Permission Regions for Race-Free Parallelism

Edwin Westbrook Jisheng Zhao Zoran Budimlić Vivek Sarkar
Rice University
{*emw4, jisheng.zhao, zoran, vsarkar*}*@rice.edu*

Abstract. It is difficult to write parallel programs that are correct. This is because of the potential for *data races*, where parallel tasks can modify each other's data in complex and unexpected ways. A classic approach to this problem is dynamic race detection, which has the benefits over other approaches that it works transparently to the programmer and it can work with any pattern of synchronization. Unfortunately, dynamic race detection is very slow; further, it can only detect low-level races, not high-level races, also known as atomicity violations. In this paper, we present a new approach to dynamic race detection that works by inserting constructs called permission regions, which specify regions of code that have permission to read or write certain variables. Dynamic checks are then used to ensure that no conflicting permission regions execute in parallel, essentially increasing the granularity of checks in dynamic race detection. We demonstrate that permission regions can be used to achieve significantly better performance than dynamic race detection while maintaining the scalability of the uninstrumented code, to the point where they could be used in actual production code.

1 Introduction

As Moore's Law becomes obsolete and chip manufacturers turn towards multi-core for performance, parallel programming is becoming more and more essential for scalability. Unfortunately, it is difficult to write parallel programs that are correct because of the potential for *data races*, where parallel tasks can modify each other's data in complex and unexpected ways. Except rare cases of parallelism experts writing race-tolerant code, a data race is a bug, as it can cause code to run in ways that were not intended by the programmer. It is a more devious sort of bug than most, however, because it is not even possible to know when a data race occurs.

A classic approach to dealing with data races is dynamic race detection [17, 21, 30, 20, 32, 36, 18]. Under this approach, each memory access of a program is instrumented to check, at runtime, whether it conflicts with a parallel access. This approach has two powerful benefits: it works transparently to the programmer, since the instrumentation is done by the compiler or runtime system; and it can work with any pattern of synchronization, including barriers and lock-free concurrent data structures, since the dynamic checks need not be aware of how synchronization is achieved. This last point is in contrast with many other approaches, such as static race detection [12, 38, 4, 19, 3], transactional

memory [28], or approaches to deterministic parallelism [15, 35], which generally require very specific approaches to synchronization and parallelism. Unfortunately, dynamic race detection is very slow, limiting its usefulness in practice. In addition, dynamic race detection cannot detect *high-level* races, or atomicity violations, where a task modifies the data of another in the abscence of standard, or *low-level*, data races. [1]. High-level races are especially insidious because they depend on programmer intent, and can occur even in well-synchronized code.

In this paper, we introduce a new programming language model that enables a form of "always on" race detection for both low- and high-level races. More specifically, our approach enforces a property which we call the *permission property*, which ensures that no task is permitted to write to a memory location while another task has permission to access that location. The permission property is stronger than race-freedom, and in fact corresponds to the way most programmers write code. To enforce this property, we introduce a new construct called a *permission region*. These constructs mark a region of code with read and write sets of variables, to indicate that the region has permission to read or write those variables while it executes. Two permission regions are said to *conflict* when the write set of one overlaps the read or write set of the other. The runtime system then ensures that no two conflicting permission regions execute in parallel, throwing an exception if a conflict is detected.

Permission regions can be seen as an extension of dynamic race detection that increases the granularity of dynamic checks to entire regions of code, instead of to individual memory accesses. They are inserted by the compiler in a manner that is guaranteed to ensure race-freedom, but can also be used by the user to increase the granularity of checking even further, specifying atomicity requirements of their code. User insertion can also increase performance and scalability.

The rest of the paper is organized as follows. Section 2 introduces the Habanero-Java (HJ) parallel programming language and the Java Memory Model. Section 3 introduces permission regions as an extension of HJ. Section 4 presents compiler techniques to automatically insert permission regions into HJ programs. Section 5 shows the performance evaluation of our implementation of permission regions on a set of HJ benchmarks. Section 6 discusses related work, and Section 7 presents conclusions and directions for future work.

2 Background: Data Races in Habanero-Java

In this section, we briefly introduce Habanero Java (HJ) [23] and explain low- and high-level races. HJ is an extension of Java with several constructs for parallelism and synchronization; in this paper, we consider only **async** and **finish**, which respectively spawn a child task and wait for all tasks spawned in a lexically-scoped block to complete.[2] These constructs are borrowed from X10 [16], and are similar to the `spawn` and `sync` constructs of Cilk, respectively.

[1] We borrow the terms "low-level race" and "high-level race" from Artho et al. [2].

[2] We use the term "task" here instead of "thread" to distinguish semantically parallel tasks from the OS threads to which they might be mapped by an implementation.

```
void push (SNode n) {
  n.next = this.next;
  this.next = n;
}
```

(a) Task 1

```
SNode pop () {
  SNode tmp = this.next;
  if (tmp != null)
    this.next = tmp.next;
  return tmp;
}
```

(b) Task 2

Fig. 1. A Simple Data Race

A low-level race in Java, and similarly in HJ, is defined by the Java Memory Model (JMM). We refer the reader to other work [22, 29, 39] for the technical details, but conceptually a low-level race occurs when two accesses to the same memory location, one of which is a write, occur in distinct tasks without some form of synchronization between them. The difficulty of low-level races is that, in their presence, the actions of a task can appear to happen in a different order to other tasks running in parallel. When a program has no low-level races, however, then the JMM ensures *sequential consistency* (SC), meaning it behaves as if each instruction of each task appears to be atomic. When there are low-level races, however, the possible behavior can be quite complex, making the program dificult to understand.

As an example, consider the two tasks depicted in Figure 1, which perform a push and a pop of the stack **this** in parallel. (**this** is assumed to be the same in both tasks.) Since there is no synchronization between them, the write of **this**`.next` in task 1 has a low-level race with the read of the same field in task 2. This means that, under the JMM, task 1 can appear to occur in a different order to task 2, allowing task 2 to see the newly pushed node `n` with the old value of `n.next`. In this scenario, the third line of the `pop()` method in task 2 would set **this**`.next` to the old value of `n.next`, completely obliterating the remainder of the stack after **this**.

Even assuming SC (e.g., if every instruction were protected by a lock), the code may still execute incorrectly if task 1 runs to completion directly after the read of **this**`.next` in task 2, since `n` would be removed from the stack when task 2 sets **this**`.next`. This represents a high-level race, or atomicity violation, as task 2 intuitively assumes that **this**`.next` does not change between the read and the write of this field.

3 Permission Regions

The syntax of a permission region is as follows:

permit read$(x_1, \ldots, x_m)$ **write**$(y_1, \ldots, y_n)$ { *BODY* }

This statement executes *BODY* under the assertion that, while *BODY* executes, no conflicting **permit** statement will execute in a different task at the same time. We call the the variables x_i and y_j the *read* and *write variables* of the permission

```
void push (SNode n) {
  permit write(this,n) {
    n.next = next;
    next = n;
  }
}
```

(a) Thread 1

```
SNode pop () {
  permit write(this) {
    SNode tmp = next;
    if (tmp != null)
      permit read(tmp)
        next = n.next;
    return tmp;
  }
}
```

(b) Thread 2

Fig. 2. Adding **permit** to Figure 1

region, respectively, and the set of objects they refer to during execution the *read* and *write sets*. Two **permit** statements are said to be *conflicting* if the write set of one overlaps the read or write sets of the other. If a permission region begins executing while a conflicting permission region is already executing in parallel, an exception is thrown. Otherwise, the permission region's execution is guaranteed to be in isolation relative to its read and write sets. This means that the body cannot see any writes from another task after entering the permission region, nor can any parallel task see its writes until the permission region has completed. "Completion" includes both normal and exceptional exit from *BODY*.

As an example, Figure 2 shows how our compiler annotates the racy example of Figure 1. The `push()` method is annotated with a permission region whose write variables include **this**, the stack on which a stack node is pushed, and `n`, the stack node being pushed onto the current stack. This permission region ensures that the call to `push()` must have write permission to these two objects while it executes. The `pop()` method is annotated with two permission regions: the first has write variable **this**, since **this** may possibly be modified to remove the next element of the stack; the second has read variable `n`, which represents the top node of the stack, since the next element of the stack after `n` must be read. Again, these permission regions ensure that `pop()` must have these permission on these two variables. Thus if one of `push()` and `pop()` begins executing before the other completes, then that method will throw a permission violation exception.

Permission regions represent a combination of static and dynamic checks. Checking that two conflicting permission regions do not run in parallel is in general an undecidable problem, and although there has been much work on may-happen-in-parallel analysis (*e.g.,* [9]), such analysis must in general be conservative. Thus we leave happens-in-parallel checking as a dynamic check. To ensure the permissions property, however, we must also be sure that all reads and writes happen inside appropriate permission regions; i.e., writes to `x.f` may only occur inside a permission region whose write variables include `x`, and similarly, reads of `x.f` may only occur inside a permission region whose read or write variables include `x`. The algorithm used to insert these checks is discussed in Section 4. We now briefly summarize some of the salient points of the design of permission region.

Read and Write Variables can be Modified: It is allowed for the variables in the variable set of a permission region to be modified in the body of the permission region. For example, the following code performs a loop inside a permission region which conditionally modifies the its write variable:

```
permit write(x) {
  while (...) {
    if (...) { x = ...; }
  }
}
```

This generalizes the semantics of permissions regions as follows: two **permit** statements are said to be conflicting if the *current values* of the write set of one overlaps the *current values* of the read or write sets of the other. Modifying a read or write variable can also cause a permission region to come into conflict with a concurrently executing permission region, and thus assignments to such variables, such as the assignment to `x` above, can cause data race exceptions to be thrown.

Final Fields: Under the JMM, reading **final** fields do not is never considered a data race. Similarly, we allow such fields to be read without inserting any permission regions.

Static Fields: Fields marked as **static** are global, and are not associated with a particular object. Thus we also allow **static** fields to be read or write variables in permission regions, where conflicts involving **static** fields can only occur between regions that both use the field itself, not the value pointed to by the field.

Constructors: The bodies of constructors are always implicitly contained inside a permission region with write variable **this**, as the purpose of a constructor is to initialize an object before it is used. Thus, although parallelism is allowed in constructors, passing **this** to another task in a constructor will cause an exception if the other task tries to access **this** before the constructor finishes.

Array Views: In order to support array-based parallelism (where tasks process pieces of an array in parallel), a permission region can specify pieces of an array in its read or write sets. This specification is supported by having users access arrays through *array views* [37, 27], which are objects in HJ that represent pieces, or sets of cells, of an array. The notion of conflicting permission regions is then extended to dynamically check for array views whose sets of cells overlap.

Inter-Method Permissions: It can often be useful to have permission regions cross method boundaries. For example, accessor methods are often used in the context of a more complex operation which is intended to be strongly isolated, and thus requiring a permission region on **this** seems an extra source of overhead. To allow inter-method permission regions, we introduce *permission method annotations*. These take the form of two new keywords allowed in method signatures, **reading**

and **writing**, which mark arguments in a method signature that must be in the read or write variables, respectively, of an enclosing permission region when the method is called. The implicit **this** argument can also be modified with these keywords by applying the keyword to an entire method, i.e., by listing the keyword in the method signature before the return type.

For example, we could change the signature of the `push()` method of Figure 2 as follows:

```
void writing push (writing SNode n)
```

This states that any calls to `p.push(q)` must always occur inside permission regions for `p` and `q`. In turn, the compiler need not insert the permission regions for this method given in Figure 2. This can be useful to reduce the number of dynamic checks performed at runtime. It also allows the user to state stronger atomicity requirements; for example, code containing three consecutive calls to `push()` with this new signature is guaranteed to push all three elements in order with no intervening pushes or pops in parallel, since such parallel accesses would result in an exception.

4 Compiler Insertion of Permission Regions

In this section, we describe the algorithm our compiler uses to insert permission regions. The basic assumption of the insertion algorithm is that, in general, a programmer does not intend for an object to be modified in parallel while that object is in scope. Thus our algorithm essentially tries to match permission regions to variable scopes. Naturally, this approach will not always exactly capture the programmer's intent; i.e., this approach can lead to spurious exceptions, or false positives, when the original code had no data races. (This is one reason why permission regions are exposed to the user as a construct in the language.) However, this approach is *almost* always correct; in fact, for the 11 benchmarks discussed below in Section 5, accounting for about 9k lines of code, only one case was found that led to a false positive, other than the requirement that regular parallel application use array views in the manner discussed in Section 3. Note also that our algorithm does not insert any of the permission method annotations of Section 3, as these could potentially change the semantics of a program in ways the user did not intend.

We proceed as follows. Section 4.1 gives our insertion algorithm, while Section 4.2 describes two cases where this algorithm gives incorrect results and describes why these cases are rare.

4.1 The Insertion Algorithm

As discussed above, our algorithm essentially tries to match permission regions to variable scopes. This goal is modified by a number of concerns. First, permission regions do not cross **async** statements; in fact, inserting an permission region for `x` outside an **async** statement is drastically different than inserting it inside the body of the **async** statement, since the former means the parent

process can access `x` while the latter means the child task can access `x`. Second, if `x` is only accessed within the bodies of **isolated** statements — which specify critical sections in HJ (instead of using monitors like Java's **synchronized** keyword) — then the algorithm assumes that `x` should only be accessed inside critical sections, and permission regions for `x` are only inserted inside the body of **isolated** statements. Finally, if the programmer explicitly writes a **permit** statement then the algorithm respects the placement of that permission region.

The inference algorithm works on a per-method basis by considering the abstract syntax tree (AST) of a method body. The algorithm first finds all nodes n in the AST where read or write access to each variable `x` is required such that n does not already occur inside of an appropriate permission region. Read or write access could be required either because of access to a field `x.f` or because of a method call that specifies **reading** or **writing** for an argument position for which `x` is passed. Next, for each such node n that requires access to `x` in the AST, the algorithm finds the highest ancestor a of n such that the path from a to n does not contain an **async** or an **isolated**. A permission region for `x` is then inserted around a in the AST, with `x` in the appropriate variable set.

4.2 Limitations of the Insertion Algorithm

The inference algorithm presented above yields false positiives in two potential programming patterns, which we call *intra-scope parallel access* and *task-dependent conditionals*. The first of these, intra-scope parallel access, is when a region of code that accesses `x` somehow passes `x` to a parallel task. Such code might look like this, where `compute()` performs some parallel computation on its argument:

```
x.f = ...; compute(x); ... = x.f;
```

In this case, the user does expect `x` to be accessed in parallel while `compute()` executes, and thus the proper placement of permission regions for `x` would be to have two regions, neither of which contains the call to `compute()`. Our algorithm, however, inserts a single region around the whole piece of code, yielding a false positive and requiring manual insertion by the user. This pattern occurred exactly once in our study of over 9000 lines of HJ benchmarks, specifically in the PDFS benchmark, so it is not incredibly common.

The second pattern that can lead to false positives, task-dependent conditionals, occurs when accesses to an object are guarded by a conditional that picks out a specific task, like this:

```
if (isTask1) { x.f = ...; }
```

Our inference algorithm will insert the permission region around the entire conditional; however, if this code is called in parallel by multiple tasks, where only one task has `isTask1` set to true, then the proper place for the permission region is arguably inside the conditional. This is a very rare programming pattern, though, that we have not seen in any of our benchmarks. Further, the problem only appears when the condition is guaranteed to hold for at most one parallel task, otherwise, there really is a potential race, which should indeed be reported.

Table 1. List of Benchmarks

Type	Benchmark Suite	Name	LoC	LoC for sub-views	# of reading	# of writing
Loop Parallelism	NPB	CG	1070	22	5	0
	JGF	Series	225	2	0	0
		LUFact	467	0	1	1
		SOR	175	4	0	0
		Crypt	402	4	0	0
		Moldyn	741	29	6	18
		RayTracer	810	22	31	22
Function Parallelism	BOTS	NQueens	95	0	1	0
		Fibnacci	70	0	0	0
		FFT	4480	209	0	0
		PDFS	537	0	0	2

5 Performance Evaluation

In this section we evaluate permission regions along two dimensions, performance and usability. To do this, we considered 11 benchmarks for HJ, including small- to large-scale benchmarks from the JavaGrande benchmark suite [26], the NAS Parallel Benchmark suite [13], the BOTS benchmark suite [10], and a Parallel Depth First Search application (PDFS). These are listed in Table 1, which also separates the benchmarks into loop vs functional parallelism in column 1 and gives the number of lines of code (LoC) in column 4.

For each benchmark, we performed the following experiment. We first converted any parallel array processing in the benchmark to use array views, as discussed in Section 3. Table 1 gives the number of lines of code that were modified in column 5. We then ran the code to determine if there were any false positives; as discussed above in Section 4, there was exactly one false positive in the PDFS benchmark. Next, we timed the benchmark with and without permission regions, to measure the slowdown of permission regions. Finally, for the 5 benchmarks with the biggest slowdowns, we added permission method annotations to key methods to increase performance and timed the results. The numbers of **reading** and **writing** keywords added to each benchmark are given in columns 6 and 7 of Table 1, respectively. All timing results were obtained on a 16-way (quad-socket, quad-core per socket) Intel Xeon 2.4GHz system with 30GB of memory, running Red Hat Linux (RHEL 5) and Sun JDK 1.6 64-bit version. We used the linux taskset command to physically restrict the number of cores involved in the experiment, from 1 to 16 cores, to measure scalability.

From a usability perspective, the results of our experiments were quite promising, when measured in terms of lines of code that must be modified to use permission regions. The biggest change required was modifying the benchmarks to use array views, requiring an average of 3% of the lines of code to be edited. Other work [37, 27] has demonstrated that array views are useful for other reasons as well, so this cannot be held against permission regions too seriously. Otherwise, only one permission region had to be added to remove a false positive, and the

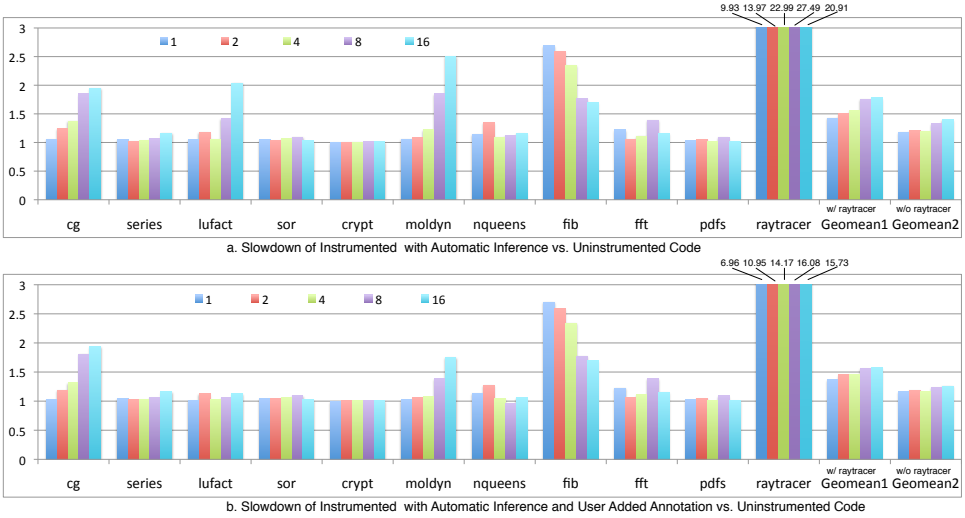

Fig. 3. Slowdown of Instrumented Code vs. Uninstrumented Code

"optimization" step of adding permission method annotation modified less than 1% of the code on average.

The timing results are given by the two graphs in Figure 3. These graphs give the slowdowns for each benchmark instrumented with permission regions as compared to the uninstrumented benchmark without permission regions, using 1, 2, 4, 8, and 16 cores. The first graph gives the slowdowns for the first timing experiment, after removing false positives, while the second gives those for the second timing experiment, including the permission method annotations. Most of the benchmarks run less than 2.5× slower than their uninstrumented versions, with a geometric mean around 1.5×. Compared with most of the state-of-art data race detection implementations [17, 30, 20, 2, 32, 36], which typically result in a slowdown of an order of magnitude or more, our overhead is relatively low. The main reason for the relatively low overhead is granularity; we are checking object permissions once for each region rather than for each memory access. In addition, the permission method annotations significantly improved performance of 3 of the 5 benchmarks with whic they were used, the `lufact`, `moldyn`, and `RayTracer` benchmarks.

One benchmark that deserves a separate discussion is the `RayTracer` benchmark which has a 27.49× slowdown when running on 8 threads. The reason for this drastic performance penalty is that `RayTracer` uses objects (3-dimensional Points) as the basic computation units, which forces the compiler to insert permission regions around each object access to ensure the correct permissions. These object accesses are done within the innermost loop of the main kernel, which does not have significant additional computation to hide the overhead. More advanced compiler optimizations such as loop interchange and loop un-

rolling should be able to enable the compiler to create large enough permission regions to eliminate a significant part of this overhead. This is future work, however.

6 Related Work

There have been significant recent work on runtime systems which detect low-level data races before they happen and throw exceptions, thus ensuring sequential consistency. DRFx [8, 14] is similar to our work except that all the regions are automatically inserted by the compiler. This and similar [11, 7] approaches cannot prevent high-level data races, which one of the main advantages of the permission regions described in this paper.

In Deterministic Parallel Java [35, 15], e ach object has to be associated with a specific *region* when allocated, which limits expressivity of the programming model. Also, methods must be annotated with effects, with an average of 12% of the lines of code requiring annotation.

In Transactional Memory [28], it is difficult to allow I/O within transactions since they may have to be restarted. Permission regions can have arbitrary code within them, including I/O code. The semantics of nested transaction and nested parallelism in transactional memory has also been a subject of much debate [1]. Permission regions offer a clear and intuitive semantic for nesting.

Dynamic race detection [17, 30, 20, 2, 32, 36] is not efficient enough to be "always on" as it may result in an order of magnitude slowdown over original code.

Type systems and static analyses that ensure shared accesses are guarded by appropriate locks or other guards [34, 12, 38, 31, 4, 19, 3]; are often too restrictive or cumbersome to use in general, preventing many concurrency patterns that are safe and useful in practice. Static analyses and model-checking [33, 25], in contrast, generally are incomplete and/or report false positives.

Also closely related are type systems based on linear types, such as fractional permissions [6, 5] and Scala capabilities [24]. Linear types can be used to control the number and allowed uses of active references to an object, allowing the programmer to express concepts such as uniqueness, immutability, and borrowing of an object. Unfortunately, linear type systems place complex restrictions on how objects can be used, often making it difficult for programmers to use them effectively. The present work can be seen as a "partially dynamic" approach to linear types, allowing linear capabilities to be acquired and changed at runtime.

There has also been much prior work on techniques that eliminate low-level data races. One approach is static race detection, which either checks that code properly uses locks and/or inserts proper locking into code [12, 38, 4, 19, 3]. Another approach is dynamic race detection, which instruments a program to detect possible low-level races at runtime [17, 30, 20, 2, 32, 36, 18]. Finally, a third approach is to give a fail-stop semantics for racy programs, throwing an exception if a low-level race occurs at runtime [8, 14]. Very little work exists that addresses high-level data races, however, and this work is either entirely based on correct

use of locks [34, 2] or on transactional memory [28, 1]. The former is unsatisfactory because many concurrency patterns, such as those based on array tiling, do not use locks. Transactional memory, although promising in many aspects, has performance issues when transactions are too big, cannot perform certain non-transactional actions such as spawning parallel tasks or performing system calls inside transactions, and seems to require special hardware for good performance.

7 Conclusions and Future Work

In this paper, we introduced the *permission regions* that enable application programmers to ensure that low-level or high-level data races will never occur during execution of their programs. The approach is based on the *permission property*: data should be only accessed by a single task in read-write mode, and any data that can be accessed by multiple tasks must be in read-only mode. Any violation of the permission property results in a `PermissionException` being thrown at runtime prior to any data access that may participate in a data race.

The foundation of our approach lies in the insertion of *read/write permission regions* in the program through a combination of 1) automatic inference, 2) manual insertion to avoid false positive `PermissionException`'s, and 3) manual insertion to improve the performance of permission checks across method call boundaries. Of the 11 benchmarks studied in this paper, 4 required no modification by the programmer for 2) and 3), and the changes made in the remaining 7 benchmarks impacted fewer than 5% of the lines of code. Further, no parallel programming expertise is necessary to understand permission regions, since these permission annotations can enable useful runtime checking for invariants in sequential programs as well. Finally, the overhead for checking permissions in our approach is far lower than that of state-of-the-art approaches for data race detection. The geometric mean of the slowdown relative to unchecked execution on 16 cores was only 1.58× when the outlying `raytracer` benchmark is included, and 1.26× if `raytracer` is excluded. Smaller slowdown factors were observed for fewer numbers of cores.

In contrast, the average slowdown reported by the state-of-the-art FastTrack dynamic low-level data race detector [21] for comparable benchmarks was 8.5× for fine-grained location-level analysis and 5.3× for coarse-grained object-level analysis. However, it is worth noting once again that the Permission Regions and FastTrack approaches address different problems *e.g.,* FastTrack does not require any user interaction but also offers no solution for high-level data races.

Permission regions offer a number of opportunities for future research. One direction is to explore approaches that catch `PermissionException`'s and perform some kind of remediation to avoid the problem entirely *e.g.,* by performing rollbacks and executing the conflicting tasks on a single worker. Another direction is to simply log permission conflicts instead of throwing an exception, and explore the use of conflict logs as debugging feedback at the end of program execution. Finally, as discussed in the paper, there is a natural complementarity

between permission regions and software transactions that offers new opportunities to explore hybrid combinations of both approaches.

References

1. Kunal Agrawal, Jeremy T. Fineman, and Jim Sukha. Nested parallelism in transactional memory. In *PPoPP '08*, pages 163–174, 2008.
2. Cyrille Artho, Klaus Havelund, and Armin Biere. High-level data races. *STVR'03*, 13(4):207–227, 2003.
3. David F. Bacon, Robert E. Strom, and Ashis Tarafdar. Guava: a dialect of java without data races. In *OOPSLA '00*, pages 382–400, 2000.
4. Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA '02*, pages 211–230, 2002.
5. John Boyland. Checking interference with fractional permissions. In *SAS '03*, pages 55–72, 2003.
6. John Boyland, William Retert, and Yang Zhao. Comprehending annotations on object-oriented programs using fractional permissions. In *IWACO'09*, 2009.
7. Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: a race and transaction-aware java runtime. In *PLDI '07*, 2007.
8. Abhayendra Singh et. al. Efficient processor support for DRFx, a memory model with exceptions. In *ASPLOS'11*, 2011.
9. Agarwal et al. May-happen-in-parallel analysis of x10 programs. *PPoPP'07*, 2007.
10. Alejandro Duran et al. Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In *ICPP'09*, 2009.
11. Brandon Lucia et. al. Conflict exceptions: simplifying concurrent language semantics with precise hardware exceptions for data-races. In *ISCA'10*, 2010.
12. Christian Hammer et. al. Dynamic detection of atomic-set-serializability violations. In *ICSE '08*, pages 231–240, 2008.
13. D. H. Bailey et al. The nas parallel benchmarks, 1994.
14. Daniel Marino et. al. Drfx: a simple and efficient memory model for concurrent programming languages. In *PLDI'10*, pages 351–362, 2010.
15. Jr. Robert L. Bocchino et. al. A type and effect system for deterministic parallel java. In *OOPSLA'09*, 2009.
16. P. Charles et. al. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA'05*, pages 519–538, New York, NY, USA, 2005. ACM.
17. Raghavan Raman et. al. Efficient data race detection for async-finish parallelism. In *RV'10*, 2010.
18. Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in cilk programs. In *SPAA'97*, pages 1–11, 1997.
19. Cormac Flanagan and Stephen N. Freund. Type-based race detection for java. In *PLDI '00*, pages 219–232, 2000.
20. Cormac Flanagan and Stephen N Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL '04*, pages 256–267, 2004.
21. Cormac Flanagan and Stephen N. Freund. Fasttrack: efficient and precise dynamic race detection. In *PLDI'09*, pages 121–133, New York, NY, USA, 2009. ACM.
22. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java™ Language Specification*. Addison Wesley, third edition, 2005.

23. Habanero multicore software research project web page. http://habanero.rice.edu, January 2008.
24. Philipp Haller and Martin Odersky. Capabilities for uniqueness and borrowing. In *ECOOP '10*, 2010.
25. Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Race checking by context inference. In *PLDI '04*, pages 1–13, 2004.
26. The Java Grande Forum benchmark suite. http://www.epcc.ed.ac.uk/javagrande/javag.html.
27. Mackale Joyner. *Array Optimizations for High Productivity Programming Languages*. PhD thesis, Rice University, 2008.
28. James R. Larus and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.
29. Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. In *POPL '05*, pages 378–391, 2005.
30. Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. Literace: effective sampling for lightweight data-race detection. In *PLDI '09*, pages 134–143, 2009.
31. Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. In *PLDI '06*, pages 308–319, 2006.
32. Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *PPoPP '03*, pages 167–178, 2003.
33. Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. In *TACAS '05*, pages 93–107, 2005.
34. Michael Roberson and Chandrasekhar Boyapati. A static analysis for automatic detection of atomicity violations in java programs. draft available on second author's websites, 2010.
35. Jr. Robert L. Bocchino, Stephen Heumann, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Adam Welc, and Tatiana Shpeisman. Safe nondeterminism in a deterministic-by-default parallel language. In *POPL'11*, 2011.
36. Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15:391–411, 1997.
37. Jun Shirako, Hironori Kasahara, and Vivek Sarkar. LCPC'08. In *Language Extensions in Support of Compiler Parallelization*, pages 78–94, 2008.
38. Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL '06*, 2006.
39. Jaroslav Ševčík and David Aspinall. On validity of program transformations in the java memory model. In *ECOOP '08*, pages 27–51, 2008.