm based on permission regions to prove data race and deadlock freedom in Habanero programs. Permission regions are user annotations to indicate the use of shared variables over spans of code. The verification algorithm restricts scheduling to permission region boundaries and isolation to reduce verification cost. The effectiveness of the algorithm is shown in benchmarks with an implementation in the Java Pathfinder (JPF) model checker. The implementation uses a verification specific library for Habanero that is tested using JPF for correctness. The results show significant reductions in cost, where cost is controlled with the size of the permission regions, at the risk of rejecting programs that are actually free of any data race or deadlock.**

I. Introduction

Despite the explosion in multi-core hardware for general purpose computing, writing programs to take advantage of the available processing power is often a task reserved for expert developers. The first programs from the uninitiated often have more in common with sequential execution than parallel performance due to excessive synchronization, or worse, those programs are fraught with concurrency errors due to an absence of needed synchronization.

The Habanero extreme scale software research project intends to bring multi-core programming to the masses through languages and frameworks for non-experts. Habanero Java itself is a task-parallel programming model built around lightweight asynchronous tasks and data transfers [1]. The programmer in the Habanero framework focuses on the high-level task constructs using simple annotations and delegates to the Habanero run-time the burden of how to correctly and efficiently implement and synchronize those constructs.

The Habanero programming model offers correctness guarantees by defining safe subsets of the language that preserve properties over concurrent interactions such as determinism, serialization, and deadlock freedom; however many of these properties rely on the absence of data races. Regardless of using a safe-subset of the language, there is no easy way to determine when and if a program is free of data races. As such, the problem of verification reduces in practice to *printf*-debugging, inefficient code inspection, and run-time failures.

Permission regions are program annotations that announce how a task interacts with specific shared objects (i.e., reading or writing), and over what region of code that interaction takes place [2]. During execution, auxiliary data structures track accesses at the region on the indicated variables and signal an error on any accesses that conflict with the permission annotations. Permission regions have been shown effective in dynamically detecting data races at run-time, while requiring only a small number of programmer annotations, for a Java implementation of Habanero (HJ) [2], [3].

This paper presents a sound model checking algorithm to prove a program, for a given input, free of data races, deadlocks, failed assertions, and exceptions based on permission regions. The algorithm treats permission regions as atomic blocks of read/write operations on shared memory to reduce the number of schedules that must be considered in the proof. The algorithm also enumerates all outcomes that arise from non-determinism in sequencing isolated atomic blocks (i.e., non-determinism that is intended by the programmer) to verify user defined assertions and exceptions. This paper includes a proof that the algorithm is sound for any Habanero program with a fixed input. As such, the cost of model checking a Habanero program is controlled with the size and number of the permission regions, at the risk of rejecting some programs that are actually free of data races.

The effectiveness of the algorithm is explored using a new implementation of HJ in the form of a verification library (HJ-V) and the Java Pathfinder model checker (JPF). HJ-V is intended for debugging, testing, and verification so it trades performance for simplicity and correctness by using Java threads for each task, and using global locks with conditions for features of Habanero that require mutual exclusion and complex synchronization. The library supports all of the constructs in the Habanero model including phasers. The implementation of permission regions with the new model checking algorithm is in an extension to JPF named JPF-HJ. An empirical study over several benchmarks comparing the cost of verification between JPF and JPF-HJ both using HJ-V show a significant reduction in the cost when using JPF-HJ that is dependent on the size and number of the permission regions with their interactions. The implementations of both HJ-V and JPF-HJ are available at `http://javapathfinder.org/jpf-hj/`

II. Habanero Java

The Habanero programming model is built around a task-parallel view of concurrency. Figure 1 is an HJ program using Habanero's most basic task constructs: *finish* and *async*. The `finish`-construct is a generalized join operation for collective

```
1  public static void main(final String[] s) {
2    Stack stk = initStack();
3
4    launchHabaneroApp(() -> {
5      finish(() -> {
6
7        async(() -> {
8          stk.push(5);
9        });
10
11       stk.peek();
12     });
13   });
14 }
```

Fig. 1. An HJ program snipet using the `async` and `finish` statements.

synchronization: the parent task executes and then waits until all tasks created within the `finish`-construct have completed (including transitively created tasks).

The `async`-construct is a mechanism for creating a new asynchronous task: the calling task (parent) creates a new task (child) to execute in parallel with the parent. The child can read or write any data in the heap and can read, but not write, any local variable belonging to the parent's lexical scope. A task created in an `async`-construct becomes ready for scheduling at the point it is declared in the program.

The program in Figure 1 enters a `finish`-construct (Line 5) where it creates a child task (Line 7) to write to the stack (Line 8). the parent task then inspects the stack (Line 11). The two stack operations are not ordered and execute logically in parallel. The parent blocks at the end of the `finish`-construct until the child task completes.

Other constructs in the Habanero model include: *isolated* and *actors* for mutual exclusion, *future* for passing data between tasks, and *phasers* for arbitrary point-to-point synchronization [1].

III. Permission Regions

Permission regions are programmer added annotations on shared objects [2], [3]. The regions are indicated as accessing shared objects in read or write mode. When the program executes with run-time checking of permission regions enabled, a state machine is associated with each shared object to track permissions on that object as indicated by the program annotations. If accesses from distinct tasks on the same object conflict (i.e., a read with a write or a write with a write), then a permission violation, indicative of a data race, is detected. An absence of violations implies an absence of any data races.

To annotate the program in Figure 1 with permission regions, Line 8 and Line 11 are wrapped in separate regions, writing and reading, respectively, as follows:

```
acquireW(stk);
stk.push(5);
releaseW(stk);
```

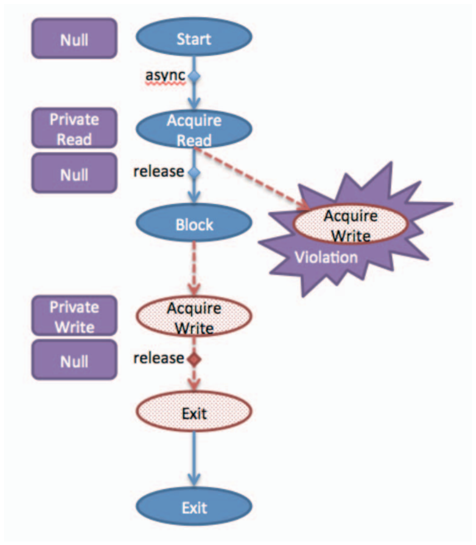

Fig. 2. Different schedules for a permission region annotated version of the program in Figure 1 with the schedule in the right branch reporting a permission violation.

```
acquireR(stk);
stk.peek();
releaseR(stk);
```

The state machine associated with each region to track accesses and detect violations is not shown due to space limitations, but it is intuitively understood from the two possible task schedules in Figure 2 for an annotated version of the program in Figure 1. The solid filled ovals and solid lines represent the parent task and the dotted filled ovals and dashed lines represent the child task created by the `async`-statement on Line 7. The squares indicate the current state of the state machine that is tracking accesses to the shared object `stk`.

The left branch of the tree is the schedule where the parent task runs until it is blocked to wait for the child task. The parent task acquires and releases private read privileges on the region and then the newly created task runs, acquiring and releasing private write privileges. If this schedule is followed in the run-time, then the permission violation in the program is undetected. The right branch is another possible schedule in the run-time. Here the child task runs just after the parent task acquires private read privileges on `stk`. When the child task tries to acquire write privileges on `stk`, its state machine detects the violation.

Permission regions are distinctly different from mutual exclusion primitives such as locks and the Habanero `isolated`-construct. The `isolated`-construct defines an atomic region that runs mutually exclusive to any other `isolated`-construct and can be used to express non-determination that is intended by the programmer. As such, isolated atomic regions are serialized with respect to one another. Permission regions do not include any serialization or synchronization semantics by themselves; rather, they check if concurrent accesses obey the permission annotations.

Algorithm 1 Permission Region Informed Search

```
 1: function SEARCH($t$, $h$, $T$)
 2:     loop: ($h$, $T$) := run($t$, $h$, $T$)
 3:
 4:     $s$ := status($t$, $T$)
 5:     permission_violation = false
 6:     if $s$ = PR_ENTRY then
 7:         ($h$, $T$, permission_violation) := acquire($t$, $h$, $T$)
 8:     else if $s$ = PR_EXIT then
 9:         ($h$, $T$) := release($t$, $h$, $T$)
10:         goto loop
11:     end if
12:
13:     if permission_violation then
14:         report permission_violation and exit
15:     end if
16:
17:     $R$ = runnable($T$)
18:     if $R = \emptyset$ then
19:         if blocked($T$) $\neq \emptyset$ then
20:             report deadlock and exit
21:         else
22:             report any uncovered sharing and exit
23:         end if
24:     end if
25:
26:     if $(h, T) \notin S$ then                    ▷ $S$ is a global variable
27:         $S = S \cup \{(h, T)\}$
28:         if $s$ = PR_ENTRY $\vee$ $s$ = ISOLATED then
29:             for all $t_i \in R$ do
30:                 search($t_i$, $h$, $T$)
31:             end for
32:         else
33:             $t_i$ := random($R$)
34:             search($t_i$, $h$, $T$)
35:         end if
36:     end if
37: end function
```

IV. JPF-HJ Search Algorithm

Permission regions create natural scheduling boundaries for model checking that can be leveraged to mitigate state explosion while preserving the essential behaviors of the program that lead to data races, deadlocks, assertion violations, or exceptions since they represent points of execution where sharing is expected. The intuition is that given a fixed program input, erroneous behavior can only arise from interactions between tasks on shared memory. As such, it is only necessary to preempt running tasks at the entrance to permission regions and `isolated`-constructs. If a program has any deadlocks, data races, assertion violations, or exceptions for a fixed program input, then such a deadlock, data race, assertion violation, or exception exists in one of the schedules that is explored from those preemption points.

Algorithm 1 is the pseudo-code for the algorithm to explore all task schedules created at entry to permission regions and `isolated`-constructs. The pseudo-code only covers the detection of data races (i.e., permission region violations) or deadlocks; though, assertion violations and exceptions can be detected similarly. The state of the program in the pseudo-code is simplified for clarity; it is represented by a heap, h, and a set of tasks, T. The lowercase t indicates a task. Line 2 updates the heap and pool of tasks by running task t until it blocks, exits, reaches a permission region boundary (i.e., entry or exit), or reaches an `isolated`-construct.

At the entry point of the permission region (PR_ENTRY), Line 7 updates the state machine for the acquired permissions on the object in the heap and checks to see if the acquisition signals a permission violation. At the exit point of the permission region (PR_EXIT), Line 9 updates the state machine for the released permissions on the object in the heap, and the algorithm restarts task t running anew at Line 2.

If there is a permission violation, then it is detected on Line 13. Similarly, Line 19 detects a deadlock. A deadlock state is indicated when there are no runnable tasks (i.e., $R = \emptyset$) and there exists tasks that are blocked. A report for either a permission violation or a deadlock includes a witness trace for validation and debugging. In the absence of a deadlock or a permission violation, and when there are simply no more tasks to run, Line 22 terminates the search and reports any detected sharing that was not annotated by a permission region or covered by an `isolated`-construct. Such sharing is detected by tracking tasks on every heap access.

The set S on Line 26 is a global set to track the visited states. Line 29 does the actual scheduling by considering all runnable tasks, including the currently running task t, as a next task to run. Note that in the current state, if the task t is preempted because it enters a permission region, then that state reflects the acquired permissions on that region. In the case that task t blocked, Line 33 chooses a random runnable task to schedule next.

Figure 2, shown previously, is the state space explored by the search algorithm for the permission region annotated version of the program in Figure 1. Recall that the example has two tasks that access the shared object `stk`: one reading and the other writing. The ovals in the diagram represent scheduling points, and as before, the blocks on the left represent the state of the state machine tracking permissions. As indicated by the pseudo-code, the algorithm only preempts running tasks at the entrance to permission regions. In this example, it schedules the child task after the main task acquires read permissions to elicit the permission violation. By observation, if the permission regions in the annotated program were replaced with `isolated`-constructs, then the explored state space would no longer include the violation, but it would include all schedules that interleave the atomic blocks defined by the `isolated`-constructs.

Algorithm 2 is a procedural flow describing the process of program validation using the new search in Algorithm 1. When Algorithm 1 finishes, the algorithm reports any heap locations that have been accessed by more than one distinct task outside a permission region or an `isolated`-construct with the input program location where that access occurs. Using this information, a user is able to manually annotate the program location appropriately, and then repeat the search. The process terminates when a permission violation or a deadlock is discovered, or no more sharing outside of permission regions or `isolated`-constructs exists.

Algorithm 2 Procedure to Validate a Program

```
procedure VALIDATE($p$)
    ($h$, $T$) := init($p$)
    $R$ := runnable($T$)
    $t$ := random($R$)
    $S := \emptyset$
    search($t$, $h$, $T$)
    while uncovered sharing is reported do
        Add permissions or isolated on sharing
        ($h$, $T$) = init($p$)
        $S := \emptyset$
        search($t$, $h$, $T$)
    end while
end procedure
```

Theorem 1. *Algorithm 1 is sound in that it only accepts programs that have no permission violations or deadlock on a given input under the restriction that the programs terminate and have all sharing correctly annotated with permission regions or wrapped in `isolated`-constructs.*

Proof: The soundness proof reasons over a slightly modified version of the algorithm that is iterative and takes as an additional input a search tree, which is similar to Figure 2, that captures all possible sequences of release and acquire statements explored thus far. The algorithm traverses that input tree and at each leaf node tries to extend that node by one generation if possible. After the traversal, the algorithm returns the new tree. The algorithm is called in an iterative manner until the tree reaches a fix-point (which is guaranteed since the program terminates).

Let $P(n)$ be the statement that this modified search algorithm returns all interesting sequences of acquire and release statements of length n or less for a given input program, where interesting means containing a permission violation or deadlock.

Basis Step: the algorithm produces all interesting sequences of length $n \leq 1$. This case is trivially established with the initial state of the program that represents a sequence of length $n \leq 1$ and cannot contain a permission violation or deadlock since the program has not yet done anything. As such, it includes all interesting sequences.

Inductive Step: assume the modified algorithm has correctly generated a tree representing all interesting sequences of n or less; it is necessary to show that from such a tree the algorithm is able to generate all interesting sequences of length $n + 1$ or less. There are three possible outcomes at any leaf of the input tree:

1) the leaf cannot be extended as it is already an interesting sequence having a permission violation or deadlock;
2) the leaf cannot be extended as there are no more tasks to run, in which case it is not interesting; or
3) the leaf is able to be extended with one or more immediate descendants.

The first two cases are directly covered by Line 13 through Line 24 of the algorithm; there is no way to have any descendants in those situations and the sequences are already classified as interesting or not.

For the third case, first consider Line 28 of the algorithm that creates the next generation in the tree for permission regions and `isolated`-constructs. Every runnable task is scheduled (Line 29) and each of those tasks must reach an immediate successor. Such a successor may be a permission violation or a deadlock, making it an interesting sequence, a preemption, a block condition, or exit by the constraint that the input program must terminate. As such, any $n + 1$ length sequence that exists, is generated.

Further, any interesting $n + 1$ sequence is generated. To see this outcome, it is important to understand that the order of acquisition relative to read or write does not matter in detecting a violation. The state machines on the objects are not dependent on acquisition order; they only depend on what tasks hold read or write permissions at the time of acquisition. As the algorithm always first acquires a permission and then schedules other tasks, it generates all the interesting $n + 1$ sequences if any exist. In this case, a sequence is interesting due to a permission violation. If a permission violation does not exist in a sequence, then a deadlock is detected as usual.

To complete the inductive step, the code under Line 32 must be considered. That code covers a blocked or exited task. The input program has all sharing annotated or `isolated` by the theorem statement, meaning that any non-determinism due to scheduling is enumerated by Line 29 so all reachable program paths on the input are considered. If an interesting sequence exists because of a deadlock, then it is either found in the $n + 1$ step, by having selected the correct task, or in a later step when the correct task is chosen. If the deadlock depends on a particular sequence of task executions, then those sequences are enumerated by Line 29. As such, the deadlock is either deterministic (i.e., independent of the schedule) or non-deterministic (i.e., a product of a data race on some shared object). In the former, the choice of task does not matter, and in the latter, Line 29 enumerates all possible orders over permission region blocks and isolated blocks. ■

As a side note, Algorithm 1 is complete when all regions cover a single operation (i.e., an individual byte-code in the case of Java). Such completeness is at the cost of the number of explored schedules.

V. Results

A new Java library was implemented to evaluate Algorithm 1 in the JPF model checker (HJ-V). The library uses Lambda support in Java 8, and it is purposed for verification in JPF. Algorithm 1 itself is an extension to JPF that implements permission regions and the search (JPF-HJ). Each is briefly discussed before the results from several benchmark programs are presented.

A. HJ Library for Verification

HJ-V is a new Java library implementation of the Habanero model designed specifically for debugging and verification. It consists of roughly 1,300 lines of code in 32 classes. Most of the classes address the programmer interface rather than the library internals. Figure 1 is the interface using Java 8 Lambda functions and is identical to other Java library implementations of the Habanero model [4].

HJ-V supports all of the constructs in the Habanero model including phasers. To increase confidence in the correctness of HJ-V, test cases were created to utilize specific features of the runtime. Each of these test cases were run within JPF with full scheduling enabled (i.e., it schedules on every bytecode related to thread synchronization or sharing). Thus, for each case, JPF is used to determine that HJ-V is free of data races and deadlocks. In total, 22 test cases were created consisting of approximately 1,000 lines of source code.

B. JPF Implementation

The implementation of permission regions in JPF spans 1,036 lines of code and covers 11 distinct class objects. It leverages JPF's ability to track thread IDs of all accesses to objects, so it not only reports violations on the permission regions, but it also identifies shared accesses that are not annotated by permission regions or covered by `isolated`-constructs. In this way, JPF updates the user when a shared access has been missed in the annotations.

The implementation uses two key features of JPF: byte-code listeners and object attributes. It installs a byte-code listener to watch for instances of the byte-code that calls methods. The actual methods for the permission regions interface are empty stubs, and when the listener activates on the interface, it gets the method's parameters from the stack and updates the associated state machines appropriately. The state machines themselves reside in an attribute of the object [5]. The important property of attributes is that they follow heap objects through the entirety of state space exploration. For arrays, a separate permissions state machine is stored for every index.

The JPF implementation of Algorithm 1 exploits the extensible nature of the tool by providing a new *scheduling-factory*. A scheduling-factory is activated on preemption, when a thread is no longer able to run, or if there is input non-determinism. It decides what threads are scheduled by inserting *choice-generators* into the state search to enumerate the available choices. The search iterates over those choices starting a new search for each choice.

The default scheduling-factory of JPF is replaced with a new factory that does not insert any choices on thread actions, locks, synchronization, or shared accesses to objects. Anything related to concurrency is turned off except for forced context switches such as a thread exiting or a thread blocking. In those cases, the new scheduling-factory inserts a choice generator with a single choice that represents a random thread that is runnable.

To insert the preemption points for permission regions and `isolated`-constructs, the byte-code listener from the implementation of permission regions is extended to also listen for the calls to `isolated`. At the entrance to permission regions, the permission regions' state machine for the object is updated as before, but after the update, a choice-generator is inserted into the search that includes choices for all runnable threads. Similarly, a choice-generator is inserted at the `isolated` call. The entire factory with the listener extension is only a few hundred lines of code but significantly reduces the verification cost.

Benchmark Results

JPF's default code to detect data races is named *PreciseRaceDetector*. Table I compares the performance of JPF-HJ with *PreciseRaceDetector* over several benchmark programs taken from materials used to teach the Habanero model or test the Habanero runtimes. These programs are not necessarily indicative of actual Habanero programs in the real world but do contain the breadth of HJ constructs: `async`, `isolated`, `finish`, `future`, and phasers. Many benchmarks also include arrays and shared arrays. The sizes of the benchmarks are indicated by the *SLOC* column (i.e., the number of program locations) and the *Tasks* column (i.e., the number of tasks created).

In Table I, an entry of N/A in the time column indicates a running time greater than 30 minutes on a Macbook Pro, core I7, with 8 Gb of ram. The *Error* column indicates if a data race or deadlock is discovered. An entry of *Detected Race** is an incorrect result: the program is actually free from data races.

The first thing to notice in the table is that the *PreciseRaceDetector* is more likely to not complete in the time bound due to exploring an excessive number of task schedules: it schedules on individual byte-codes where JPF-HJ schedules on regions. The second thing to notice is that the *PreciseRaceDetector* reports data races in *TwoDimArrays*, *Add*, and *ScalarMultiply* benchmarks where no data races exists. This error report is a limitation of *PreciseRaceDetector* where it is not able to discern accesses to the arrays on disjoint indexes; rather, it reports an error because the access on the array object looks like a data race and exits. If JPF is configured to not exit on error discovery, then the entries become N/A. That is why the *Time* entries on those examples appear to improve over the entries for JPF-HJ.

The final thing to notice in the table is that the cost of verification is difficult to predict based solely on the static number of region annotations, as it depends not just on that number, but the size of the regions, and how many tasks contain the regions. For example, the *PrimeNumberCounter* and its variants have very few regions, as defined by the annotations in the code, but those regions are part of every created task, so those benchmarks have a much larger number of schedules to explore.

VI. Related Work

The algorithm introduced in this paper is related to existing algorithms that only schedule on synchronizations [6], [7], [8], [9], [10]. Grouping synchronizations in regions trades completeness for efficiency in a manner similar to bounding. Prior methods using permission regions only detect violations at running time and do not search the schedule space [2]; though it is possible to use a gradual type system to enforce guarantees at compile time [3]. An early version of a verification specific run-time for Habanero requires a specialized Habanero Java compiler and does not implement all the Habanero constructs (most notably `future` and phasers) [11]. A preliminary version of JPF-HJ details how to extend JPF but does not define the search algorithm, prove its correctness, provide support for arrays, or provide support for `isolated`-constructs [12].

Other approaches to verify task-parallel languages create new virtual machines with verification support [13], or spe-

TABLE I. PERFORMANCE COMPARISON BETWEEN JPF-HJ AND *PreciseRaceDetector*

			Permission Regions				PreciseRaceDetector		
Test ID	SLOC	Tasks	States	Time	Regions	Error Note	States	Time	Error Note
PrimitiveArrayNoRace	29	3	5	0:00:00	0	No Race	11,852	0:00:00	No Race
PrimitiveArrayRace	39	3	5	0:00:00	2	No Race	220	0:00:00	Detected Race
TwoDimArrays	30	11	15	0:00:00	0	No Race	597	0:00:00	Detected Race*
ForAllWithIterable	38	2	9	0:00:00	0	No Race	N/A	N/A	N/A
IntegerCounterIsolated	54	10	1,013,102	0:05:53	3	No Race	N/A	N/A	N/A
PipelineWithFutures	69	5	34	0:00:00	1	No Race	N/A	N/A	N/A
SubstringSearch	83	59	8	0:00:00	2	Detected Race	N/A	N/A	N/A
BinaryTrees	80	525	632	0:00:03	0	No Race	N/A	N/A	N/A
PrimeNumCounter	51	25	231,136	0:01:08	2	No Race	N/A	N/A	N/A
PrimeNumCounterForAll	52	25	6	0:00:00	2	Detected Race*	N/A	N/A	N/A
PrimeNumCounterForAsync	44	11	449,511	0:02:51	2	No Race	N/A	N/A	N/A
ReciprocalArraySum	58	2	32	0:00:06	2	No Race	N/A	N/A	N/A
Add	67	3	62,374	0:00:33	6	No Race	4,930	0:00:03	Detected Race*
ScalarMultiply	55	3	55,712	0:00:30	2	No Race	826	0:00:01	Detected Race*
VectorAdd	50	3	17	0:00:00	4	No Race	46,394	0:00:19	No Race
AsyncTest1	23	51	54	0:00:00	0	No Race	N/A	N/A	N/A
AsyncTest2	32	3	4	0:00:00	2	Detected Race	11,534	0:00:04	Detected Race
FinishTest1	32	3	6	0:00:00	0	No Race	2,354	0:00:02	No Race
FinishTest2	33	3	5	0:00:00	0	No Race	25,243	0:00:09	No Race
FinishTest3	44	4	7	0:00:00	0	No Race	34,459	0:00:12	No Race
ClumpedAccess	30	3	15	0:00:00	2	No Race	N/A	N/A	N/A

cialize existing verification tools to work with existing run-times [14]. These assume the run-times correctly implement the language. Dynamic methods that intercept the run-time calls or instrument the program input can be effective for verification using stateless search or bounding techniques [7], [8]. Bounded techniques apply equally well to the algorithm in this paper. Static method to find data races often have better performance but are more likely to reject correct programs

VII. CONCLUSIONS & FUTURE WORK

This paper presents a model checking algorithm to prove when a Habanero program does not contain any data races, deadlocks, assertion violations, or exceptions for a given program input. The algorithm, based on permission regions, only considers scheduling points in the search tree at the boundaries of permission regions and `isolated`-constructs. The paper includes a proof of soundness for the algorithm, meaning that the algorithm may reject a correct program due to the size of the permission regions.

The effectiveness of the algorithm is shown in several benchmark programs that cover many of the Habanero concurrency constructs. The analysis is done using a new Java library implementation of the Habanero runtime that is intended for debugging and verification. The new algorithm, with permission regions, is implemented as an extension to the JPF model checker. The results from the benchmark programs indicate a significant cost reduction when using the new algorithm.

Future work includes automating the annotation of permission regions based on the sharing detection in JPF; automating the validation of any counter-example; developing techniques to automatically refine permission regions from counter-examples when needed; a partial order reduction over permission regions; static-analysis to prevent scheduling on regions that cannot race; applying symbolic techniques to reason over input; and studying benchmarks that are representative of real world Habanero programs.

REFERENCES

[1] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar, "Habanero-Java: the new adventures of old X10," in *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, ser. PPPJ '11. New York, NY, USA: ACM, 2011, pp. 51–61.

[2] E. Westbrook, J. Zhao, Z. Budimlić, and V. Sarkar, "Permission regions for race-free parallelism," in *Proceedings of the Second international conference on Runtime verification*, ser. RV'11. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 94–109.

[3] ——, "Practical permissions for race-free parallelism," in *Proceedings of the 26th European conference on Object-Oriented Programming*, ser. ECOOP'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 614–639.

[4] S. Imam and V. Sarkar, "Habanero-Java library: A Java 8 framework for multicore programming," in *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, ser. PPPJ '14. New York, NY, USA: ACM, 2014, pp. 75–86.

[5] C. S. Păsăreanu, W. Visser, D. H. Bushnell, J. Geldenhuys, P. C. Mehlitz, and N. Rungta, "Symbolic Pathfinder: integrating symbolic execution with model checking for Java bytecode analysis," *Autom. Softw. Eng.*, vol. 20, no. 3, pp. 391–425, 2013.

[6] S. Vakkalanka, G. Gopalakrishnan, and R. M. Kirby, "Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings," in *Proceedings of the 20th international conference on Computer Aided Verification*, ser. CAV '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 66–79.

[7] M. Musuvathi and S. Qadeer, "Iterative Context Bounding for Systematic Testing of Multithreaded Programs." in *PLDI*, J. Ferrante and K. S. McKinley, Eds. ACM, 2007, pp. 446–455.

[8] M. Emmi, S. Qadeer, and Z. Rakamarić, "Delay-bounded scheduling," in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '11. New York, NY, USA: ACM, 2011, pp. 411–422.

[9] P. Godefroid, "Model Checking for Programming Languages Using Verisoft." in *POPL*, 1997, pp. 174–186.

[10] C. Flanagan and S. N. Freund, "FastTrack: Efficient and precise dynamic race detection," *SIGPLAN Not.*, vol. 44, no. 6, pp. 121–133, Jun. 2009.

[11] P. Anderson, B. Chase, and E. Mercer, "JPF verification of Habanero Java programs," *SIGSOFT Softw. Eng. Notes*, vol. 39, no. 1, pp. 1–7, Feb. 2014.

[12] P. Anderson, N. Vrvilo, E. Mercer, and V. Sarkar, "JPF verification of Habanero Java programs using gradual type permission regions," *SIGSOFT Softw. Eng. Notes*, vol. 40, no. 1, pp. 1–5, Feb. 2015.

[13] T. Zirkel, S. Siegel, and T. McClory, "Automated verification of Chapel programs using model checking and symbolic execution," in *NASA Formal Methods*, ser. Lecture Notes in Computer Science, G. Brat, N. Rungta, and A. Venet, Eds. Springer Berlin Heidelberg, 2013, vol. 7871, pp. 198–212.

[14] M. Gligoric, P. C. Mehlitz, and D. Marinov, "X10X: Model checking a new programming language with an "old" model checker." in *ICST*, G. Antoniol, A. Bertolino, and Y. Labiche, Eds. IEEE, 2012, pp. 11–20.